EFFICIENT DATA STRUCTURES AND PROTOCOLS WITH APPLICATIONS IN SPACE-TIME CONSTRAINED SYSTEMS

By

YAN QIAO

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2014

I would like to dedicate this work to my parents and my husband, without whom I could never reach so far. I love you.

ACKNOWLEDGMENTS

It was such a long journey since the second I decided to pursue a Ph.D. degree, to this moment when I finally get one. I have received so many help, inspiration, and encouragement from my family, friends, teachers, and colleagues. Without them, I can not imagine how dark this path would be.

Looking back to the past few years of my study at University of Florida, it is undoubtedly clear that the first and most important person along this journey was Dr. Shigang Chen, my Ph.D. advisor and my mentor. Thanks to his open-door policy, I can always discuss unsolved issues with him and seek his suggestions. He has spent countless hours working with me even though he doesn't have to. Sometimes he was still up at 3am just to finish reviewing my paper so that I have more time to revise it before the submission deadline. I am so grateful. In life Dr. Chen is a very nice person. Every thanksgiving he invites our group to dinner with his family. It feels so warm especially that my own family was far back in China. He is not just an advisor to my research, a mentor from whom I learn, but also a friend and family.

I want to give my gratitude to my Ph.D. committee members. They are Dr. Jih-Kwon Peir, Dr. Jose Fortes, Dr. Sartaj Sahni and Dr. Yuguang Fang. Thank you for your advice and support during my study at University of Florida. I also would like to thank the fellow students and researchers in my research group. Their names are Ming Zhang, Tao Li, Wen Luo, Zhen Mo, Yian Zhou, Min Chen, Xi Tao, You Zhou, Qingjun Xiao, Zhiping Cai, and Liping Zhang. Thanks for the fresh insights you gave me and all the fun you brought me.

Lastly, I would like give my greatest thanks to my parents. They are such hard working people and they have provided me with the best resources they can get. I also want to thank my husband, who has always believed in me even when I didn't. Thanks for being by my side for all these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

EFFICIENT DATA STRUCTURES AND PROTOCOLS WITH APPLICATIONS IN SPACE-TIME CONSTRAINED SYSTEMS

By

Yan Qiao

May 2014

Chair: Shigang Chen
Major: Computer Engineering

Space-time efficient data structures can benefit many application systems. Space-efficiency describes a data structure that can fit compactly into the memory. The term time-efficiency implies that accessing/updating it requires short period of time. In this work, we cover space-time efficient data structures related to four abstract problems: set membership checking, multi-set membership checking, distributed set joining, and set size estimation. The goal is to improve the space-time efficiency of existing solutions.

We first study an efficient set representation – the Bloom filter. Traditional Bloom filters offer excellent space efficiency, but each lookup requires multiple hash computations and multiple memory accesses. We propose a new family of space-time efficient Bloom filters that provide the same space efficiency, but requires much fewer memory accesses for each lookup. Due to wide applicability, any improvement to the performance of Bloom filters can potentially have a broad impact in many areas of research.

Then we further propose a data structure for representing multi-sets, where a set ID is associated with each member element. Our proposed data structure can reduce the space and computational complexity, yet keep the error ratio in a pretty low level.

Next, we design another Bloom filter variant called adaptive Bloom filter for efficient joining two distributed sets. When two nodes in distributed system exchange their

elements to find out common ones, the Bloom filter can be used to encode element IDs with fewer network overhead. Our design can further reduce the amount of data that is exchanged.

Then, we apply our efficient Bloom filter idea to cyber-physical systems. We show that the space-time domain in computer/network system can be mapped to the time-energy domain of an RFID system. Based on the partitioned Bloom filter and our efficient Bloom filter idea, we design two time-energy efficient protocols for RFID systems. We show how to tweak the data structures to adapt to application-specific features.

Last, we propose two light-weighted protocols to estimate the cardinality of vehicles in a region. Future vehicles may be equipped with wireless devices to form a powerful vehicular peer to peer network. For a stationary wireless network, this problem has been extensively studied. However, it becomes much more challenging for mobile vehicular peer-to-peer networks whose topologies are constantly changing, and the estimation protocol has to be fast to take a useful snapshot of the dynamic network. Our protocols are based on circled random walks and tokened random walks. With simulations under two different mobility models, we show the efficiency of our protocols.

CHAPTER 1
INTRODUCTION

1.1 Background

Space-time efficient data structures and protocols can benefit many application systems. Space-efficiency describes a data structure that can fit compactly into the memory. The term time-efficiency has two dimensions: For a data structure, it implies that accessing/updating it requires short period of time; For a protocol that contains continuous time slots, it means the execution time of the protocol is short. Space-time efficiency implies both space-efficiency and time-efficiency. Space-time efficient data structures and protocols play an important role when space/time has major influence on performance. In this work, we will show several space-time efficient data structure and protocol designs and their applications in such systems.

For example, with modern high-end routers, today's network system is able to process millions of packets per second, and forward each packet in a few clock cycles (40 Gbps for OC-768 and 16.4 Tbps in experimental systems [49]). To keep up with such high throughput, many network functions that deal with real-time packets also have to be implemented in hardware, e.g. traffic measurement, packet scheduling, access control, quality of service, and so on. However, the data associated with the functions may not be stored in DRAM because the bandwidth and delay of DRAM access cannot match the packet throughput at the line speed. Consequently, the recent research trend is to implement network functions in the high-speed on-die cache memory, which is typically SRAM for network processor chips. SRAM has limited size, which is typically a few megabytes. Further increasing on-chip memory to more than 10MB is technically feasible, but it is more expensive and the access time is longer. Off-chip SRAM is larger [124], but it is slower to access and memory bandwidth is constrained by the number of I/O pins. Furthermore, with a limited size, on-chip memory may have to be shared by various integrated functions that are implemented together. For example, traffic monitoring and measurement functions provide critical information for service provision, anomaly detection, capacity planning, accounting and billing in modern computer networks [43, 45, 74, 90, 163]. If these functions are implemented in off-chip DRAM, they will not be able to keep up with today's line speed. As a result, it is highly desired to use space-time efficient data structures in high-speed network systems.

As another example, a distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages [37]. In distributed systems such as the ones mentioned in [75, 82], communication between distributed components is in large scale and happens frequently. It is desirable to make the communication extremely efficient. In this work, we study space-time efficient data structures and protocols in space-time constrained systems in general. We show how these data structures can be applied in but not restricted to high-speed network systems, distributed systems, RFID systems, and vehicular systems.

1.2 Research Scope of This Work

In this work, we first study a general efficient data structure, i.e. the Bloom filter. Bloom filters are efficient data structures for membership check against large data sets [7, 12]. They are widely applied in routing-table lookup [40, 142, 143], per-flow traffic measurement [76], flow label identification [97], firewall design [103], intrusion detection [147], and so on. Traditional Bloom filters offer excellent space efficiency, but each lookup requires multiple hash computations and multiple memory accesses [7, 12, 98]. We propose a new family of space-time efficient Bloom filters that provide the same space efficiency, but requires much fewer memory access for each lookup (Chapter 3). Due to wide applicability, any improvement to the performance of Bloom filters can potentially have a broad impact in many areas of research.

The Bloom filter can compactly encode a single set, but using it alone does not efficiently represent a multi-set, where a set ID is associated with each member element. Inspired by the space-time efficient Bloom filter design, we propose an efficient data structure to encode multi-Set membership. Our design goal is to reduce the space and computational complexity, yet keeping the error ratio caused by false positive in a low level. Chapter 4 shows the detailed design.

Bloom filters have wide application in distributed systems. We design an adaptive Bloom filter for distributively joining two sets in distributed systems (Chapter 5). When the Bloom filter and the compressed Bloom filter were first introduced to this problem, the communication cost was greatly reduced. Our design can further reduce the amount of message that has to be exchanged.

Next, we shift our view from the space domain to the time domain and show how an efficient protocol can benefit cyber-physical systems. In a typical computer/network system, information is stored in memory as 0/1 bits. While in a wireless environment such as an RFID system, information is sent/received via continuous time slots. The length of overall time slots defines the execution time of the protocol. As a result, the space domain in computer/network systems becomes the time domain in RFID systems. Meanwhile, RFID tags send/receive information to/from one or more time slots. The amount of data that a tag send/receive determines its energy expenditure. Therefore, the time domain in computer/network systems (accessing/updating the data structure) can be mapped to the energy domain of RFID tags (sending/receiving information to/from time slots). As a result, designing a time-energy efficient RFID protocol shares the same discipline as implementing a space-time data structure in a computer/network system. Through a specific problem in an RFID system, i.e. how to collect information efficiently from a subset of RFID tags, we show that an efficient protocol design can reduce the battery consumption of RFID tags to a constant level (Chapter 6).

Last, we further discuss applications in vehicular peer to peer networks (Chapter 7). Collecting information from VP2P networks has important applications. One problem is to determine the cardinality of a large VP2P network, i.e., the number of vehicles in the network. For a stationary wireless network, this problem can be trivially solved through a flooding-based query. However, it becomes much more challenging for mobile VP2P networks whose topologies are constantly changing. Meanwhile, the protocol has to run quick in order to provide a useful snapshot of the network. We propose two light-weighted protocols based on circled random walks and tokened random walks for estimating the cardinality. Simulation is performed under two different mobility models: the street model and the random waypoint model.

At the end, we conclude this work and discuss future works.

CHAPTER 2
SYSTEM MODEL, METRICS, RELATED WORKS, AND CONTRIBUTIONS

In this chapter, we introduce the system models, performance metrics, related works and contributions of the this work.

2.1 System Model

In this work, we focus on space-time constrained systems. The applications we used in this work are diversified: including high-speed network system, distributed system, RFID system, and vehicular peer-to-peer system. Despite application specific features, these systems share some similarities: Space and time are precious resources of the systems that directly or indirectly constrains the system performance. Therefore, the space or time or both that a data structure/protocol may use is limited. For instance, a high-speed network system such as that in high-end routers has huge throughput demands and high speed requirements [49]. We make the following reasonable assumptions while designing the data structures for such systems:

- Improving per-operation overhead brings large performance gains.
- Available space is limited in size (e.g. online SRAM is normally smaller than 10MB).
- The system fetches a block of memory (e.g. 32 bits, or 64 bits) into the processor at a time. This is mostly true for modern processors.

More application specific models will be discussed in corresponding chapters.

2.2 Performance Metrics

The performance of the Bloom filter and its many variants is judged based on three criteria: The first one is the processing overhead, which is includes the number of memory accesses and the number of hash operations. The overhead limits the highest throughput that the data structure can support. Because both SRAM and the hash function circuit may be shared among different network functions, it is important for them to minimize their processing overhead in order to achieve good system performance. The second performance criterion is the space requirement. A smaller space requirement reduces the usage of SRAM, saving space for richer online functions. If the data structure is mainly transferred via network, then we can call it as the bandwidth requirement. The third criterion is the correctness. For example, a Bloom filter may mistakenly claim a non-member to be a member due to its lossy encoding method, which is called the false positive ratio. As another example, some variants of Bloom filters may introduce false negatives, which means a member is mistakenly claimed as a non-member. There is a tradeoff between the space requirement and the correctness. We can improve the latter by allocating more memory.

In summary, the performance metrics we consider include:

- Processing Overhead:
  - Memory Access: number of memory access for each membership query.
  - Hash Complexity: number of hash bits required for each membership query.
- Space Requirement:
  - If the data structure is stored in memory, how much memory space it takes.
  - If the data structure is passed as a message, what is the message size.
- Correctness:
  - False Positive Ratio: probability for a non-member being considered as a member.
  - False Negative Ratio: probability for a member being claimed as a non-member.
  - Conflict Classification: probability for a multi-set membership query to return multiple possible set-IDs to a member.
  - Mis-Classification Ratio: probability for a member of a multi-set being recognized as a non-member.

2.3 Related Works

Since the Bloom filter was first proposed in 1970 [7], many variants have emerged to meet different requirements, such as supporting set updates, improving the space-efficiency, memory access efficiency, false positive ratio, and hash complexity of the Bloom filter. A comprehensive survey on Bloom filter variants and their applications can be found in [150]. Other Bloom filter variants were proposed to solve specific problems, e.g. representing multi-sets or sets with multi-attribute members [59, 158].

In the following, we introduce prior works that are related to this dissertation. We divide these works into four categories. The first category aims to improve the space, hash, accuracy, and memory access efficiency of traditional Bloom filters. The second category includes variants of Bloom filters that allow element update or delete. The third category involves the Bloom filter use in the distributed join problem. The last category deals with multi-set membership lookup. Application specific (i.e. RFID, VP2P network) related works will be further discussed in corresponding chapters.

2.3.1 Better Bloom Filter

2.3.1.1 Improving Space Efficiency

Some Bloom filter variants aim to improve the space-efficiency [111, 121, 128]. It has been proven that the Bloom filter is not space-optimal [12, 93, 121]. Pagh et al. designed a new RAM data structure whose space usage is within a lower order term of the optimal value [121]. Porat designed a dictionary data structure that maps keys to values with optimal space [128]. Mitzenmacher proposed the compressed Bloom filters [111], which is suitable for message passing, for example, during web cache sharing. The idea is to use a large, sparse Bloom filter at the sender/receiver, and compress/decompress the filter before/after transmission. We will cover this work in more details in Chapter 5.

2.3.1.2 Reducing False Positive Ratio

Reducing the false positive ratio of the Bloom filter is another subject of research. Lumetta and Mitzenmacher [99] proposed to use two choices (i.e. two sets of hash functions) for each element encoded in the Bloom filter. The idea is to pick the set of hash functions that produce the least number of ones, so as to reduce the false positive ratio. This approach generates fewer false positives than the Bloom filter when the load factor is small, but it requires twice the number of hash bits and twice the number of memory access than the Bloom filter does.

Hao et al. use partitioned hashing to divide members into several groups, each group with a different set of hash functions [46]. Heuristic algorithms are used to find the “best” partition. However, this scheme requires the keys of member elements to compute the final partition. Therefore, it only works well when members are inserted all at once and membership queries are performed afterwards.

Tabataba and Hashemi proposed the Dual Bloom Filter to improve the false positive ratio of a single Bloom filter [148]. The idea is to use two equal sized Bloom filters to encode the set with different sets of hash functions. An element is considered a member only when both filters produce positives. In order to reduce the space usage for storing random numbers for different hash functions, the second Bloom lter encodes ones complemented values of the elements.

Lim et al. use two cross-checking Bloom filters together with traditional Bloom filter to reduce the false positives [92]. It first divides the set S to be encoded to two disjoint subsets A and B ($S = A \cup B$, $A \cap B = \emptyset$). Then they use three Bloom filters to encode S, A, B respectively, denoted as $F(S), F(A), F(B)$. Different hash functions are used to provide cross checking. All three filters are checked during membership lookup. Only when $F(S)$ gives positive result, while $F(A)$ and $F(B)$ do not both give negative results, it claims the element as a member. Their evaluation shows that it improves the false positive of Bloom filters with the same size by magnitudes.

2.3.1.3 Improving Read/Write Efficiency

Other works try to improve the read/write efficiency of the Bloom filter. This category is most related to our work in [**? ?**]. Zhu et al. proposed the hierarchical Bloom filter, which builds a small, less accurate Bloom filter over the large, accurate filter to reduce the number of queries to the large filter [175]. As the small filter has better locality, the total number of memory accesses is reduced. Their method is suitable for the scenarios where large filter access is costive, and element queries are not uniformly distributed [175].

The Partitioned Bloom filter [65, 114] can also improve the read/write efficiency. It divide the memory into k segments. k hash functions are used to map an element to one bit in each segment. If the segments reside in different memory banks, reads/writes of the membership bits can proceed in parallel, so that the overall read/write efficiency is improved.

Kim et al. proposed a parallelised Bloom filter design to reduce the power consumption and increase the computation throughput of Bloom filters [68]. The idea is to transform multiple hash functions of Bloom filters into the multiple stages. When the membership bit from earlier hash function is ‘0’, the query string is not progressed to the next stage.

Chen et al. proposed a new Bloom filter variant that allocates two membership bits to each memory block [30]. This reduces the total number of memory accesses by half. It is equivalent to the Bloom-$k/2$ filter in our Bloom-g family. Our work can be interpreted as a generalization of this work. However, this paper did not further analyze the problem to extremes: $k/2$ memory blocks are accessed for each membership query, which is not necessary as our simulation shows. Besides, this work does not thoroughly study the trade-offs between block sizes, number of hash bits, and false positive ratio.

Canim et al. proposed a two-layer buffered Bloom filter to represent a large set stored in flash memory (a.k.a. Solid State Storage, SSD) [17]. The design considers several important characteristics of flash memory: (1) Data can be read/write by pages. (2) A page write is slower than a page read and data blocks are erased first before they are updated (in-place update problem). (3) Each cell allows a limited number of erase operations in flash memory life cycle. Therefore, it is important to reduce the number of writes to flash memory. In the design of [17], the filter layer in flash consists of several sub-filters, each with the size of a page; while the buffer layer in RAM buffers the read and write operations to each sub-filter, and applies them in bulk when a buffer is full. The buffered Bloom filter adopts an off-line model, where element insertions and queries are deferred and handled in batch.

Debnath et al. adopted an online model in their design of the BloomFlash, which is also a set representative stored in flash memory [38]. The idea is to reduce random write operations as much as possible. First, bit updates are buffered in RAM so that updates to the same page are handled at once. Second, the Bloom filter is segmented to many sub-filters, with each sub-filter occupying one flash page. For each element insertion/query, an additional hash function is evoked to choose which sub-filter to access. In their design, the size of each sub-filter is fixed to the size of a flash page, typically 2KB or 4KB [38]. Comparing to our work, they did not study a more generic case where the memory block size is 32 bits, 64 bits, or more, or how to remedy the false positive ratio induced by imbalanced distribution of membership bits.

Lu et al. proposed a Forest-structured Bloom filter for dynamic set in flash storage [94]. The proposed data structure resides partially in flash and partially in RAM. It partitions flash space into a collection of flash-page sized sub-filters and organizes them into a forest structure. As the set size is increasing, new sub-filters are added to the forest.

2.3.1.4 Reducing Hash Complexity

There are also works that aim to reduce the hash complexity of the Bloom filter. Kirsch and Mitzenmacher have shown that for the Bloom filter, only two hash functions are necessary. Additional hash functions can be produced by a simple linear combination of the output of two hash functions [65]. This gives us an efficient way to produce many hash bits, but it does not reduce the number of hash bits required by the Bloom filter or its variants. There are also other works that design efficient yet well randomized hash functions that are suitable for Bloom filter-like data structures. Since this topic is less related to this work, we do not enumerate them.

2.3.2 Bloom Filter for Dynamic Set

The Bloom filter does not support element deletion or update well, and it requires the number of elements as a prior to optimize the number of hash functions. Almeida et al. proposed the Scalable Bloom Filters for dynamic sets [3], which creates a new Bloom filter when current Bloom filters are “full”. Membership query checks all the filters. Each successive bloom filter has a tighter maximum false positive probability on a geometric progression, so that the compounded probability over the whole series converges to some wanted value. However, this approach uses a lot of space and does not support deletion.

The counting Bloom filter (CBF) [82] addresses the element deletion problem by replacing the bit array with a counter array. Insertions (deletions) to CBF are done by incrementing (decrementing) the counters by 1. Membership query checks the positiveness of the counters. CBF and its variants (e.g. [54, 134]) require multiple times of the space used by a standard Bloom filter, depending on how many bits each counter takes.

Rottenstreich et al. proposed the Variable-Increment Counting Bloom Filter (VI-CBF) [134] to improve the space efficiency of CBF. For each element insertion, counters are updated by a hashed variable increment instead of a unit increment. Then, during a query, the exact value of a counter is considered, not just its positiveness. Due to possible counter overflow, CBF and VI-CBF both introduce false negatives [53].

The Bloom filter with variable-length signatures (VBF) proposed by Lu et al. also enables element deletion [98]. Instead of setting k membership bits, a VBF sets $t(t \leq k)$ bits to one to encode an element. To check the membership of an element, if $q(q \leq t \leq k)$ membership bits are ones, it claims that the element is a member. Deletion is done by setting several membership bits to zero such that remaining ones are less than q. VBF also has false negatives.

Rothenberg et al. achieved false-negative-free deletions by deleting element probabilistically [133]. Their proposed Deletable Bloom filter divides the standard Bloom filter into r regions. It uses a bit map of size r to indicate if there is any “bit collision” (i.e. two members setting the same bit to ‘1’) among member elements in each region. When trying to delete an element, it only resets the bits that are located in collision-free regions. There is a probability that an element can be successfully deleted: when there is at least one membership bit located in a collision-free region. As the load factor increases, the probability of successful deletion decreases. Also, as elements are inserted and deleted, even if the load factor remains the same, the deletion probability still drops. This is because bits the bitmap does not reflect the “real” bit collision when elements previously causing the collision were already deleted. But there is no way to reset bits in the bit map back to ‘0’s.

In same applications, new data is more meaningful than old data, so some data structure deletes old data in first-in-first-out manner. For example, Chang et al. proposed an aging scheme called double buffering where two large buffers are used alternatively [21]. In this scheme, the memory space is divided into two equal-sized Bloom filters called active filter and warm-up filter. The warm up filter is a sub-set of the active filter, which consists only elements that appeared after the active filter is more than half full. When the active filter is full, it is cleared out, and the two filters exchanges roles. Each time the active filter is cleared out, a number of old data are deleted, the size of which is half the capacity of the Bloom filter.

Yoon proposed an active-active scheme to better utilize the memory space of double buffering [162]. The idea is to insert new elements to one filter (instead of possibly two in double buffering) and query both filters. When the filter that elements are currently inserting to becomes full, the other filter is cleared out and the filters switch roles. It can store more data with the same memory size and tolerable false positive rate compared to double buffering.

Deng and Rafiei proposed the Stable Bloom filter to detect duplicates in infinite input data stream [39]. Similar to the Counting Bloom filter, a Stable Bloom filter is an array of counters. When an element is inserted, the corresponding counters are set to the maximum value. The false positive ratio depends on the ratio of zeros, which is a fixed value for a Stable Bloom filter. It achieves this by decrementing some random counters by one whenever a new element is inserted.

Bonomi et al. proposed time-based deletion [8] with a flag associated with each bit in the filter. If a member is accessed, the flags of its corresponding bits are set to ‘1’. At the end of each period, bits with unset flags are reset to zeros, and then all flags are unset to ‘0’s to prepare for the next period. Elements that are not accessed during a period are deleted consequently. However, it has to wait a time period for the deletion to take effect.

2.3.3 Bloom Filter for Distributed Join

Bloom filters are widely used in distributed systems. One application is to efficiently process database joins in distributed setting. Mackert and Lohman proposed Bloomjoin, which sends a Bloom filter to filter out a large portion of unneeded data before transferring real data [104]. Bloomjoin largely reduces the communication cost for distributively joining two sets.

Mullin suggests to use partitioned Bloom filter (PBF) to encode the data, and send one segment at a time [114]. When the data size of filtered out elements is smaller than the size of a PBF segment, it stops sending PBF segments but sends the remaining elements instead. The advantage of this approach over Bloomjoin is that it does not need a false positive ratio as a prior, so that the k that is actually used is more close to the optimal value.

Michael et al. study the intersection of multiple sets [109]. They optimized the join sequence and cache the Bloom filters of key indices in block-partitioned Bloom filters so that rehashing is not necessary.

Ramesh et al. investigate the join problem of multiple sites and propose four Bloomjoin extensions: result merging at participating sites, result merging at user site,caching at a participating site, and caching at coordinator site [131]. They also analyze each extension and construct a query optimizer for selecting the best extension for each query.

[79] and [127] study the use of Bloom filters in distributed join with MapReduce. Phan et al. use Bloom filters and partitioned Bloom filters for both set intersection and set join in MapReduce to save communication cost [127]. Zhang et al. investigate the use of Graphics Processing Units (GPU) in generating the intersection of two lists [168]. To the best of our knowledge, the idea using Bloom filters iteratively with decreasing size and global optimization is under exploit.

2.3.4 Bloom Filter for Multi-Set

A Multi-set is defined as a virtual set that consists of several disjoint sets. To encode a multi-set, we may use one separate Bloom filter to encode each set [21, 56]. However, this method has two problems: First, it is hard to allocate proper amount of memory for each filter if the sizes of sets are unknown. Second, it makes too much memory access as the number of filters required increases with the number of sets. Coded Bloom filters [21, 98] are proposed to significantly cut the number of Bloom filters by encoding the set IDs to binary codes. Each bit of the code is represented by a Bloom filter. However, they are still sensitive to the size of each set, meaning that some filters may be overloaded while others are under loaded.

The combinatorial Bloom filter (COMB) proposed by Hao et al. uses hash function sets instead of separate filters to encode set IDs [57]. It assigns each set S_i a code c_i that has a fixed number θ of ones. For instance, if the code length is 20 bits and $\theta = 5$, there will be $\binom{20}{5} = 15,504$ different valid codes, which can represent as many set IDs. An example of such a code is 00010011000010000001. For each bit in the code, COMB generates a group of k hash functions. To encode a member in S_i, COMB examines c_i

bit by bit. For each bit of value 1, it uses the corresponding group of hash functions to set k bits in the filter to ones. To look up for an element, COMB uses all groups of hash functions, each group corresponding to one bit in the code. For each hash group, COMB performs a classical Bloom filter lookup: Hash the element to k bits in the filter and see if they are all set. If so, the corresponding bit in the code is 1; otherwise, the bit is 0. Again, the value of k determines the probability of false positive (mistakenly concluding a bit as 1 although it should be 0). After knowing all bits in the code, COMB translates the code back to a set ID.

Another related work is the Bloomier filter [22]. It appends each bit in the Bloom filter with a multi-bit entry, which can store a value (set ID). To store an set ID S_e of an element e, it hashes e to k entries in the Bloomier filter, and makes sure that the XOR of these entries is equal to S_e. To perform lookup on e, we simply hash e to its k entries, and their XOR gives its set ID. Once an entry in the filter is used by an element, its value cannot be changed. When we insert a new element e' that belongs to set S_e', after we hash e' to k entries, at least one entry must be unused in order to encode e'. If there is an unused entry, we can set its value such that the XOR of the k elements is equal to S_e'. However, if all k entries are already used, the element cannot be inserted into the filter. To handle insertion failure, separate data structure or infrastructure such as TCAM is used.

Goodrich and Mitzenmacher proposed the Invertible Bloom Lookup Table (IBLT) to encode both keys and values of elements [52]. An IBLT uses a cell (including a counter, a KeySum, and a ValueSum) to replace each bit of a Bloom filter. An element is mapped to k cells just like the Bloom filter does. The counter stores how many elements are mapped to each cell, while KeySum (ValueSum) stores the sum of the keys (values) mapped to the cell. During the membership lookup of an element e, if any mapped cell has a counter of '0', or any mapped cell that has a counter of '1' but $KeySum \neq e$, e is not a member. Otherwise if there is at least one mapped cell with counter equal to '1' and $KeySum = e$, e is a member. Otherwise, it produces “not found”, meaning that it is not sure whether e is encoded or not. A great advantage of IBLT is that it supports element deletion and enumeration of all stored key-value pairs with a bounded failure ratio. However, it need to store keys of elements in the table, which may take too much space when the elements to be encoded have long keys.

There are also some works focusing on encoding keys with multiple occurrence. This problem can also be modeled as a (key, value) pair, where the “value” is a counter indicating how may time the corresponding key appears in the set. One application example of this problem is to measure flow sizes on a router. Each flow id is the “key” and the number of packets in the flow is counted and stored as the “value”. The Spectral Bloom filter[35], Virtual Bit Maps [84] and Counter Braids [97], for example, are proposed to solve this problem. However, these data structures can not be directly used to solve our problem. This is because these approaches normally give an estimated “value” instead of the real one, which translates to an incorrect set-ID. Also, [84] and [97] use offline model instead of online checking model.

2.4 Contributions

The contribution of this work can be summarized as follows:

We summarize the metrics of evaluating the Bloom filter and its variants in network systems scenarios. The trade-off relationship contains space, false positive ratio, memory access, and hash bit requirement. The latter two metrics are often overlooked by existing works.

We conducted a detailed study in one memory access Bloom filter designs and their generalizations. We opened a door for more design choices.

We propose an efficient representation of multi-set.

We design adaptive Bloom filter that is efficient for distributed join operation in distributed systems.

We discover the underlying connection between space-time efficient data structure and time-energy RFID protocol. We apply the efficient Bloom filter ideas to information collection in RFID systems. Our proposed protocol consumes only O(1) tag energy, compared to existing best $O(m)$ protocols, where m is the number of RFID tags from whom information is collected.

We design two efficient cardinality estimation protocols for mobile vehicular peer-to-peer networks. The protocols show good estimation results with relatively low network overhead.

CHAPTER 3
SPACE-TIME EFFICIENT BLOOM FILTERS FOR SET MEMBERSHIP LOOKUP

3.1 Background

The Bloom filters are compact data structures for high-speed online membership check against large data sets [7, 12]. They have wide applications in routing-table lookup [40, 142, 143], online traffic measurement [76, 97], peer-to-peer systems [77, 132], cooperative caching [82], firewall design [103], intrusion detection [147], bioinformatics [106], database query processing [114, 155], stream computing [164], distributed storage system [20], etc [12, 150]. Many network functions require membership check. A firewall may be configured with a large watch list of addresses that are collected by an intrusion detection system. If the requirement is to log all packets from those addresses, the firewall must check each arrival packet to see if the source address is a member of the list. Another example is routing-table lookup. The lengths of the prefixes in a routing table range from 8 to 32. A router can extract 25 prefixes of different lengths from the destination address of an incoming packet, and it needs to determine which prefixes are in the routing tables [40]. Some traffic measurement functions require the router to collect the flow labels [96, 97], such as source/destination address pairs or address/port tuples that identify TCP flows. Each flow label should be collected only once. When a new packet arrives, the router must check whether the flow label extracted from the packet belongs to the set that has already been collected before. As a last example for the membership check problem, we consider the context-based access control (CBAC) function in Cisco routers [47]. When a router receives a packet, it may want to first determine whether the addresses/ports in the packet have a matching entry in the CBAC table before performing the CBAC lookup.

In all of the previous examples, we face the same fundamental problem: For a large data set, which may be an address list, an address prefix table, a flow label set, a CBAC table, or other types of data, we want to check whether a given element belongs to this set or not. If there is no performance requirement, this problem can be easily solved using textbook data structures such as binary search [58] (which stores the set in a sorted array and uses binary search for membership check), or a traditional hash table [41] (which uses linked lists to resolve hash collision). However, these approaches are inadequate if there are stringent speed and memory requirements.

Modern high-end routers and firewalls implement their per-packet operations mostly in hardware. They are able to forward each packet in a couple of clock cycles. To keep up with such high throughput, many network functions that involve per-packet processing also have to be implemented in hardware. However, they cannot store the data structures for membership check in DRAM because the bandwidth and delay of DRAM access cannot match the packet throughput at the line speed. Consequently, the recent research trend is to implement membership check in the high-speed on-die cache memory, which is typically SRAM. The SRAM is however small and must be shared among many online functions. This prevents us from storing a large data set directly in the form of a sorted array or a hash table. A Bloom filter [7] is a bit array that encodes the membership of data elements in a set. Each member in the set is hashed to k bits in the array at random locations, and these bits are set to ones. To query for the membership of a given element, we also hash it to k bits in the array and see if these bits are all ones.

The performance of the Bloom filter and its many variants is judged based on three criteria: The first one is the processing overhead, which is k memory accesses and k hash operations for each membership query. The overhead[1] limits the highest throughput that the filter can support. Because both SRAM and the hash function circuit

[1] In the rest of this chapter, we interchangeably use “processing overhead” and “overhead” without further notification.

may be shared among different network functions, it is important for them to minimize their processing overhead in order to achieve good system performance. The second performance criterion is the space requirement. Minimizing the space requirement to encode each member allows a network function to fit a large set in the limited SRAM space for membership check. The third criterion is the false positive ratio. A Bloom filter may mistakenly claim a non-member to be a member due to its lossy encoding method. There is a tradeoff between the space requirement and the false positive ratio. We can reduce the latter by allocating more memory.

Given the fact that Bloom filters have been applied so extensively in the network research, any improvement to their performance can potentially have a broad impact. We study a data structure, called Bloom-1, which makes just one memory access to perform membership check. Therefore, it can be configured to outperform the commonly-used Bloom filter with constant k. We point out that, due to its high overhead, the traditional Bloom filter is not practical when the optimal value of k is used to achieve a low false positive ratio. We generalize Bloom-1 to Bloom-g, which allows g memory accesses. We show that they can achieve the low false positive ratio of the Bloom filter with optimal k, without incurring the same kind of high overhead. We further generalize Bloom-1 to Bloom-α, which achieves better performance with very small increase in overhead. We perform a thorough analysis to reveal the properties of this family of filters. We discuss how they can be applied for static or dynamic data sets. We also conduct experiments based on a real traffic trace to study the performance of the new filters.

3.2 Bloom-1: One Memory Access Bloom Filter

3.2.1 Bloom Filter

A Bloom filter is a space-efficient data structure for membership check. It includes an array B of m bits, which are initialized to zeros. The array stores the membership information of a set as follows: Each member e of the set is mapped to k bits that are randomly selected from B through k hash functions, $H_i(e)$, $1 \leq i \leq k$, whose range is $[0, m-1)$. [2] To encode the membership information of e, the bits, $B[H_1(e)]$, ..., $B[H_k(e)]$, are set to ones. These are called the "membership bits" in B for the element e. Some frequently-used notations in this chapter can be found in Table 3-1.

To check the membership of an arbitrary element e', if the k bits, $B[H_i(e')]$, $1 \leq i \leq k$, are all ones, e' is considered to be a member of the set. Otherwise, it is not a member.

We can treat the k hash functions logically as a single one that produces $k \log_2 m$ hash bits. For example, suppose m is 2^{20}, $k = 3$, and a hash routine outputs 64 bits. We can extract three 20-bit segments from the first 60 bits of a single hash output and use them to locate three bits in B. Hence, from now on, instead of specifying the number of hash functions required by a filter, we will state "the number of hash bits" that are needed, which is denoted as h.

A Bloom filter doesn't have false negatives, meaning that if it answers that an element is not in the set, it is truly not in the set. The filter however has false positives, meaning that if it answers that an element is in the set, it may not be really in the set. According to [7] and [12], the false positive ratio f_B, which is the probability of mistakenly treating a non-member as a member, is [3]

$$f_B = \left(1 - (1 - \frac{1}{m})^{nk}\right)^k \approx (1 - e^{-\frac{nk}{m}})^k, \tag{3–1}$$

[2] The hash functions can be any random functions as long as the k functions are different, i.e., $H_i \neq H_j$, $i \neq j \in [1, k]$. However, hash function outputs are random and two hash results may happen to be the same, i.e. $\exists i, j, e, i \neq j, H_i(e) = H_j(e)$.

[3] Formula 3–1 is an approximation of the false positive ratio. More precise analysis can be found in [31]. However, when m is large enough ($> \quad 1024$), the approximation error can ge neglected [31].

Table 3-1. Notations in Chapter 3

Symbol	Meaning
n	number of members in a set
B or $B1$	bit array
m	number of bits in the bit array B or $B1$
k	number of membership bits for each element
h	number of hash bits for locating all membership bits
k^*	the optimal value of k that minimizes the false positive ratio of a Bloom filter
$k1^*$	the optimal value of k that minimizes the false positive ratio of a Bloom-1 filter
f_B, f_{B1}, f_{Bg}	false positive ratios of a Bloom filter, a Bloom-1 filter, and a Bloom-g filter, respectively
l	number of words in a bit array
w	number of bits in a word, $m = l \times w$
$B(k = 3)$	Bloom filter that uses three bits to encode each member
$B1(k = 3)$	Bloom-1 filter that uses three bits to encode each member
$B1(h = 3\log_2 m)$	Bloom-1 filter that uses up to $3\log_2 m$ hash bits
$B(\text{optimal } k)$	Bloom filter that uses the optimal number k^* of bits to encode each member to minimize its false positive ratio
$B1(\text{optimal } k)$	Bloom-1 filter that uses the optimal number $k1^*$ of bits to encode each member to minimize its false positive ratio

where n is the number of members in the set. Obviously, the false positive ratio decreases as m increases, and increases as n increases. The optimal value of k (denoted as k^*) that minimizes the false positive ratio can be derived by taking the first-order derivative on (3–1) with respect to k, then letting the right side be zero, and solving the equation. The result is

$$k^* = \ln 2 \times m/n \approx 0.7m/n. \tag{3–2}$$

The optimal k sometimes can be very large. To avoid too many memory accesses, we may also set k as a small constant in practice.

3.2.2 Bloom-1 Filter

To check the membership of an element, a Bloom filter requires k memory accesses. We introduce the Bloom-1 filter, which requires one memory access for membership check. The basic idea is that, instead of mapping an element to k bits randomly selected from the entire bit array, we map it to k bits in a word that is randomly selected from the bit array. A word is defined as a continuous block of bits that can be fetched from the memory to the processor in one memory access. In today's computer architectures, most general-purpose processors fetch words of 32 bits or 64 bits. Specifically designed hardware may access words of 72 bits or longer.

A Bloom-1 filter is an array $B1$ of l words, each of which is w bits long. The total number m of bits is $l \times w$. To encode a member e during the filter setup, we first obtain a number of hash bits from e, and use $\log_2 l$ hash bits to map e to a word in $B1$. It is called the "membership word" of e in the Bloom-1 filter. We then use $k \log_2 w$ hash bits to further map e to k membership bits in the word and set them to ones. The total number of hash bits that are needed is $\log_2 l + k \log_2 w$. Suppose $m = 2^{20}$, $k = 3$, $w = 2^6$, and $l = 2^{14}$. Only 32 hash bits are needed, smaller than the 60 hash bits required in the previous Bloom filter example under similar parameters.

To check if an element e' is a member in the set that is encoded in a Bloom-1 filter, we first perform hash operations on e' to obtain $\log_2 l + k \log_2 w$ hash bits. We use $\log_2 l$ bits to locate its membership word in $B1$, and then use $k \log_2 w$ bits to identify the membership bits in the word. If all membership bits are ones, it is considered to be a member. Otherwise, it is not.

The change from using k random bits in the array to using k random bits in a word may appear simple, but it is also fundamental. An important question is how it will affect the false positive ratio and the processing overhead. A more interesting question is how it will open up new design space to configure various new filters with different performance properties. This is what we will investigate in depth.

The false negative ratio of a Bloom-1 filter is also zero. The false positive ratio f_{B1} of Bloom-1, which is the probability of mistakenly treating a non-member as a member, is derived as follows: Let F be the false positive event that a non-member e' is mistaken for a member. The element e' is hashed to a membership word. Let X be the random variable for the number of members that are mapped to the same membership word. Let x be a constant in the range of $[0, n]$, where n is the number of members in the set. Assume we use fully random hash functions. When $X = x$, the conditional probability for F to occur is

$$Prob\{F|X = x\} \quad = \quad (1 - (1 - \frac{1}{w})^{xk})^k. \tag{3–3}$$

Obviously, X follows the binomial distribution, $Bino(n, \frac{1}{l})$, because each of the n elements may be mapped to any of the l words with equal probabilities. Hence,

$$Prob\{X = x\} = \binom{n}{x}(\frac{1}{l})^x(1 - \frac{1}{l})^{n-x}, \ \ \forall\, 0 \leq x \leq n. \tag{3–4}$$

Therefore, the false positive ratio can be written as

$$\begin{aligned} f_{B1} = Prob\{F\} &= \sum_{x=0}^{n}\Big(Prob\{X = x\} \cdot Prob\{F|X = x\}\Big) \\ &= \sum_{x=0}^{n}\left(\binom{n}{x}(\frac{1}{l})^x(1 - \frac{1}{l})^{n-x}\Big(1 - (1 - \frac{1}{w})^{xk}\Big)^k\right). \end{aligned} \tag{3–5}$$

3.2.3 Impact of Word Size

We first investigate the impact of word size w on the performance of a Bloom-1 filter. If n, l and k are known, we can obtain the optimal word size that minimizes (3–5). However, in reality, we can only decide the amount of memory (i.e., m) to be used for a filter, but cannot choose the word size once the hardware is installed. In the left plot of Figure 3-1, we compute the false positive ratios of Bloom-1 under four word sizes: 32, 64, 72, and 256 bits, when the total amount of memory is fixed at $m = 2^{20}$ and k is set to 3. Note that the number of words, $l = \frac{m}{w}$, is inversely proportional to the word size.

The horizontal axis in the figure is the load factor, $\frac{n}{m}$, which is the number of members stored by the filter divided by the number of bits in the filter. Since most applications require relatively small false positive ratios, we zoom in at the load-factor range of [0, 0.2] for a detailed look in the right plot of Figure 3-1. The computation

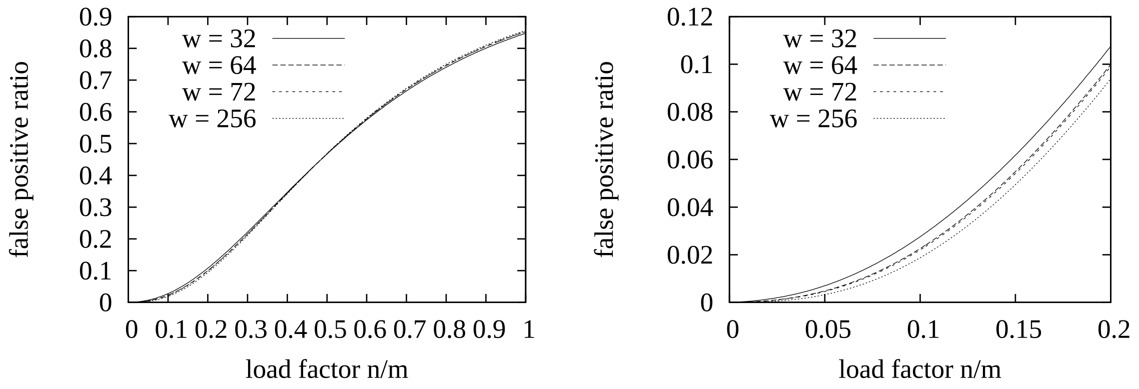

Figure 3-1. False positive ratio with respect to word size w. Left Plot: False positive ratios for Bloom-1 under different word sizes. Right Plot: Magnified false positive ratios for Bloom-1 under different word sizes.

results based on (3–5) show that a larger word size helps to reduce the false positive ratio. In that case, we should simply set $w = m$ for the lowest false positive ratio. However, in practice, w is given by the hardware, not a configurable parameter. Without losing generality, we choose $w = 64$ in our computations and simulations for the rest of this chapter.

3.2.4 Bloom-1 v.s. Bloom with Small k

Although the optimal k always yields the best false positive ratio of the Bloom filter, a small value of k is sometimes preferred to bound the processing overhead in terms of memory accesses and hash operations. We compare the performance of the Bloom-1 filter and the Bloom filter in both scenarios. In this section, we use the Bloom filter with $k = 3$ as the benchmark. Using $k = 4, 5, ...$ produces quantitatively different results, but the qualitative conclusion will stay the same.

We compare the performance of three types of filters: (1) $B(k = 3)$, which represents a Bloom filter that uses three bits to encode each member; (2) $B1(k = 3)$, which represents a Bloom-1 filter that uses three bits to encode each member; (3) $B1(h = 3\log_2 m)$, which represents a Bloom-1 filter that uses the same number of hash bits as $B(k = 3)$ does.

Table 3-2. Query overhead comparison of Bloom-1 filters and Bloom filter with $k = 3$.

Data Structure	Number of Memory Access	Number of Hash Bits $m = 2^{16}$	$m = 2^{20}$	$m = 2^{24}$
$B(k = 3)$	3	48	60	72
$B1(k = 3)$	1	28	32	36
$B1(h = 3\log_2 m)$	1	16–46	20–56	24–72

Let $k1^*$ be the optimal value of k that minimizes the false positive ratio of the Bloom-1 filter in (3–5). For $B1(h = 3\log_2 m)$, we are allowed to use $3\log_2 m$ hash bits, which can encode 3 or more membership bits in the filter. However, it is not necessary to use more than the optimal number $k1^*$ of membership bits. Hence, $B1(h = 3\log_2 m)$ actually uses the k that is closest to $k1^*$ to encode each member.

Table 3-2 presents numerical results of the number of memory accesses and the number of hash bits needed by the three filters for each membership query. They together represent the query overhead and control the query throughput. First, we compare $B(k = 3)$ and $B1(k = 3)$. The Bloom-1 filter saves not only memory accesses but also hash bits. For example, in these examples, $B1(k = 3)$ requires about half of the hash bits needed by $B(k = 3)$. When the hash routine is implemented in hardware (such as CRC [46]), the memory access may become the performance bottleneck, particularly when the filter's bit array is located off-chip or, even if it is on-chip, the bandwidth of the cache memory is shared by other system components. In this case, the query throughput of $B1(k = 3)$ will be three times of the throughput of $B(k = 3)$.

Next, we consider $B1(h = 3\log_2 m)$. Even though it still makes one memory access to fetch a word, the processor may check more than 3 bits in the word for a membership query. If the operations of hashing, accessing memory, and checking membership bits are pipelined and the memory access is the performance bottleneck, the throughput of $B1(h = 3\log_2 m)$ will also be three times of the throughput of $B(k = 3)$.

Finally, we compare the false positive ratios of the three filters in Figure 3-2. The figure show that the overall performance of $B1(k = 3)$ is comparable to that of $B(k = 3)$, but its false positive ratio is worse when the load factor is small. The

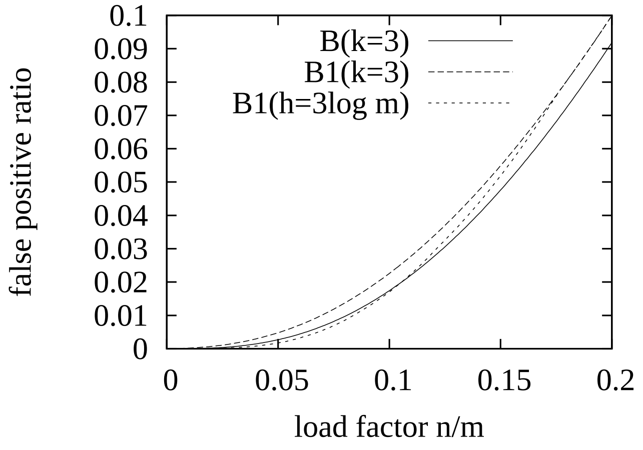

Figure 3-2. Performance comparison in terms of false positive ratio.

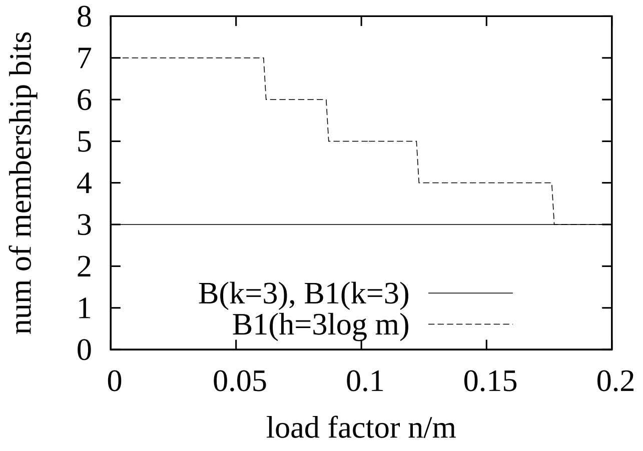

Figure 3-3. Number of membership bits used by the filters.

reason is that concentrating the membership bits in one word reduces the randomness. $B1(h = 3\log_2 m)$ is better than $B(k = 3)$ when the load factor is smaller than 0.1. This is because the Bloom-1 filter requires less number of hash bits on average to locate each membership bit than the Bloom filter does. Therefore, when available hash bits are the same, $B1(h = 3\log_2 m)$ is able to use a larger k than $B(k = 3)$, as shown in Figure 3-3.

3.2.5 Bloom-1 v.s. Bloom with Optimal k

We can reduce the false positive ratio of a Bloom filter or a Bloom-1 filter by choosing the optimal number of membership bits. From (3–2), we find the optimal value k^* that minimizes the false positive ratio of a Bloom filter. From (3–5), we can find the

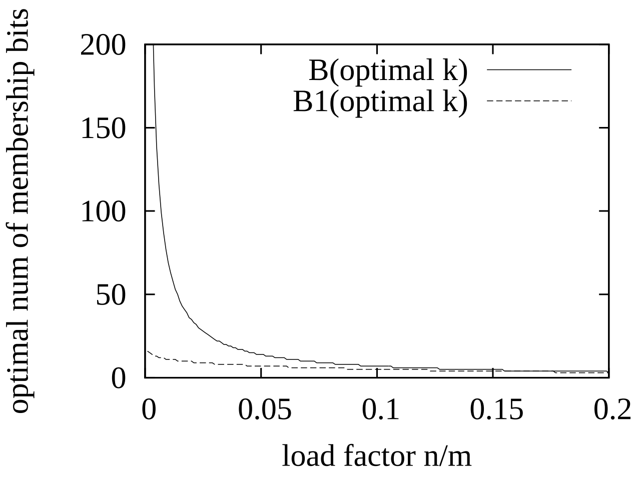

Figure 3-4. Optimal number of membership bits with respect to the load factor.

Table 3-3. Query overhead comparison of Bloom-1 filter and Bloom filter with optimal number of membership bits. Parameters: $m = 2^{20}$ and $w = 64$.

a. Number of memory accesses per query

Data Structure	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	69	35	17	9	4
$B1$(optimal k)	1	1	1	1	1

b. Number of hash bits per query

Data Structure	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	1380	700	340	180	80
$B1$(optimal k)	80	74	62	50	38

optimal value $k1^*$ that minimizes the false positive ratio of a Bloom-1 filter. The values of k^* and $k1^*$ with respect to the load factor are shown in Figure 3-4. When the load factor is less than 0.1, $k1^*$ is significantly smaller than k^*.

We use B(optimal k) to denote a Bloom filter that uses the optimal number k^* of membership bits, and $B1$(optimal k) to denote a Bloom-1 filter that uses the optimal number $k1^*$ of membership bits.

To make the comparison more concrete, we present the numerical results of memory access overhead and hashing overhead with respect to the load factor in Table 3-3. For example, when the load factor is 0.04, the Bloom filter requires 17

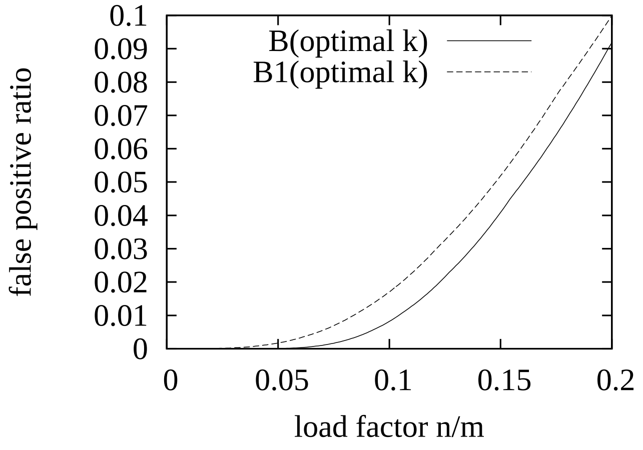

Figure 3-5. False positive ratios of the Bloom filter and the Bloom-1 filter with optimal k.

memory accesses and 340 hash bits to minimize its false positive ratio, whereas the Bloom-1 filter requires only 1 memory access and 62 hash bits. In practice, the load factor is determined by the application requirement on the false positive ratio. If an application requires a very small false positive ratio, it has to choose a small load factor.

Next, we compare the false positive ratios of $B(\text{optimal } k)$ and $B1(\text{optimal } k)$ with respect to the load factor in Figure 3-5. The Bloom filter has a much lower false positive ratio than the Bloom-1 filter. On one hand, we must recognize the fact that, as shown in Table 3-3, the overhead for the Bloom filter to achieve its low false positive ratio is simply too high to be practical. On the other hand, it raises a challenge for us to improve the design of the Bloom-1 filter so that it can match the performance of the Bloom filter at much lower overhead. In the next section, we generalize the Bloom-1 filter to allow performance-overhead tradeoff, which provides flexibility for practitioners to achieve a lower false positive ratio at the expense of modestly higher query overhead.

3.3 Bloom-g: A Generalization of Bloom-1

3.3.1 Bloom-g Filter

As a generalization of Bloom-1 filter, a Bloom-g filter maps each member e to g words instead of one, and spreads its k membership bits evenly in the g words. More specifically, we use $g \log_2 l$ hash bits derived from e to locate g membership words, and then use $k \log_2 w$ hash bits to locate k membership bits. The first one or multiple words are each assigned $\lceil \frac{k}{g} \rceil$ membership bits, and the remaining words are each assigned $\lfloor \frac{k}{g} \rfloor$ bits, so that the total number of membership bits is k.

To check the membership of an element e', we have to access g words. Hence the query overhead includes g memory accesses and $g \log_2 l + k \log_2 w$ hash bits.

The false negative ratio of a Bloom-g filter is zero and the false positive ratio f_{Bg} of the Bloom-g filter, is derived as follows: Each member encoded in the filter randomly selects g membership words. There are n members. Together they select gn membership words (with replacement). These words are called the "encoded words". In each encoded word, $\frac{k}{g}$ bits are randomly selected to be set as ones during the filter setup. To simplify the analysis, we use $\frac{k}{g}$ instead of taking the ceiling or floor.

Now consider an arbitrary word D in the array. Let X be the number of times this word is selected as an encoded word during the filter setup. Assume we use fully random hash functions. When any member randomly selects a word to encode its membership, the word D has a probability of $\frac{1}{l}$ to be selected. Hence, X is a random number that follows the binomial distribution $Bino(gn, \frac{1}{l})$. Let x be a constant in the range $[0, gn]$.

$$Prob\{X = x\} = \binom{gn}{x} (\frac{1}{l})^x (1 - \frac{1}{l})^{gn-x}. \tag{3–6}$$

Consider an arbitrary non-member e'. It is hashed to g membership words. A false positive happens when its membership bits in each of the g words are ones. Consider an arbitrary membership word of e'. Let F be the event that the $\frac{k}{g}$ membership bits of e' in this word are all ones. Suppose this word is selected for x times as an encoded word during the filter setup. We have the following conditional probability:

$$Prob\{F | X = x\} = \big(1 - (1 - \frac{1}{w})^{x\frac{k}{g}}\big)^{\frac{k}{g}}. \tag{3–7}$$

The probability for F to happen is

$$\begin{aligned} Prob\{F\} &= \sum_{x=0}^{gn}\Big(Prob\{X=x\}\cdot Prob\{F|X=x\}\Big) \\ &= \sum_{x=0}^{gn}\left(\binom{gn}{x}\cdot(\frac{1}{l})^{x}\cdot(1-\frac{1}{l})^{gn-x}\cdot\big(1-(1-\frac{1}{w})^{x\frac{k}{g}}\big)^{\frac{k}{g}}\right) \end{aligned} \tag{3–8}$$

Element e' has g membership words. Hence, the false positive ratio is

$$f_{Bg} = (Prob\{F\})^{g} = \left[\sum_{x=0}^{gn}\left(\binom{gn}{x}\cdot(\frac{1}{l})^{x}\cdot(1-\frac{1}{l})^{gn-x}\cdot\big(1-(1-\frac{1}{w})^{x\frac{k}{g}}\big)^{\frac{k}{g}}\right)\right]^{g}. \tag{3–9}$$

When $g = k$, exactly one bit is set in each membership word. This special Bloom-k is identical to a Bloom filter with k membership bits. (Note that Bloom-k may happen to pick the same membership word more than once. Hence, just like a Bloom filter, Bloom-k allows more than one membership bit in a word.) To prove this, we first let $g = k$, and (3–9) becomes,

$$\begin{aligned} f_{Bk} &= \left[\sum_{x=0}^{kn}\left(\binom{kn}{x}\cdot(\frac{1}{l})^{x}\cdot(1-\frac{1}{l})^{kn-x}\cdot\big(1-(1-\frac{1}{w})^{x}\big)\right)\right]^{k} \\ &= \left[1-\sum_{x=0}^{kn}\left(\binom{kn}{x}\cdot(\frac{1}{l}\cdot(1-\frac{1}{w}))^{x}\cdot(1-\frac{1}{l})^{kn-x}\right)\right]^{k} \\ &= \big(1-(1-\frac{1}{lw})^{kn}\big)^{k}. \end{aligned} \tag{3–10}$$

As $m = lw$, we have $f_{Bk} = f_{B}$. In other words, a Bloom-k filter is identical to a Bloom filter.

3.3.2 Bloom-g v.s. Bloom with Small k

We compare the performance and overhead of the Bloom-g filters and the Bloom filter with $k = 3$. Because the overhead of Bloom-g increases with g, it is highly desirable to use a small value for g. Hence, we focus on Bloom-2 and Bloom-3 filters as typical examples in the Bloom-g family.

We compare the following filters: (1) $B(k = 3)$, the Bloom filter with $k = 3$; (2) $B2(k = 3)$, the Bloom-2 filter with $k = 3$; (3) $B2(h = 3\log_2 m)$, the Bloom-2 filter that is

Table 3-4. Query overhead comparison of Bloom-2 filter and Bloom filter with $k = 3$.

Data Structure	Number of Memory Access	Number of Hash Bits $m = 2^{16}$	$m = 2^{20}$	$m = 2^{24}$
$B(k = 3)$	3	48	60	72
$B2(k = 3)$	2	38	46	54
$B2(h = 3\log_2 m)$	2	32–44	40–58	48–72

allowed to use the same number of hash bits as $B(k = 3)$ does. In this subsection, we do not consider Bloom-3 because it is equivalent to $B(k = 3)$, as we have discussed in Section 3.3.1.

From (3–9), when $g = 2$, we can find the optimal value of k, denoted as $k2^*$, that minimizes the false positive ratio. Similar to $B1(h = 3\log_2 m)$, the filter $B2(h = 3\log_2 m)$ uses the k that is closest to $k2^*$ under the constraint that the number of required hash bits is less than or equal to $3\log_2 m$.

Table 3-4 compares the query overhead of three filters. The Bloom filter, $B(k = 3)$, needs 3 memory accesses and $3\log_2 m$ hash bits for each membership query. The Bloom-2 filter, $B2(k = 3)$, requires 2 memory accesses and $2\log_2 l + 3\log_2 w$ hash bits. It is easy to see $2\log_2 l + 3\log_2 w = 3\log_2 m - \log_2 l < 3\log_2 m$. Hence, $B2(k = 3)$ incurs fewer memory accesses and fewer hash bits than $B(k = 3)$. On the other hand, $B2(h = 3\log_2 m)$ uses the same number of hash bits as $B(k = 3)$ does, but makes fewer memory accesses.

Figure 3-6 presents the false positive ratios of $B(k = 3)$, $B2(k = 3)$ and $B2(h = 3\log_2 m)$; Figure 3-7 shows the number of membership bits used by the filters. The figures show that $B(k = 3)$ and $B2(k = 3)$ have comparable false positive ratios in load factor range of [0.1, 1], whereas $B2(h = 3\log_2 m)$ performs better in load factor range of [0.00005, 1]. For example, when the load factor is 0.04, the false positive ratio of $B(k = 3)$ is 1.5×10^{-3} and that of $B2(k = 3)$ is 1.6×10^{-3}, while the false positive ratio of $B2(h = 3\log_2 m)$ is 3.1×10^{-4}, about one fifth of the other two. Considering that $B2(h = 3\log_2 m)$ uses the same number of hash bits as $B(k = 3)$ but only 2 memory

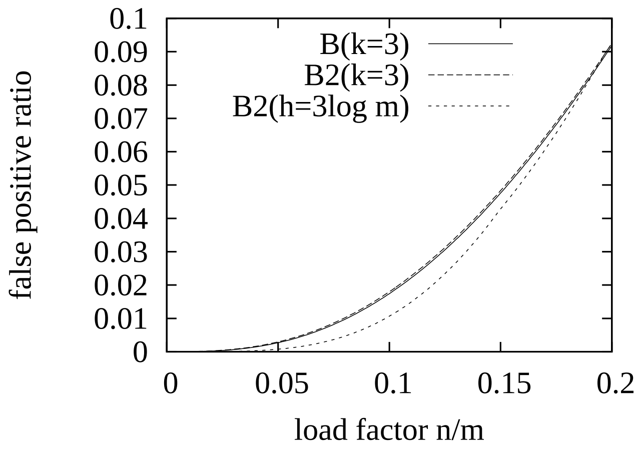

Figure 3-6. False positive ratios of Bloom filter and Bloom-2 filter.

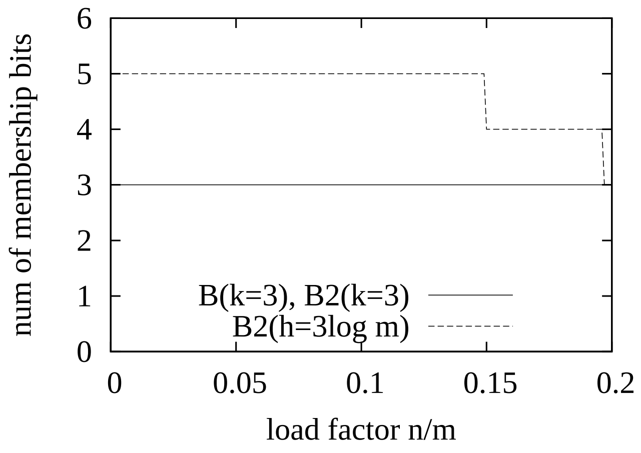

Figure 3-7. Number of membership bits used by the filters.

accesses per query, it is a very useful substitute of the Bloom filter to build fast and accurate data structures for membership check.

3.3.3 Bloom-g v.s. Bloom with Optimal k

We now compare the Bloom-g filters and the Bloom filter when they use the optimal numbers of membership bits determined from (3–1) and (3–9), respectively. We use $B(\text{optimal } k)$ to denote a Bloom filter that uses the optimal number k^* of membership bits to minimize the false positive ratio. We use $Bg(\text{optimal } k)$ to denote a Bloom-g filter that uses the optimal number kg^* of membership bits, where $g = 1$, 2 or 3. Figure 3-8

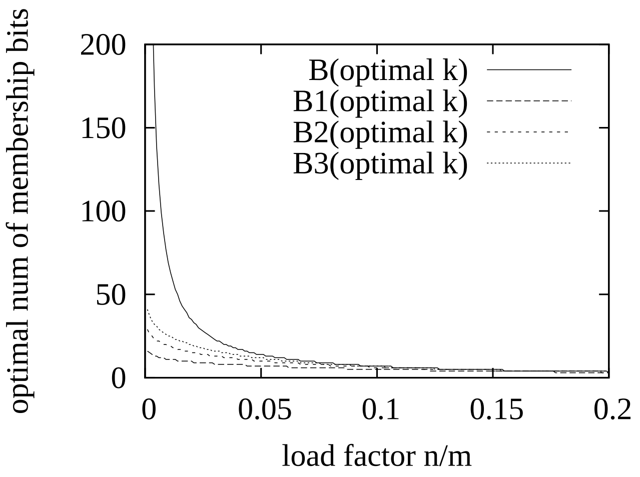

Figure 3-8. Optimal number of membership bits with respect to the load factor.

Table 3-5. Query overhead comparison of Bloom filter and Bloom-g filter with optimal k. Parameters: $m = 2^{20}$ and $w = 64$.

a. Number of memory accesses per query

Data Structure	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	69	35	17	9	4
$B1$(optimal k)	1	1	1	1	1
$B2$(optimal k)	2	2	2	2	2
$B3$(optimal k)	3	3	3	3	3

b. Number of hash bits per query

Data Structure	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	1380	700	340	180	80
$B1$(optimal k)	80	74	62	50	38
$B2$(optimal k)	142	118	94	70	52
$B3$(optimal k)	198	162	126	90	66

compares their numbers of membership bits (i.e., k^*, $k1^*$, $k2^*$ and $k3^*$). It shows that the Bloom filter uses many more membership bits when the load factor is small.

Next we compare the filters in terms of query overhead. For $1 \leq g \leq 3$, Bg(optimal k) makes g memory accesses and uses $g \log_2 l + kg^* \log_2 w$ hash bits per membership query. Numerical comparison is provided in Table 3-5. In order to achieve a small false positive ratio, one has to keep the load factor small, which means that B(optimal k) will have to make a large number of memory accesses and use a

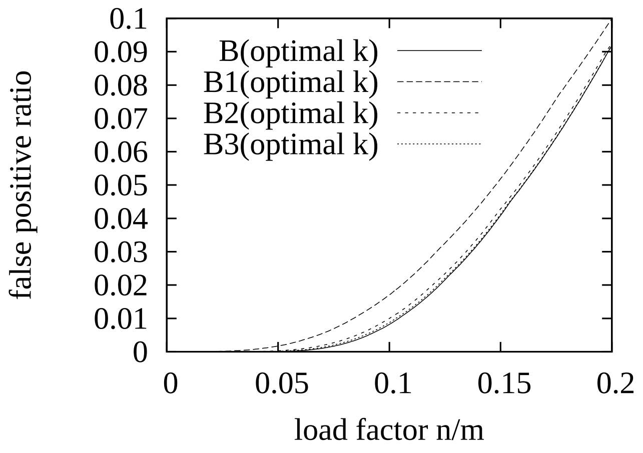

Figure 3-9. False positive ratios of Bloom and Bloom-g with optimal k.

large number of hash bits. For example, when the load factor is 0.08, it makes 9 memory accesses with 180 hash bits per query, whereas the Bloom-1, Bloom-2 and Bloom-3 filters make 1, 2 and 3 memory accesses with 50, 70 and 90 hash bits, respectively. When the load factor is 0.02, it makes 35 memory accesses with 700 hash bits, whereas the Bloom-1, Bloom-2 and Bloom-3 filters make just 1, 2 and 3 memory accesses with 74, 118 and 162 hash bits, respectively.

Figure 3-9 presents the false positive ratios of the Bloom and Bloom-g filters. As we already showed in Section 3.2.5, $B1$(optimal k) performs worse than B(optimal k). However, the false positive ratio of $B2$(optimal k) is very close to that of B(optimal k). Furthermore, the curve of $B3$(optimal k) is almost entirely overlapped with that of B(optimal k) for the whole load-factor range. The results indicate that with just two memory accesses per query, $B2$(optimal k) works almost as good as B(optimal k), even though the latter makes many more memory accesses.

3.3.4 Discussion

The mathematical and numerical results demonstrate that Bloom-2 and Bloom-3 have smaller false positive ratios than Bloom-1 at the expense of larger query overhead. Below we give an intuitive explanation: Bloom-1 uses a single hash to map each member to a word before encoding. It is well known that a single hash cannot achieve an evenly distributed load; some words will have to encode much more members than others, and some words may be empty as no members are mapped to them. This uneven distribution of members to the words is the reason for larger false positives. Bloom-2 maps each member to two words and splits the membership bits among the words. Bloom-3 maps each member to three words. They achieve better load balance such that most words will each encode about the same number of membership bits. This helps them improve their false positive ratios.

3.3.5 Using Bloom-g in a Dynamic Environment

In order to compute the optimal number of membership bits, we must know the values of n, m, w, and l. The value of m, w and l are known once the amount of memory for the filter is allocated. The value of n is known only when the filter is used to encode a static set of members. In practice, however, the filter may be used for a dynamic set of members. For example, a router may use a Bloom filter to store a watch list of IP addresses, which are identified by the intrusion detection system as potential attackers. The router inspects the arrival packets and logs those packets whose source addresses belong to the list. If the watch list is updated once a week or at the midnight of each day, we can consider it as a static set of addresses during most of the time. However, if the system is allowed to add new addresses to the list continuously during the day, the watch list becomes a dynamic set. In this case, we do not have a fixed optimal value of k^* for the Bloom filter. One approach is to set the number of membership bits to a small constant, such as three, which limits the query overhead. In addition, we should also set the maximum load factor to bound the false positive ratio. If the actual load factor exceeds the maximum value, we allocate more memory and set up the filter again in a larger bit array.

The same thing is true for the Bloom-g filter. For a dynamic set of members, we do not have a fixed optimal number of membership bits, and the Bloom-g filter will also have to choose a fixed number of membership bits. The good news for the Bloom-g filter

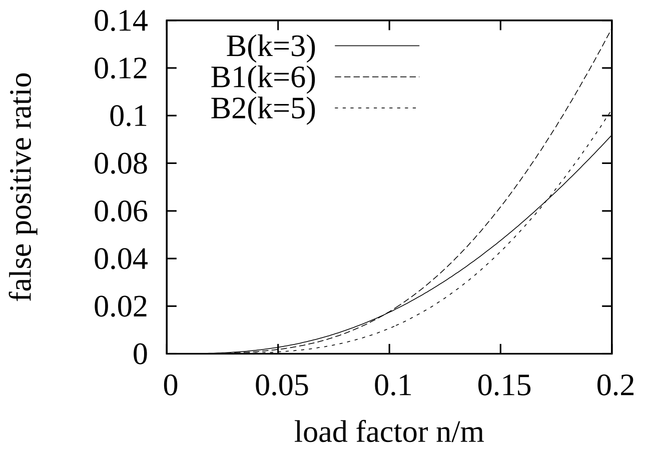

Figure 3-10. False positive ratios of the Bloom filter with $k = 3$, the Bloom-1 filter with $k = 6$, and the Bloom-2 filter with $k = 5$.

is that its number of membership bits is unrelated to its number of memory accesses. The flexible design allows it to use more membership bits while keeping the number of memory accesses small or even a constant one.

Comparing with the Bloom filter, we may configure a Bloom-g filter with more membership bits for a smaller false positive ratio, while in the mean time keeping both the number of memory accesses and the number of hash bits smaller. Imagine a filter of 2^{20} bits is used for a dynamic set of members. Suppose the maximum load factor is set to be 0.1 to ensure a small false positive ratio. Figure 3-10 compares the Bloom filter with $k = 3$, the Bloom-1 filter with $k = 6$, and the Bloom-2 filter with $k = 5$. As new members are added over time, the load factor increases from zero to 0.1. In this range of load factors, the Bloom-2 filter has significantly smaller false positive ratios than the Bloom filter. When the load factor is 0.04, the false positive ratio of Bloom-2 is just one fourth of the false positive ratio of Bloom. Moreover, it makes fewer memory accesses per membership query. The Bloom-2 filter uses 58 hash bits per query, and the Bloom filter uses 60 bits. The false positive ratios of the Bloom-1 filter are close to or slightly better than those of the Bloom filter. It achieves such performance by making just one memory access per query and uses 50 hash bits.

3.3.6 Experiment

We further evaluate the Bloom-g filters through experiments using real network traces.

3.3.6.1 Experiment Setup

Imagine that an intrusion detection system maintains a watch list consisting of previously-identified external sources, which exhibit suspicious behaviors that match the patterns of worm attacks, DDoS attacks, scanning or reconnaissance. The intrusion detection system wants to further analyze the packets from these hosts in order to capture the real offenders. It needs to match the source addresses of the incoming packets against the watch list and log the ones whose addresses are members of the list. While the whole watch list is stored in a hash table located in DRAM, it is also encoded in a Bloom or Bloom-g filter whose small size can fit in SRAM in order to keep up with the line speed. Suppose the watch list is updated once a day. It can be treated as a static list during operation.

We obtain inbound packet header traces from the main gateway at our campus. They contain 2,064,081 distinct source IP addresses and 2,192,707 distinct destination addresses. We first generate 10 watch lists from the trace, each with 25,000 randomly selected source addresses. We then feed the traffic traces through the filters as well as the hash table to identify the matching source addresses and compute false positive ratios based on the number of matches by the filters and the number of matches by the hash table. Results are averaged among the 10 watch lists.

Each experiment consists of two phases: the initialization phase and the execution phase. In the initialization phase, we set up the Bloom/Bloom-g filters, as well as the hash table, by using a watch list of addresses. We always allocate the same amount of memory to the Bloom and Bloom-g filters for fair comparison. We use six filters. They are (a) two Bloom filters: $B(k = 3)$ and $B(\text{optimal } k)$, and (b) four Bloom-g filters:

Table 3-6. Query overhead comparison of Bloom filter with $k = 3$ and Bloom-g filter with $h = 3\log_2 m$. Parameters: $n = 25,000$ and $w = 64$.

a. Number of memory accesses per query

Data Structure	# Memory Access
$B(k=3)$	3
$B1(h=3\log_2 m)$	1
$B2(h=3\log_2 m)$	2

b. Number of hash bits per query

Data Structure	m		
	125Kb	250Kb	500Kb
$B(k=3)$	51	54	57
$B1(h=3\log_2 m)$	29	42	55
$B2(h=3\log_2 m)$	40	54	56

$B1(h = 3\log_2 m)$, $B2(h = 3\log_2 m)$, $B1(\text{optimal } k)$, and $B2(\text{optimal } k)$. Their definitions can be found in Section 3.2 and Section 3.3.

In the execution phase, we perform a membership query in each filter for the source address of each packet. If a filter claims that it is a member but the source is not found in the hash table, it is a false positive.

We vary the size m of the filters from 125Kb to 500Kb, which translates into 5 to 20 bits per member in the watch list, and load factor from 0.2 to 0.05. Let $w = 64$. $l = \frac{m}{w}$ varies from 2,000 to 8,000.

3.3.6.2 Performance Comparison of Bloom and Bloom-g

First, we compare the performance of $B(k = 3)$, $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$. Table 3-6 presents the query overhead of the Bloom filter and the Bloom-g filters. Note that Table 3-6b is computed based on the number of membership bits (i.e. k) for each filter. In our experiment, $B1(h = 3\log_2 m)$ uses 7 membership bits, and $B2(h = 3\log_2 m)$ uses 5. $B(k = 3)$, $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$ requires 3, 1 and 2 memory accesses respectively for each query. $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$ also require less hash bits than $B(k = 3)$. For example, when $m = 125$Kb, $B(k = 3)$ requires 51 hash bits per query, while $B1(h = 3\log_2 m)$ and $B2(h = 3\log_2 m)$ only need 29 and 40 hash bits respectively for each query.

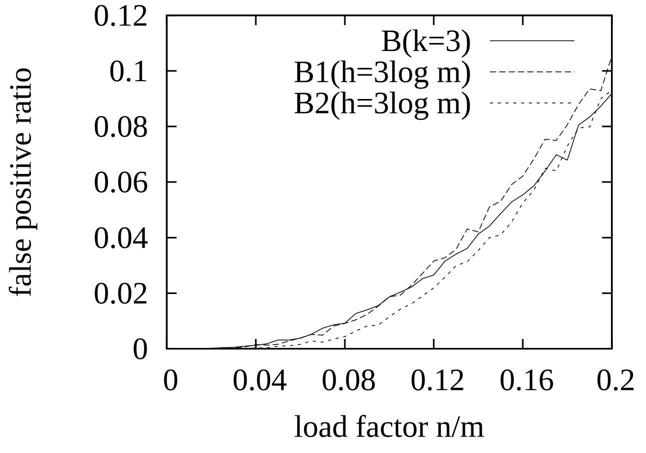

Figure 3-11. False positive ratios of Bloom with $k = 3$ and Bloom-g with $h = 3\log_2 m$ in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

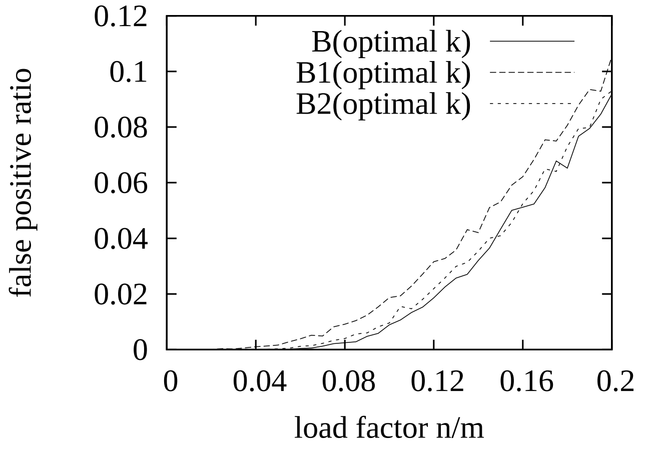

Figure 3-12. False positive ratios of Bloom and Bloom-g with optimal k in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

Figure 3-11 presents the false positive ratios of the filters with respect to the amount of memory m. As our theoretical analysis has predicted, the false positive ratio of $B2(h = 3\log_2 m)$ is smaller than that of $B(k = 3)$. $B1(h = 3\log_2 m)$ also has a slightly smaller false positive ratio than $B(k = 3)$ when m is larger than 250Kb. When m is smaller than 250Kb, it yields a slightly larger false positive ratio than $B(k = 3)$. Given that the throughput of $B1(h = 3\log_2 m)$ can potentially be up to three times that of $B(k = 3)$, it is an attractive option in practice despite of its slightly higher false positive ratio.

Table 3-7. Query overhead comparison of Bloom filter and Bloom-g filter with optimal k. Parameters: $n = 25,000$ and $w = 64$.

a. Number of memory accesses per query

Data Structure	m		
	125Kb	250Kb	500Kb
B(optimal k)	3	7	14
$B1$(optimal k)	1	1	1
$B2$(optimal k)	2	2	2

b. Number of hash bits per query

Data Structure	m		
	125Kb	250Kb	500Kb
B(optimal k)	51	126	266
$B1$(optimal k)	29	42	55
$B2$(optimal k)	40	60	86

Next, we compare B(optimal k), $B1$(optimal k) and $B2$(optimal k). We use the optimal k as shown in Figure 3-8 for each filter. Table 3-7 presents the query overhead of the Bloom filter and Bloom-g filters. When m varies from 125Kb to 500Kb, the query overhead of B(optimal k) increases dramatically. When $m = 500$Kb, for example, B(optimal k) requires 14 memory accesses and 266 hash bits for each query, making it impractical. In comparison, $B1$(optimal k) requires only one memory access and 55 hash bits, while $B2$(optimal k) requires just two memory accesses and 86 hash bits. They remain practical solutions under this setting.

Figure 3-12 presents the false positive ratios of the filters with respect to m. These results match the theoretical analysis we give in Section 3.3.3. The false positive ratio of $B2$(optimal k) is comparable to that of B(optimal k) even though its query overhead is much smaller. It is an excellent candidate for applications that require a very low false positive ratio. The false positive ratio of $B1$(optimal k) is larger than that of $B2$(optimal k), but has even smaller overhead, which represents a performance-overhead tradeoff.

3.4 Bloom-α: Another Generalization of Bloom-1

3.4.1 Motivation

From Figure 3-6, we see that Bloom-2 is comparable to the Bloom filter in terms of false positive ratio when $k = 3$. However, it performs much better if it uses the same number of hash bits as Bloom does. From Figure 3-9, when the optimal number of membership bits is used, Bloom-2 is almost as good as Bloom and better than Bloom-1. The downside is that Bloom-2 needs two memory accesses per query, whereas Bloom-1 needs just one. Our goal is to make a performance tradeoff between Bloom-1 and Bloom-2 such that false positive ratio is close to what Bloom-2 has, whereas overhead is close to what Bloom-1 has.

We have briefly touched upon the reason why Bloom-2 outperforms Bloom-1 in Section 3.3.4. Now we give a closer look. Recall that a Bloom-1 filter uses an array of words. Let's define the "load" of a word to be the number of members that are encoded in the word. False positive ratios incurred in different words may vary. Naturally, the false positive ratio of a heavily-loaded word will be higher than that of a lightly-loaded word. This is confirmed by our simulation that keeps track of which word each false positive occurs in. We classify the words into groups based on their load values, and find the average false positive ratio for each group by simulation. The results are shown in Figure 3-13. For words whose loads are 6 or smaller, false positive ratios are very small. However, false positive ratio grows superlinearly. When the loads are 10 or greater, false positive ratios are very high. Clearly, reducing the number of heavily-loaded words helps reduce the overall false positive ratios. The approach adopted by Bloom-2 maps each member to two membership words. Each word only carries half of the membership bits. In other words, each word only encodes half of the member. Consider a heavily loaded word that serves as the membership word for 14 members. In Bloom-2, since the word encodes half of each member, its load is reduced from 14 to 7. Figure 3-14 shows the

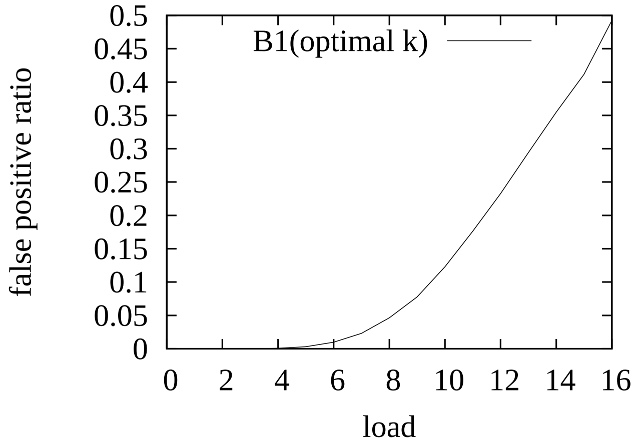

Figure 3-13. False positive ratio of Bloom-1 with respect to load of words. Parameters: $m = 2^{20}$, $w = 64$, load factor $n/m = 0.1$.

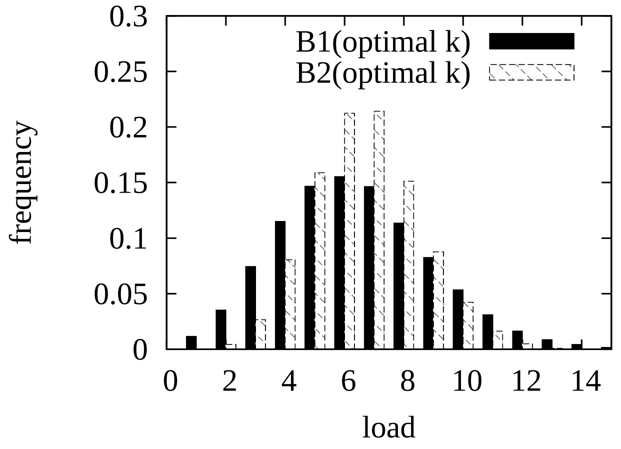

Figure 3-14. Frequency distribution of words with respect to load values. Parameters: $m = 2^{20}$, $w = 64$, load factor $n/m = 0.1$ for both filters.

frequency distribution of words with respect to load values. Fewer words have heavy loads of 10 or higher in Bloom-2 than Bloom-1. That is why Bloom-2 performs better.

However, two membership words require two memory accesses for each query. What about only some members have two membership words while others have one? In this case, some queries need two memory accesses, but others needs just one. The average number of memory access per query will be between 1 and 2. Intuitively, for a lightly loaded word, we want its encoded members to have just one membership word. However, for a heavily loaded word, we want all or some of its encoded members to have additional membership words so that some of their membership bits will be moved elsewhere.

3.4.2 Bloom-α Filter

A Bloom-α filter (denoted as $B\alpha$) is also an array of l words, each of which is w bits long. The first $w-1$ bits in each word are used to encode members of a set, and the last bit is a flag to indicate whether members mapped to this word have second membership words. Initially, all bits (including flag bits) are zeros. The setup procedure of a Bloom-α filter consists of two phases. In the first phase, we map elements in the set to the words in the same way as we do for Bloom-1. Each element is mapped to and encoded in exactly one word. We keep track of the elements that are encoded in each word. In the second phase, we pick a word W_1 that has the largest number of encoded members. We then "split" these members: for each member, we use $\log_2 l$ of its hash bits to locate a second word W_2 and move its last $\lfloor k/2 \rfloor$ membership bits from W_1 to W_2. Finally the flag bit in W_1 is set to one; we call W_1 a "split word". We keep splitting the heaviest loaded word until a certain pre-defined *fraction α of members* are split, i.e., the number of members having two membership words reaches αn. As split words can not be split again, this process will terminate as soon as enough members are split.

The construction of Bloom-α ensures that, if the flag of a word is one, the members mapped to the word must have two membership words; if the flag is zero, the members have one membership word. Bloom-1 is a special case of Bloom-α with $\alpha = 0$.

To perform membership check on an element e', we first hash the element to locate its membership word W_1. If the flag of the word is zero, we check k membership bits in this word. If all these bits are ones, e' is a member of the set; Otherwise, it is not. If the flag of the word is one, we check the first $\lceil k/2 \rceil$ membership bits in W_1. If any of these bits is zero, we are sure that e' is not a member and there is no need to check the second word. However, if all these bits are ones, we find the second membership word

Table 3-8. Query overhead comparison of Bloom-1, Bloom-2 and Bloom-α filter.

Data Structure	# Memory Access	# Hash Bits
$B1(k=3)$, $B1(\text{optimal } k)$	1	$\log_2 l + k \log_2 w$
$B2(k=3)$, $B2(\text{optimal } k)$	2	$2 \log_2 l + k \log_2 w$
$B\alpha(k=3)$, $B\alpha(\text{optimal } k)$	$\leq 1+\alpha$	$2 \log_2 l + k \log_2 w$

by using additional hash bits and check the remaining $\lfloor k/2 \rfloor$ membership bits in that word. We claim that e' is a member of the set only if all those bits are also ones.

We analyze the query overhead in Table 3-8. Consider an arbitrary element e'. If e' is mapped to a word whose flag is zero, it takes one memory access. If e' is mapped to a split word whose flag is one, it takes one or two memory accesses, depending on whether any of the first $\lceil k/2 \rceil$ membership bits is zero. The chance for e' to be mapped to any word is equal. Hence, the probability for e' to be mapped to a split word is equal to the fraction (or percentage) of words that are split. Considering that split words are most heavily loaded, the average load of these words is larger than (or equal to, only if members are evenly distributed) that of the whole word array. In order to split α fraction of members, splitting no more than α fraction of words is enough. Stringent proof is easy to make and is omit. Therefore, the average number of memory accesses per query is bounded by $1 \times (1-\alpha) + 2 \times \alpha = 1 + \alpha$. Our experiment results in the next subsection show that, for $\alpha = 0.5$, the average number of memory accesses is actually very close to one.

Figure 3-15 compares false positive ratios of Bloom-1, Bloom-2, and Bloom-α with $k = 3$ by simulations. The false positive ratio of Bloom-α is between those of Bloom-1 and Bloom-2. When the value of α is increased from 25% to 50%, the false positive ratio of Bloom-α is decreased, suggesting a performance-overhead tradeoff since the average number of memory accesses will increase. When $\alpha = 50\%$, the false positive ratio of Bloom-α is close to that of Bloom-2.

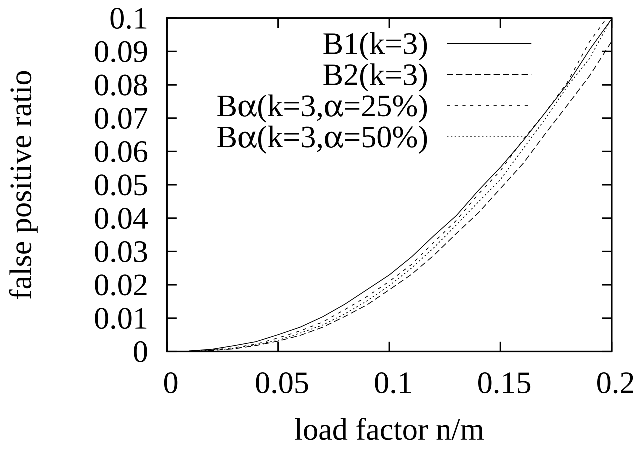

Figure 3-15. False positive ratios of Bloom-1, Bloom-2 and Bloom-α Filter with $k = 3$.

3.4.3 Experiment

We further evaluate the Bloom-α filter through experiments using network traces. The experimental settings are identical to those we used in Section 3.3.6.1. We compare eight filters. They are (a) two Bloom-1 filters: $B1(k = 3)$ and $B1(\text{optimal } k)$; (b) two Bloom-2 filters: $B2(k = 3)$ and $B2(\text{optimal } k)$, and (c) four Bloom-α filters: $B\alpha(k = 3, \alpha = 25\%)$, $B\alpha(k = 3, \alpha = 50\%)$, $B\alpha(\text{optimal } k, \alpha = 25\%)$, and $B\alpha(\text{optimal } k, \alpha = 50\%)$. It is difficult to determine the theoretical optimal k for Bloom-α, so we use the value of k that produces the lowest false positive ratio in simulations.

First, we compare the performance of $B1(k = 3)$, $B2(k = 3)$, $B\alpha(k = 3, \alpha = 25\%)$, and $B\alpha(k = 3, \alpha = 50\%)$. Figure 3-16 shows memory access overhead. We see that both $B\alpha(k = 3, \alpha = 25\%)$ and $B\alpha(k = 3, \alpha = 50\%)$ need much fewer memory accesses than the upper bound $(1 + \alpha)$ in Table 3-8. In fact, the numbers are close to one. There are two reasons for this: First, the fraction of split words is smaller than α. In fact, in our experiments, around 17.6% of the words are split when $\alpha = 25\%$, and 39.2% are split when $\alpha = 50\%$. As a result, the probability that an arbitrary element is mapped to a split word is smaller than α. Second, even if an element is mapped to a split word, it is not necessary that additional memory access is made, as we have discussed in Section 3.4.2. Figure 3-17 compares false positive ratios of the filters. We see that false positive

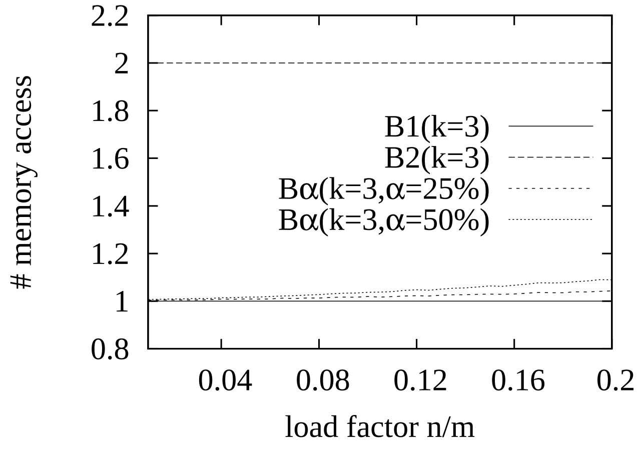

Figure 3-16. Average number of memory access for Bloom-1, Bloom-2 and Bloom-α Filter with $k = 3$ in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

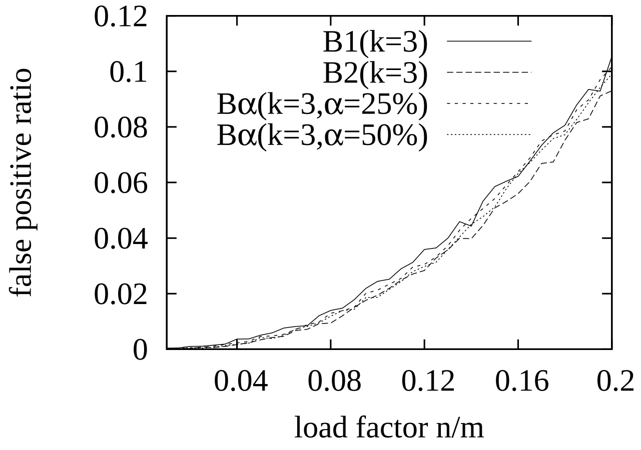

Figure 3-17. False positive ratios of Bloom-1, Bloom-2 and Bloom-α Filter with $k = 3$ in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

ratios of $B\alpha$ are smaller than Bloom-1 but slightly worse than Bloom-2, particularly when $\alpha = 50\%$.

Next, Figures 3-18 and 3-19 present performance comparison of $B1$(optimal k), $B2$(optimal k), $B\alpha$(optimal $k, \alpha = 25\%$), and $B\alpha$(optimal $k, \alpha = 50\%$). When the optimal k is used, the gap between $B1$(optimal k) and $B2$(optimal k) in terms of false positive ratio becomes larger. $B\alpha$(optimal $k, \alpha = 25\%$) and $B\alpha$(optimal $k, \alpha = 50\%$) fill the gap with smaller false positive ratios than $B1$(optimal k) and fewer memory accesses than

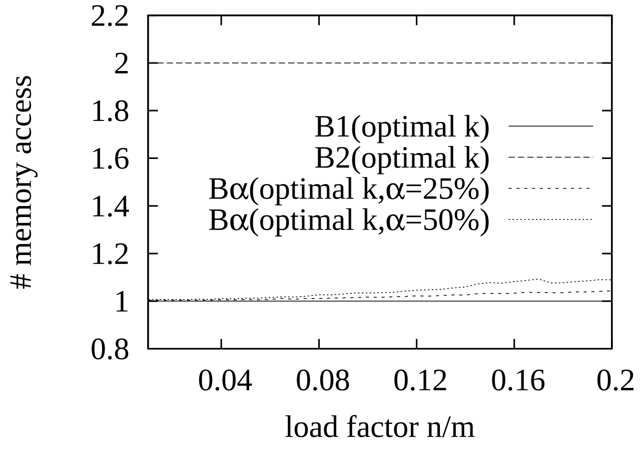

Figure 3-18. Average number of memory access for Bloom-1, Bloom-2 and Bloom-α Filter with optimal k in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

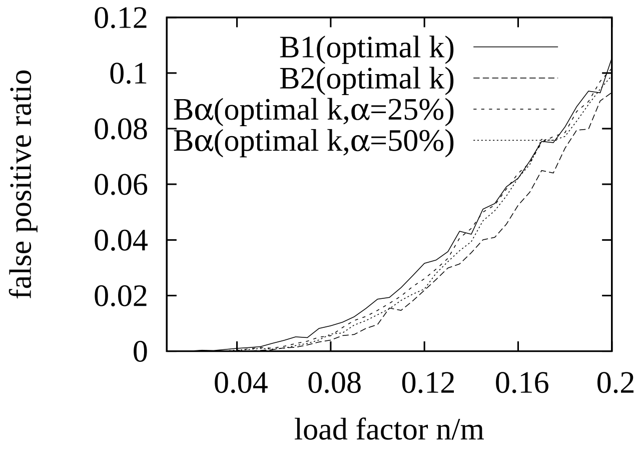

Figure 3-19. False positive ratios of Bloom-1, Bloom-2 and Bloom-α Filter with optimal k in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

$B2$(optimal k). Figure 3-20 presents the optimal number of membership bits for Bloom-α filters. Comparing to Figure 3-8, the optimal k for the Bloom-α is smaller than that of the Bloom-1 and Bloom-2 filters.

3.5 Summary

In this chapter, we introduce one memory access Bloom filters and their generalization. This family of data structures enriches the design space of the Bloom filters and their application scope by reducing the query overhead to allow high throughput. Using a

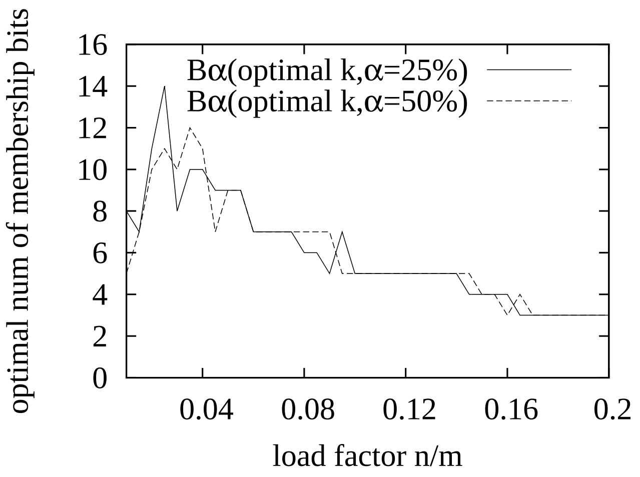

Figure 3-20. Optimal number of membership bits for the Bloom-α Filter in real trace experiment. Parameters: $n = 25,000$ and $w = 64$.

number of random bits in a word instead of from the entire bit array, we analyze the impact of this design change in terms of overhead and performance. This change also opens the door for constructing other variants for performance tradeoff. In this enlarged design space, we can configure filters that not only make fewer memory accesses but also have comparable or superior false positive ratios in scenarios where the standard Bloom filter with the optimal value of k incurs too much overhead to be practical.

CHAPTER 4
SPACE-TIME EFFICIENT MULTI-SET MEMBERSHIP LOOKUP

4.1 Background

Many important network functions require online membership lookup against a large set of addresses, flow labels, signatures, etc. For example, a router may be configured with counters [13] to collect per-client information for a given set of client addresses. For each arrival packet, the router needs to perform membership lookup to see if the source address of the packet belongs to the client set. A router may also be instructed to identify the set of current flows. This requires the router to collect flow labels [96, 97], such as client addresses or address/port tuples that identify TCP flows. Since each flow label should be collected only once. When a new packet arrives, the router must check whether the flow label extracted from the packet belongs to the set that has already been collected before. For routing-table lookup, we need to determine whether a given destination address prefix is in the routing table [143]. These online membership lookup problems have been extensively studied. Many influential solutions [40, 76, 97, 103, 142, 143, 147] are designed using Bloom filters [7, 12], which are compact data structures suitable for hardware implementation in on-die SRAM memory.

This chapter studies a more difficult, yet less investigated problem, called "multi-set membership lookup" (MSM), which involves multiple (sometimes in hundreds) sets, and classifies an incoming stream of elements, e.g., addresses, ports, a combination of them and other fields in packet headers, or even packet content [1] . The classification determines not only whether an element is a member of the sets but also which set it belongs to. It has many important applications. For example, with MSM, the routers of an ISP can classify packets for differentiated services based on their sources which are

[1] In the literature, there is another type of "multi-set", which describes that an element may appear multiple times in a set.

placed into different sets according to different types of service contracts. A firewall may be supplied with an action list for addresses that are collected by an intrusion detection system. Depending on the suspicious levels of source addresses, some packets may be logged, some may be content inspected, while others may be dropped. The firewall must check each arrival packet to see if the source address is a member of the different sets, and if it is, what further action it should take, depending on which set the packet belongs to. In another example, as the VMs are placed onto the servers of a data center, each server has a set of VMs. The gateway can be tasked to determine which incoming packets should be forwarded to which server (where the corresponding VMs are hosted).

Traditional exact-match data structures such as binary search tree [58], trie [48], or hash table [41] have to store both keys and values (set IDs), and some need additional space for pointers. Therefore, they require much more space than Bloom filter variations. Tries can organize the keys in a compact way, but still extra pointers are required for sparse key space [44]. This chapter studies probabilistic-match data structures to reduce the memory requirement, which is useful when multi-set membership lookup needs to be implemented on small but fast on-die memory to keep up with high line speed. Our solution combines the Bloom-g filter with a multi-hashing table employing a novel load-to-left, candidate-to-right policy for element placement. These techniques together allow the proposed multi-set membership lookup function to work in very compact memory, take only a few memory accesses and hash operations for each lookup, and have much lower error probabilities when comparing with alternative data structures.

4.2 Problem Definition

4.2.1 Multi-Set Membership Lookup (MSM)

Consider a number s of sets whose IDs are $1, 2, \ldots, s$, respectively. Each set has a certain number of elements. Given an arbitrary element e, the multi-set membership lookup function is to find the set ID that e belongs to. We consider the sets to be disjoint for applications that classify an incoming stream of elements into different sets. So the returned value for the MSM function is either a valid set ID S_e ($1 \leq S_e \leq s$) if $e \in S_e$, or 0 if e does not belong to any set, where we use S_e for both ID and the set for convenience.

4.2.2 Performance Metrics

The performance criteria considered in our design of the MSM function are given below:

- Space: For space efficiency, we should reduce the number of bits it takes to encode each member and its set ID. This is extremely important if the data structures are placed in on-die SRAM.
- Memory Access: We should reduce the number of memory accesses to SRAM for each membership lookup. This is particularly important if operations are performed very frequently, e.g. on a per-packet basis in a router.
- Hash Computation: We also want to reduce the number of hash bits that are needed for each membership lookup. This helps reduce the computational overhead.

To further elaborate on correctness, we define a few concepts:

- False Positive: For an element that does not belong to any set, false positive happens if the MSM function mistakenly believes the element belongs to a set.
- Conflict Classification: For an element that belongs to a set, conflict classification occurs if the MSM function cannot definitively determine the right set ID but knows it must be among several candidate sets.
- Mis-Classification: For a member of an existing set, mis-classification occurs if any of the following two conditions happens: (1) the MSM function claims that the element belongs to a different set, or (2) the MSM function claims that the element does not belong to any set.

False positive and conflict classification are direct consequence of bit sharing in Bloom-like data structures, where the set of bits to which a non-member is mapped happen to be all ones. While misclassification can be avoided [57], false positive and conflict classification cannot be eliminated because, for a virtually unlimited number of non-members, the probability for a non-member to share the same set of bits with members cannot be reduced to zero in hash-based designs adopted by Bloom filters and their alike.

In this chapter, we focus on data structures that may have false positives or conflict classifications, but do not have mis-classifications. In other words, no member in an existing set is treated as non-member or member of a different set.

4.3 Design of MSM Function

4.3.1 Motivation

If coded Bloom filters [21, 57, 98] are used, each entry in the membership lookup table (i.e., each member in S_e, $1 \leq S_e \leq s$) has to be encoded multiple times. To reduce the number of bits needed per entry, we should encode each entry just once, as the straightforward solution [21] does. One possibility is to encode each entry, including both the member identifier and the set ID, i.e. (e, S_e) , as a member in a Bloom filter [7, 12]. The problem is that when performing lookup for an element, we only have the element identifier and will have to try all set IDs to see if any of them, when combined with the element identifier, is encoded in the filter. This is equivalent to what the straightforward solution does. The number of different set IDs is in thousands, causing huge lookup overhead. The Bloomier filter [22] does not perform well either as our later evaluation will show.

Our idea is to create indirection in the lookup process by separating membership encoding and set ID storage in two data structures, called “index encoder” and “set-id table” (abbreviated as SID-table), respectively. In the index encoder, we encode the membership of a member as well as a small index that points out where to find the right set ID in the SID-table. This index may take a few different values (e.g., from 1 to 10). The lookup process consists of two steps: Given a member identifier, the first step tests whether the member is a member and checks the few index values (instead of set IDs in thousands) to see which one is encoded in the index encoder. Using the right index, the second step finds out where to fetch the set ID from the SID-table.

Because the first step checks all possible index values, if their encoding locations scatter all over the index encoder, we will have to make many memory accesses. To reduce the number of memory accesses, we cluster the encoding locations of all indices in one or a few contiguous blocks in the index encoder, where each block can be retrieved in one memory access. In this way, we only need to make one or a few memory accesses before checking all indices in parallel within a multi-core processor.

4.3.2 MSM Function

4.3.2.1 Set-ID Table

The SID-table stores the set IDs of the members. Each entry in the SID-table consists of two fields: a checksum and a set ID. The checksum is used for resolving lookup conflict. If the set ID field is zero, it means the entry is unused (Valid set IDs start from 1 and go up).

4.3.2.2 Index Encoder

The index encoder shares similarity with a Bloom filter. It is an array of bits that are organized in a two-level structure. At the first level, the array is divided into consecutive blocks, each of which may be 64 bits long and can be retrieved in one memory access. At the second level, each block consists of consecutive bits, which encode the key of member elements together with the location in SID-table where their set-IDs are stored.

4.3.2.3 Insertion

To insert a new member e of set S_e, we hash e to k entries in the SID-table, where k is a system parameter. These entries are called the “candidate entries” for member e. We will select one of them to store set ID S_e; that entry is called the “primary entry” for e.

More specifically, the SID-table is divided into q equal-sized segments,[2] where $q \leq k$. For convenience, we refer to the order from the first segment to the qth segment as "from left to right", following [8, 69]. We hash e to at least one candidate entry in each segment. When $k > q$, some segments will have more than one candidate entry. The candidate entries are indexed from 1 to k by the order of their segments and, for those in the same segment, by the order of hashing.

If any candidate entry is unused, we will be able to successfully insert the member. Let a be the index of the primary entry for e; it is also called the "primary index". We insert a checksum computed from e, and set the set ID field to be S_e. However, if all candidate entries for e are used, we fail in finding an entry to store S_e in the SID-table. In this case, we insert the pair, e and S_e, in a small ternary content-addressable memory (TCAM) [122].

In order to support efficient lookup, we encode the primary index a in the index encoder by two steps: In the first step, we hash the member identifier e to a number g of blocks in the index encoder, where g may be one or a small integer. We then fetch these blocks to the processor. They can be logically thought of as a small Bloom-like filter, denoted as C_e, now residing in the processor for encoding a. In the second step, we hash a (together with e) to k' bits in C_e and set them to '1'. These operations are performed within the processor. After that, the blocks are written back to SRAM. The way that the index encoder works is very similar to the Bloom-g filter we mentioned in Chapter 3. In this chapter, when we say "we hash the primary index a" (or hash another index), we always mean to hash it together with the member identifier e.

[2] Technically speaking, we may allow variable-sized segments [11, 66]. However, equal-sized segments are easier to handle in practice for multi-banked memory and dynamic memory allocation. See Section 4.4.

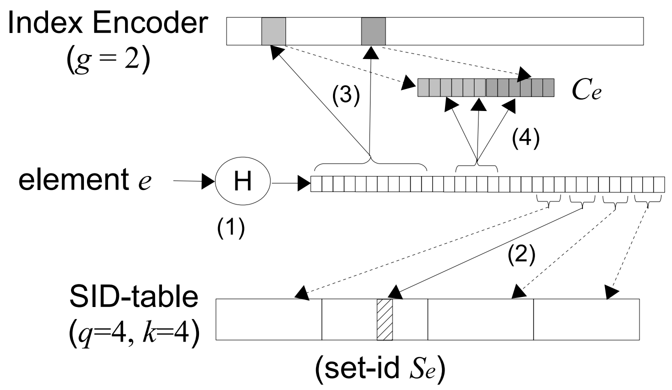

Figure 4-1. An example of inserting a member to the MSM function.

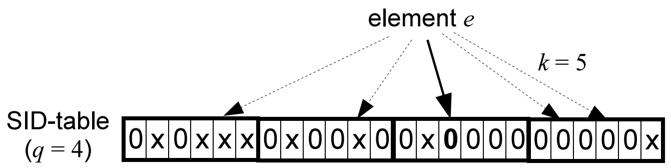

Figure 4-2. An illustration of using “load-to-left, candidate-to-right” policy to insert a member to a SID-table. An entry marked with ‘x’(‘0’) means is used (unused).

Figure 4-1 illustrates how insertion to the MSM function is done through a simple example. (1) We perform hash operations on the member’s id e and obtain a sequence of hash bits. (2) Using the hash bits, we find k candidate entries from the SID-table, and store the set ID S_e of e with a checksum computed from e to one of the entries; the index of the entry is a. (3) Using obtained hash bits, g blocks are fetched from the index encoder, which form a virtual Bloom-like filter C_e. (4) We encode (e, a) to C_e.

4.3.2.4 Load-to-left, Candidate-to-right

For insertion, there are two remaining problems: How to choose the primary entry? How many candidate entries should each segment have? Our objective is to increase the probability of finding an unused entry for each new member. To achieve this objective, we adopt a “load-to-left and candidate-to-right” policy to address those two problems, where load-to-left has appeared in previous hashing schemes [8, 11, 69] and candidate-to-right is original.

The “load-to-left” policy deals with the choice of the primary entry: If e is hashed to more than one unused entry, we always choose the one from the segment most to the left. This creates a skew in the segment load, which is measured as the fraction of entries in a segment that are used. A segment to the left tends to have a higher load than a segment to the right, as shown in Figure 4-2. A skew in the load improves the probability for a new member to be successfully inserted. For example, suppose $k = q = 2$ and the segment loads are 0.9 and 0.1, respectively. The successful-insertion probability is $1 - 0.9 \times 0.1 = 0.91$. In comparison, if the segment loads are uniform, i.e., 0.5 and 0.5, the successful-insertion probability will be only $1 - 0.5 \times 0.5 = 0.75$.

Let α_i be the load of the ith segment. Its value can be recursively computed. Let n be the number of members and l be the number of entries in the SID-table. Each segment has l/q entries. Suppose $k = q$. The probability P_i for an arbitrary entry in the ith segment to be unused is

$$\begin{aligned} P_1 &= (1 - \frac{1}{l/q})^n \approx e^{-qn/l} \\ P_i &= (1 - \prod_{j=1}^{i-1}(1 - P_j) \times \frac{1}{l/q})^n \\ &\approx e^{-\prod_{j=1}^{i-1}(1-P_j) \times qn/l}, \quad \forall i \in [2, q]. \end{aligned} \tag{4–1}$$

Clearly, the value of P_i increases with respect to i. The expected value of α_i is simply $1 - P_i$, which decreases with respect to i. In fact, the segment loads decrease quickly from α_1 to α_q. Similar analysis can be done for the case of $k > q$, i.e., a segment may have more than one candidate entry.

The “candidate-to-right” policy determines the number of candidate entries in each segment. Taking advantage of the load skew, it further improves the probability for a new member to be successfully inserted into the SID-table. If $k = q$, the successful-insertion probability is $1 - \prod_{i=1}^{q} \alpha_i$. If $k > q$, the successful-insertion probability is $1 - \prod_{i=1}^{q}(\alpha_i)^{d_i}$, where d_i is the number of candidate entries in the ith segment. Since α_q has the

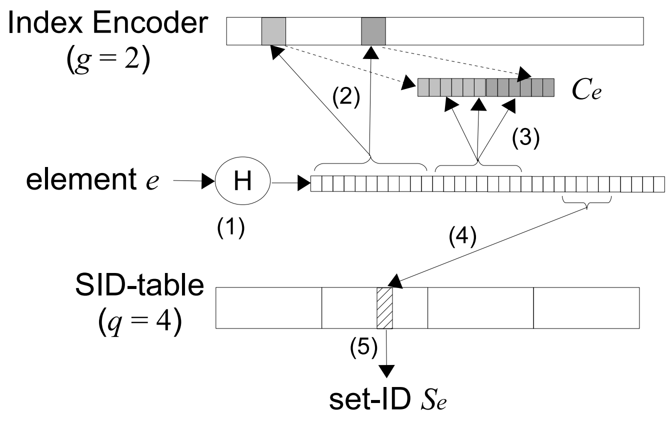

Figure 4-3. An example of looking up a member in the MSM function.

smallest value, we naturally want to push the candidate entries to the rightmost. As shown in Figure 4-2, $d_i = 1, \forall i \in [1, q-1]$, and $d_q = k - q + 1$. The successful-insertion probability becomes $1 - \prod_{i=1}^{q-1} \alpha_i \times (\alpha_q)^{k-q+1}$.

The effect of the above candidate-to-right policy may be amplified if we reduce the value of α_q. This can be achieved by allowing more than one candidate entry in the second rightmost segment. If we use two candidate entries in the $(q-1)$th segment, a new member will have a better chance to find an unused entry in the $(q-1)$th segment, without having to resort to the qth segment, thereby reducing its load α_q. In this case, the successful-insertion probability is $1 - \prod_{i=1}^{q-2} \alpha_i \times (\alpha_{q-1})^2 \times (\alpha_q)^{k-q}$. It performs better if the reduction in α_q more than compensates for the reduction in its exponent.

In general, if the values of n, l, q and k are fixed, we can mathematically compute the optimal values of d_i, $i \in [1, q]$, that maximize the successful-insertion probability.

4.3.2.5 Lookup

Consider an arbitrary element e, we use MSM to determine its set ID. The steps are shown in Figure 4-3. First, we generate a sequence of hash bits using element e. Using the hash bits, we locate and fetch g blocks in the index encoder, i.e., C_e, to the processor, where the remaining operations are performed. The processor has k units that tests in parallel whether any candidate index i is encoded in C_e, for $1 \leq i \leq k$. Each unit hashes i to k' bits in C_e and checks whether all bits are ones. If so, i is encoded in C_e.

If e is a live member and it has been previously inserted in the MSM function, its primary index a must have been encoded in C_e. Ideally, all other indices should be tested as not encoded in C_e. However, false positive may happen such that the k' bits of another index are all ones because they are set during the insertion of other members (which are also hashed to these bits by chance). In this case, we need to figure out which one is the primary index that we can use to find the right entry in the SID-table for member e. The checksum field in the SID-table will serve for this purpose. We emphasize that the probability of false positive should be made very small such that in most cases only one index (namely, a) will be found in C_e. If no index is found in C_e, member e is not a member and the packet will be dropped.

For each index i that is encoded in C_e, we hash e for its ith candidate entry in the SID-table. We then fetch the entry and compare its checksum field with what's computed from e. If the checksum does not match, i must not be the primary index. Otherwise, we output the set-ID field of that entry. Figure 4-3 shows an example of looking up a member in the MSM function.

Clearly, the primary index a will pass the above checksum test. However, the checksum field cannot totally resolve false positive because the probability for another index i to pass the checksum test by chance is small but not zero. When this indeed happens, we have conflict classification.

Because some members may be stored in TCAM, the lookup happens simultaneously in both TCAM and index encoder/SID-table. If there is a match in TCAM, the lookup in the index encoder/SID-table will be aborted and its result will be discarded.

4.3.2.6 Hash Functions

The MSM function requires many hash operations. For example, it needs to hash a member identifier e to b blocks in the index encoder, and then hash each index in

$[1, k]$ to k' bits in C_e during lookup. For insertion, it needs to hash e to k entries in the SID-table. That will add up to $g + q \times k' + k$ hash operations. If $g = 2$, $q = k = 4$ and $k' = 5$, the number of hash operations will be 26. But the actual implementation does not require so many if we take a hash-bit's point of view, instead of a hash-operation's point of view.

Let l be the number of entries in the SID-table, l' be the number of blocks in the index encoder, and each block be 64 bits long. For the index encoder, the lookup operation needs the most hash bits: $g \log_2 l'$ hash bits to locate C_e, and $k' \log_2 64g$ hash bits to locate bits for each index in $[1, k]$. If $l' = 1$M, $g = 2$, $q = k = 4$ and $k' = 5$, it needs just 180 hash bits, which can be produced by three hashing operations if each generates 64-bit output. Suppose 192 hash bits are produced by hardware. We take the first $g \log_2 l' = 40$ bits to locate C_e. Then we take the next $k' \log_2 64g = 35$ bits to locate the k' bits in C_e for index 0, take the yet next 35 bits to locate the bits for index 1, and so on.[3]

For the SID-table, the insertion operation needs the most hash bits. It needs $k \log_2(l/q)$ hash bits to locate k entries. If $l = 0.25$M,[4] we need 80 hash bits, which can be produced by two hashing operations if each generates 64-bit output.

4.3.3 Conflict Classification

Conflict classification happens under the following condition: (1) false positive occurs in the index encoder such that more than one index is found in C_e, and (2) the

[3] It may be counter-intuitive to implement hashing of an index (together with the member identifier) in this non-conventional way. A normal hash function would take the index and the member identifier as two inputs. In the above implementation, it uses the first input to produce a sequence of hash bits and then uses the second input to choose some bits from the sequence.

[4] Our analysis and experiments will show that the number of entries in the SID-table can be much fewer than the number of bits in the index encoder.

checksum field in the SID-table fails in resolving the false positive. Below we derive the probability of conflict classification.

For each member inserted in the MSM function, the k' bits for its primary index are set to ones. Consider an arbitrary bit z in C_e. It remains zero if it was not chosen by the primary index of any member. Let w be the number of bits in each block. The probability that z is not chosen by the primary index of member e is $(1-\frac{1}{wg})^{k'}$, where wg is the number of bits in C_e. The probability that z is not chosen by any other member e' is $1-[1-(1-\frac{1}{l'})^g]\times[1-(1-\frac{1}{wg})^{k'}]$, where the first term in the product is the probability for $C_{e'}$ to contain z's block and the second term in the product is the probability for z to be chosen by the primary index of e'. Hence, the probability for z to be zero is

$$P_0 = (1-\frac{1}{wg})^{k'} \times \left(1-[1-(1-\frac{1}{l'})^g]\cdot[1-(1-\frac{1}{wg})^{k'}]\right)^{n-1}, \tag{4–2}$$

where n is the number of members.

Consider an arbitrary non-primary index of member e. It chooses k' bits in C_e. False positive occurs if all of them are ones. Hence, the false-positive probability is

$$P_{fp} = (1-P_0)^{k'}. \tag{4–3}$$

This is not strictly correct as it assumes independence for the probabilities of each bit being set. However, it is a very close approximation.

The probability that the checksum in the SID-table cannot resolve this false positive is $\frac{1}{2^c}$, where c is the number of bits in the checksum. Hence, the probability for one non-primary index to cause conflict classification is $\frac{1}{2^c}\times P_{fp}$. Because there are $k-1$ non-primary indices, the conflict-classification probability is

$$P_{conflict} = 1-(1-\frac{P_{fp}}{2^c})^{k-1}. \tag{4–4}$$

4.3.4 False Positive

Let e be the identifier of an element that does not belong to any set. The MSM function first hashes e to find C_e, and then checks if any candidate index i is encoded in C_e. A candidate index is not in C_e if one of its k' bits is zero. If no candidate indices are found in C_e, the MSM function knows that e is not a member. Due to false positive, however, one or more candidate indices may be found in C_e. In this case, the MSM function moves to the SID-table. Each candidate index in C_e points to a candidate entry in the SID-table. If the checksum of that entry does not match what's computed from e, we know that it is a false-positive caused by the index encoder; if this happens to all indices found in C_e, the MSM function will recognize e as a non-member. However, if the checksum of one entry matches e, the set ID in that entry will be outputted, resulting in false positive.

We derive the probability of false positive. Following a similar analysis that leads to (4–2), we give the probability for an arbitrary bit in C_e to be zero as follows:

$$P'_0 = \left(1 - [1 - (1 - \frac{1}{l'})^g] \cdot [1 - (1 - \frac{1}{wg})^{k'}]\right)^n. \tag{4–5}$$

It is slightly different from (4–2), without the term $(1 - \frac{1}{wg})^{k'}$, because no primary index of e is encoded.

Consider an arbitrary index of member e. It has k' bits in C_e. False positive occurs if all of them are ones. Hence, the false-positive probability is

$$P'_{fp} = (1 - P'_0)^{k'}. \tag{4–6}$$

The probability that the checksum in the SID-table cannot resolve this false positive is $\frac{1}{2^c}$. Hence, the probability for one index to cause false positive is $\frac{1}{2^c} \times P'_{fp}$. Because there are k indices, the false positive probability is

$$P_{false} = 1 - (1 - \frac{P'_{fp}}{2^c})^k. \tag{4–7}$$

4.3.5 Mis-Classification

Next, we consider a member element e that belongs to set S_e. By definition, mis-classification occurs if any of the following two conditions holds: (1) the MSM function claims that e belongs to a different set S'_e, or (2) the MSM function claims that e does not belong to any set.

Since the member has been inserted in the MSM function, the lookup will certainly find the primary index a in the index encoder, and that leads to the primary entry in SID-table for the set ID S_e. So condition (2) will not happen for MSM function. Condition (1) only happens when the MSM function outputs S'_e as the sole answer, which is impossible given that S_e is guaranteed to be included in output. In other words, the membership lookup result for element e can either be S_e or a conflict classification.

4.4 Discussions

4.4.1 Deletion

As Bloom filters does not support deletion inherently due to randomized sharing of bits. In order to support deletion in the MSM function, we may use one of the two techniques: counter based deletion and time-based deletion.

4.4.1.1 Counting-based Deletion

We replace the bit array C_e with a counter array as counting Bloom filter does [54, 82, 134]. When we insert the primary index a of a member e, we hash e to find C_e. We then hash a to k' counters (instead of bits) and increase all those counters by one. When we delete a member e' of set $S_{e'}$, after finding $C_{e'}$, we check all indices $i \in [1, k]$ to see which one is encoded in $C_{e'}$. If there is only one index in $C_{e'}$, it must be the primary index a. We hash e' for its ath candidate entry in the SID-table and set the set ID field to zero, which releases the entry for future use.

The advantage with counting based deletion is that the logic is simple and it's easy to implement. However, what if we find more than one index in $C_{e'}$ (due to false positive)? They give us more than one candidate entry in the SID-table. If the checksums of two or more candidate entries match e, we will have no idea which entry to remove. Although the probability for this to happen is small, it can happen and we will not be able to release a used entry. Over time, such wasted entries may accumulate.

4.4.1.2 Time-Based Deletion

If each member only remains in a set for a period of time, we can perform deletion periodically by removing unvisited elements in previous period. More specifically, we replace each bit in the index encoder with a two-bit code, and add one flag bit to each entry of SID-table. The purpose of these is to track whether an element is visited (being looked up).

Initially, all flag bits of SID-table entries are ‘0’, if an entry in SID-table is visited and the checksum matches, we set the flag bit of that entry to ‘1’. After each period, the MSM function recycles all entries in the SID-table whose flags are ‘0’. To change their status to unused, we simply assign zero to the set ID field. We also reset the flags of all entries to ‘0’, preparing for deletion after the next period.

The MSM function also clears the codes that are not accessed in the previous period. All codes are initially “00”. When a member is inserted or visited, the codes it maps to are set to “10”. Code “11” is used to resolve conflict classification as we will show later. Membership lookup checks the positiveness of codes. During periodical deletion,

- Codes that are “01” (not being accessed during the previous period) are set to “00”.
- Codes that are “10” or “11” (having been accessed during the previous period) are set to “01”, initializing them for the next period.

Figure 4-4 shows the detailed transition of code states.

Deletion needs to access the entire index encoder and SID-table. If the SRAM space allocated to the MSM function is 10MB and each memory access retrieves a block of 64 bits into a hardware unit for deletion, there are 1.25M blocks and it takes 2.5M memory accesses (for reading the blocks and writing them back) to process all

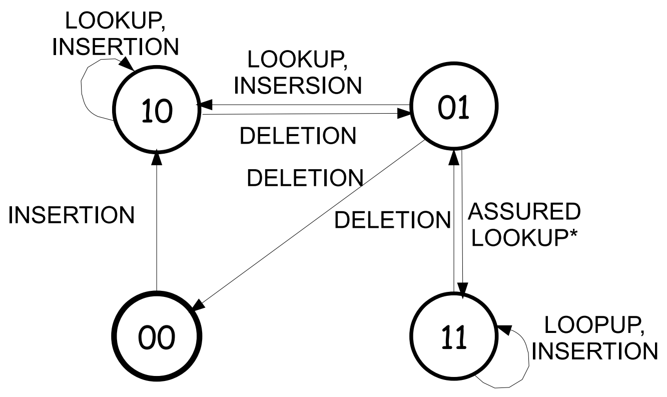

Figure 4-4. State transition diagram for codes in index encoder with time-based deletion. “00” is the starting state.

blocks. This overhead is manageable if we amortize it over the numerous packets that the MSM function receives over each deletion period. For example, suppose MSM is used to classify inbound packets of a router, and deletion is performed after every 10 seconds. If the inbound line speed is 100 Gbps and the average packet size is 1000 bytes, the number of packets received over a period is around 125M.[5] If we interleave memory accesses for deletion with packet processing, we may process one block for deletion after forwarding an inbound packet. In this way, for the first 1.25M packets out of 125M, two memory accesses for deletion will be made after each packet. That translates into a sub-period of about 0.1 second (out of 10 seconds). After that, the deletion is completed. Alternatively, we may process one block for deletion after forwarding multiple packets, which reduces the impact of deletion even at the micro time level.

Code Value “11”: Suppose conflict classification happens to a member with identifier e. To prevent subsequent lookups of member e from encountering conflict classification, the MSM function may issue an exact lookup to confirm the real set ID of e. For example, some application may store the exact membership in larger but slower

[5] For simplicity in discussion, we ignore the physical layer overhead during transmission which cuts down the packet rate.

DRAM. Then, it performs a lookup on e and finds the primary entry in the SID-table whose set ID field stores x. Once it finds such an entry, it knows that the corresponding index must be the primary index. It then goes back to the index encoder, finds the codes for the primary index, and sets them to “11”.

For a subsequent lookups of member e, when the MSM function performs lookup and finds multiple indices in C_e, it will use the index whose codes are all “11”.

Although the above approach helps resolving conflict classification, it cannot totally eliminate conflict classification because in theory false positive may still occur such that the codes of a non-primary index are all “11”, no matter how small the likelihood is. Nevertheless, with the help of code value “11”, we can reduce the chance for conflict classification to happen. False positive may even occur such that the codes of a non-primary index are all “11”, while the codes of the primary index are not yet set to all “11”. However, the probability for this to happen is extremely small.

4.4.2 Multiple Memory Banks

So far we have not considered multi-banked on-chip memory, which implements separate SRAM arrays such that each can be accessed independently. Normally, each bank interleaves its address blocks with other banks. It may also be designed to occupy a contiguous section of addresses. Multi-banked memory allows us to make multiple memory accesses in parallel. It helps increase the memory bandwidth if on-chip memory becomes a performance bottleneck. For example, the combinatorial Bloom filter (COMB) [57] may need to make over a hundred memory accesses per packet, depending on its parameter setting. If numerous memory banks are used to support parallel accesses, the actual per-packet delay due to memory access will be greatly reduced.

Multiple banks can equally benefit the MSM function. First of all, if the g blocks in C_e are taken from g memory banks respectively, they can be fetched in parallel. If the SID-table resides on a different memory bank, the operations on the index encoder and the SID-table can be pipelined: For example, when the lookup of a packet moves to the SID-table, the lookup of the next packet can begin to fetch blocks from the index encoder.

For each membership query, the MSM function makes g memory accesses to fetch C_e from the index encoder. If there is no conflict classification, it makes one memory access to fetch an entry from the SID-table. Hence, when $g = 2$, three memory accesses are needed in total. If the number of memory banks available is more than that, we can still make full use of the parallelism by looking up multiple elements simultaneously. We divide memory banks into groups and assign members to different groups. Each group has its own index encoder and SID-table, as well as a dedicated set of hardware units for insertion, deletion and lookup. When inserting a member e, we hash it to a memory-bank group and then stores its set ID in the SID-table and index encoder of that group in exactly the same way as we have described previously.

4.4.3 System Overload

When the number of members is too many to be stored in the SID-table, insertion failure will happen more frequently, which may overflow the TCAM, resulting in system overload. The MSM function can be designed to dynamically increase the SID-table to accommodate more members. To do so, we add a number r of more segments by preempting less important functions that are also implemented on the EN-interface.[6] Recall that the SID-table consists of q segments from which k candidate entries are accesses for each new member during insertion. Besides adding segments, we also need to increase the number of candidate entries from k to $k + r$, assuming that each

[6] For example, a traffic measurement function may have lower priority than cloud membership lookup. It can be stopped to give up its memory until the overload condition eases.

new segment has one candidate entry and that there are extra hardware units to test whether the additional indices are encoded in C_e during lookup.

When the overload condition eases and existing members can be held in k segments, we will release the added segments from the SID-table when their entries become unused.

4.5 Evaluation

We compare our data structure (MSM) with other two Bloom filter based data structures: the Bloomier Filter (Bloomier) [22] and the combinatorial Bloom filter (COMB) [57] using synthetic data. The details of COMB and Bloomier can be found in Section 2.3.4.

4.5.1 Simulation Setup

In our simulation, we allocate the amount of memory ($m = 16\ Mbits$) for each data structure and insert n random generated members with random set IDs. The number of sets s is 4000, and the set IDs are 1, 2, ..., 4000. We vary n so that the average memory per member changes from 20 bits to 50 bits ($\frac{m}{50} \leq n \leq \frac{m}{20}$). Given that we need $\lceil \log_2 g \rceil = 12$ to represent a set ID, there are actually 8 bits to 42 bits per member left to encode the membership.

We use the optimal configurations of the data structures as follows:

Bloomier: We assign $k = \ln 2 \frac{l}{n}$ elements to encode each member, where $l = \frac{m}{\log_2 g + 1}$ is the number of entries. This will minimize both false positives and insertion failures [12].

COMB: We choose COMB(15, 6) which can encode up to $\binom{15}{6} = 5005$ sets. We use $k = \ln 2 \frac{m}{6n}$ hash function for each code bit to obtain fewest false positives [57].

MSM: We choose $g = 2$ to bound the number of memory access per lookup; We divide the SID-table to $q = 8$ segments and set the number of candidate entries k to 10 to obtain reasonable hash overhead and insertion failure rate. We choose the optimal k' that minimizes the false positive ratio of the index encoder (P'_{fp}) based on

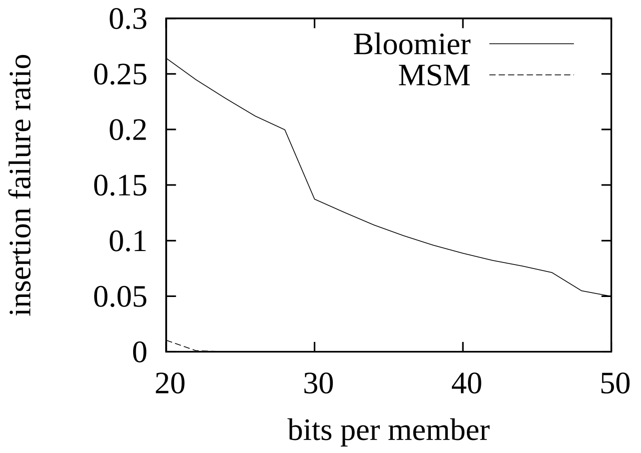

Figure 4-5. Insertion failure ratio of the Bloomier filter and the MSM filter.

Equation (4–3). Then, the only left open variables are c and l'. As c and l' are bounded integers, with the help of linear programming, we can compute the optimal values which minimizes the false positive ratio in Equation (4–7).

4.5.2 Simulation Results

4.5.2.1 Insertion Failure

First we keep inserting members to the data structures until the number of members encoded reaches n. For the Bloomier filter, insertion failure happens when there is no available element for a new member; For MSM, insertion failure occurs when all k candidate entries for the new member are used. Both Bloomier and MSM handle this by inserting the new member into TCAM. Figure 4-5 shows the ratio of members that encounter insertion failure. When there are 20 bits per member, the Bloomier filter fails to insert 26% of the elements, while MSM only fails 1%. When there are 50 bits per member, the Bloomier filter has 5% insertion failure, while MSM does not encounter any insertion failure since bits per member reaches 25. The consequence of insertion failure is consumption of expensive TCAM or using larger SRAM memory instead. It is a tough decision to make as both TCAM and SRAM are expensive for a network device.

4.5.2.2 Membership Lookup Overhead

Next we examine the hash overhead and memory access overhead for each membership lookup. Figure 4-6 shows the average number of memory accesses for each membership lookup. While the number of bits per member increases, the number of memory accesses for COMB and Bloomier also increases. This is because they use larger k to reduce false positive ratio. MSM has less memory accesses when given more bits to encode each member. This is because smaller false positive ratio in the index encoder results in fewer memory accesses to SID-table. The Bloomier filter has 1–3 memory access per query. MSM has 7.8–4.5 memory accesses for members, and 7.3–3.6 memory accesses for non-members. The minor difference is due to the nature of the index encoder: if no index is found from index encoder, there is no need to access the SID-table. COMB requires 30 to 90 memory accesses for each membership lookup, which is 10 times that of MSM, and 30 times that of the Bloomier filter.

Figure 4-7 presents the number of hash bits required by the data structures for each membership lookup. Among them, the Bloomier filter works the best with 21–63 hash bits per query. MSM requires 278 hash bits, and COMB requires 720–2160 hash bits.

4.5.2.3 Correctness

Finally, we examine the correctness of the data structure by initiating membership queries for both members and non-members. Two types of errors may happen: false positive which happens when a valid set-ID is outputted for a non-member; and conflict classification which means more than one valid set-ID is outputted for a member. Figure 4-8 and 4-9 presents the false positive ratios and conflict classification ratios of the three data structures respectively. The Bloomier filter outputs one and only one valid set-ID for member elements. Therefore, it does not have conflict classifications. COMB has the smallest false positive ratio, but it has large conflict classification ratio. However, the Bloomier filter has magnitude larger false positive ratio than the other two filters. In practice, one can determine which data structure to use if he/she has the knowledge

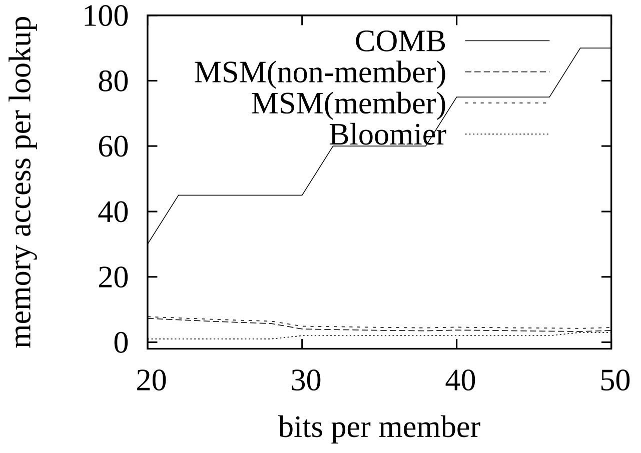

Figure 4-6. Average number of memory access per membership lookup by the Bloomier filter, COMB, and the MSM filter.

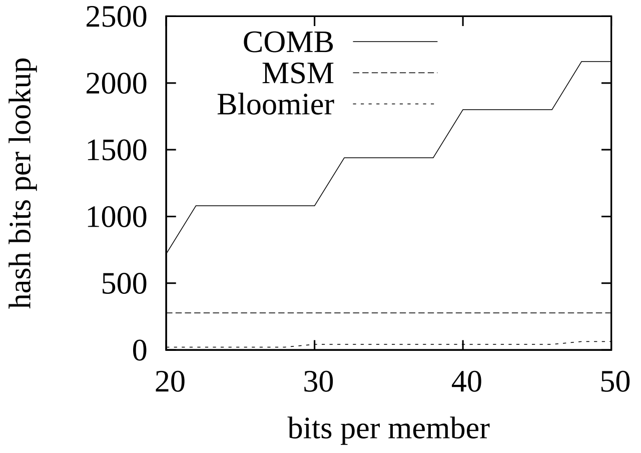

Figure 4-7. Number of hash bits required by the Bloomier filter, COMB, and the MSM filter for each membership lookup.

about the query set: If the query are mostly for members, MSM or the Bloomier filter is a good option. Otherwise if the query is mostly for non-members, COMB makes a good candidate as well.

Figure 4-10 shows the ratio of elements that is either false positive or conflict classification[7] , defined as the “error ratio”. Among them, COMB has larger error ratio

[7] The actual error ratio can vary depending on different query set.

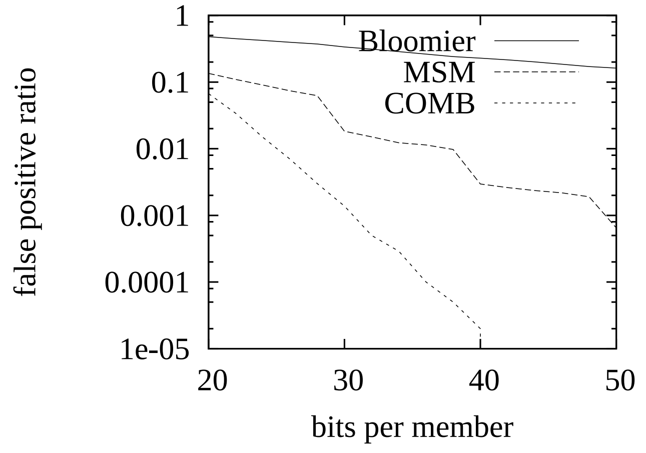

Figure 4-8. False positive ratio comparison among the Bloomier filter, COMB, and the MSM filter.

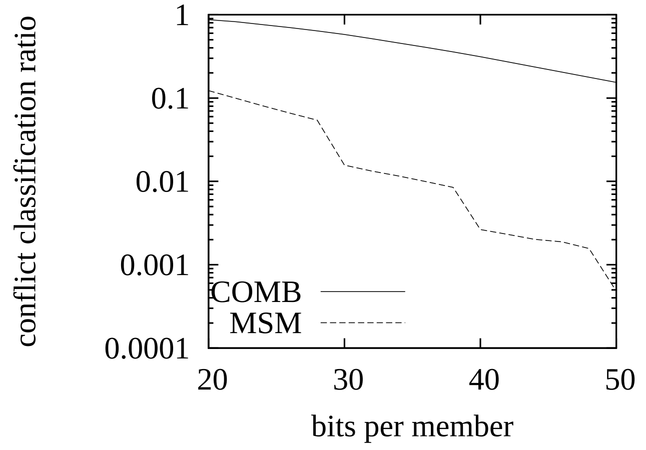

Figure 4-9. Conflict classification ratio comparison between COMB and MSM.

than the Bloomier filter. MSM has the lowest error ratio, and this is achieved by only 7.8 – 3.6 memory accesses and 278 hash bits.

4.6 Summary

In this chapter, we propose a novel data structure for multi-set membership lookup based on an index encoder and a SID-table that are highly space-efficient, yet need a small number of memory accesses for each query. We propose a load-to-left, candidate-to-right policy to improve insertion performance. Simulation show that our design significantly outperforms known alternatives.

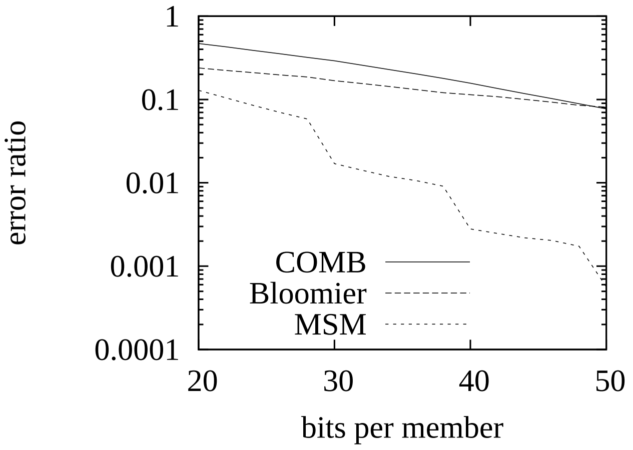

Figure 4-10. Error ratio of the Bloomier filter, COMB, and the MSM filter.

CHAPTER 5
ADAPTIVE BLOOM FILTERS FOR DISTRIBUTED JOIN

5.1 Background

Bloom filters are widely used in distributed systems, which can be used in anomaly detection, attack detection, network traffic measurement, account management, and so on. For example, distributed intrusion detection systems (IDSes) detect address/port scanning [145], in which an external host attempts to establish connections to a unusual number of internal hosts. Multiple IDSes guarding geographically dispersed sub-nets report to a central coordinator about their measurements on the number of distinct destinations that each source has contacted, and the central coordinator combines the measurements to detect scanners. As another security application, a honeynet detects attack by placing several fake servers called "honey-pots" in the network, which log all attempting accesses to it [144]. As honey-pots do not provide any real service, the one who contacts them is probably the source of a scanner attack or a worm [107]. The center can collect the log information from all honey-pots to locate attacker. As an example in traffic engineering, some ISPs measure their network and analyze the measurement results for improved QoS [26, 27, 86]. Some measure flows that travel in a particular path. In order to do this, each routers in the path will record all bypassing flows. Later, by joining the flows that passed two routers, we know the set of flows that were traveling between them. Lastly, in a peer-to-peer network such as a DHT, information is stored on millions of sites in the network. Each keyword is mapped to a node in the network, where an inverted list containing document IDs corresponding to the keyword is stored [23]. If multiple keywords are searched, the inverted lists from corresponding nodes are joined together to form the final search results.

In all above examples, there is a fundamental problem to be solved: finding the common elements of sets, which could be lists of source IP addresses captured by honeypots, sets of flows labels collected by routers, or inverted lists of document IDs.

We define this problem as the “distributed join problem”, which is to find the common elements of two sets that are distributed in different locations in the network.

The straight forward solution is to let one node to send the keys of elements to the other node. However, when the sets are large, it is not efficient to send the whole set. Suppose in a distributed monitor system, there is a million flows in a measurement period of 1 minutes, and each flow identifier consists of 100 bits [1] . Now in order to find the common flows collected by two monitors, one monitor needs to send $1M \times 100 = 100Mb$ flow IDs per minute, which translates to $1.6Mbps$ upstream/downstream traffic. As one monitor may share monitoring results with many others, the bandwidth requirement is too large.

Another solution is to use Bloom filter to filter unneeded flow IDs before transferring the real IDs [104]. One monitor transfers a Bloom filter that encodes its set of element to the other monitor. The other monitor then performs a membership lookup using its own set of flow identifiers. Now it only needs to send the flow identifiers that remain positive in the Bloom filter. Suppose the Bloom filter takes $20M$ (20 bit per element) and the false positive ratio is 0.1%. If there are 1,000 common flows, the real traffic for transferring flow identifiers is $(1M * 0.1\% + 1,000) * 100$ and the overall traffic for both Bloom filter and flow identifiers are $20.2Mb$. It brings down the traffic by around 80%.

The compressed Bloom filter [111] can be used in previous solution so further reduce communication cost. The compressed Bloom filter uses arithmetic coding to encode a sparse Bloom filter. It reduces the transmission size of the Bloom filter with the cost of compression/decompression at sender/receiver.

In this chapter, we will introduce another approach called adaptive Bloom filter, that is designed for efficiently distributed joining two sets. Our idea is to iteratively eliminate

[1] Suppose we use 5 tuples <source IP, source port, destination IP, destination port, protocol> to identify each flow.

elements that is not in the common set. We found that when the join is highly selective (the size of common elements is ignorable comparing to sizes of each set), the adaptive Bloom filter out performs both the Bloom filter and the compressed Bloom filter.

5.2 System Model

5.2.1 Problem Definition

Assume there are two sets of elements residing in different locations in the network. Without loss of generality, we call one of them the “local host”, and the other the “remote host”. We denote the two sets as S_l and S_r respectively. Our goal is for the local host to find out the set T of common elements between the two sets, i.e. $T = S_l \cap S_r$.

Exactly finding T can be expensive if the sets are very large. A large message containing element information has to be sent over the network. If the operation is performed frequently, it could have great impact to the network traffic. Alternatively, we may allow a few elements that do not belong to T to be included in the final result. In other words, we want to identify a set T^* such that $T \subseteq T^*$, and $\frac{|T^*-T|}{|T|}$ is small. This will enable us to compactly encode the sets using efficient data structures such as Bloom filters. If the goal is to exactly identify T, we can send over the elements in T^* (instead of the whole set) for exact checking. Tremendous saving is still possible.

5.2.2 Performance Metrics

The first performance metrics evaluates the accuracy of T^*. We define the false positive ratio as the probability for an element in $S_l - T$ to be included in T^*, i.e.,

$$f_p = \frac{\left|T^* \cap S_l - T\right|}{\left|S_l - T\right|}. \tag{5–1}$$

The second performance metrics describes the traffic overhead brought to the network. We define the message size as the amount of bits sent and received by the local host, denoted as M [2] .

The goal is to reduce the message size M and the false positive ratio f_p as much as possible. In fact, there is a tradeoff relationship between the message size and the false positive ratio, regardless what data structure is used to encode the message. Therefore, following the convention of [131], we bound the false positive ratio with a pre-defined parameter α, and compare the message sizes of different approaches. With the performance constraints, we define the optimization problem as follows: minimize message size M, under the constraint $f_p \leq \alpha$.

5.3 Bloom filter Approaches

5.3.1 Bloom filter (BF)

If Bloom filters are used, we can ask the remote host to send a Bloom filter encoding all its elements S_r, denoted as B_r. On receiving the Bloom filter, the local host checks its own elements one by one. If an element is tested to be a member in B_r, we insert it to the result set T^*. In the following, we prove that $T \subseteq T^*$.

From our description, $T^* = B_r \cap S_l$, where B_r stands for the set of elements encoded by the Bloom filter. As Bloom filters have false positives, B_r is slightly larger than S_r, and $S_r \subseteq B_r$. As a result, $T = S_r \cap S_l \subseteq B_r \cap S_l = T^*$. We have $T \subseteq T^*$.

For an element in $S_l - T$, the probability for it to be in T^* is the probability for an arbitrary element to be in B_r, which is exactly the false positive ratio of B_r. Recall that the false positive ratio of a Bloom filter is,

$$f_B = \big(1 - (1 - \frac{1}{m})^{nk}\big)^k \approx (1 - e^{-\frac{nk}{m}})^k, \tag{5–2}$$

[2] To simplify the analysis, we do not consider various headers added for physically transferring the message.

where m is the size of the Bloom filter B_r, $n = n_r$ is the number of elements in S_r. and k is the number of hash functions used. Please refer to Chapter 3 Section 3.2.1 for detailed derivation. When $k = \frac{m}{n} \ln 2$ is used, the false positive ratio is minimized with respect to m and n. The optimal false positive ratio is

$$f_B = \left(1 - e^{-m/n \ln 2 n/m}\right)^{m/n \ln 2} = e^{-\frac{m}{n}(\ln 2)^2}. \tag{5–3}$$

Adopting the performance constraint $f_B \leq \alpha$ to (5–3), we have,

$$f_B = e^{-\frac{m}{n}(\ln 2)^2} \leq \alpha. \tag{5–4}$$

Therefore,

$$m \geq -\frac{n \ln \alpha}{(\ln 2)^2}. \tag{5–5}$$

From (5–5), we know that in order to satisfy the false positive requirement, the minimum size for the Bloom filter is $M = -\frac{n_r \ln \alpha}{(\ln 2)^2}$.

5.3.2 Compressed Bloom filter (CBF)

The compressed Bloom filter is proposed by Mitzenmacher in [111] to reduce the size of Bloom filters when they are transmitted as a message. The idea is to use lossless compression algorithms to compress the Bloom filter before transmission, and decompress it at the receiver. The real size of the Bloom filter can be very large, yet it can be compressed for efficient transmission. If CBF is used for the problem of this chapter, the process is similar to that with the Bloom filter: a CBF CB_r is generated in remote host, compressed, and sent to the local host. The local host receives it and decompresses it back to CB_r. Then it checks its elements one by one to get the final result set T^*.

Let m be the size of the Bloom filter before compression, M be the size of transmitted message after compression. If optimal compressor exists, the message size can be $M = mH(p)$, where p is the probability for a bit to be ‘0’, and $H(p) =$

$-p\log_2 p - (1-p)\log_2(1-p)$ is the entropy function. The authors of CBF suggests to use arithmetic coding [113, 156] to achieve near-optimal compression.

It has been proven that the number of hash functions that minimizes the false positive rate without compression in fact maximizes the false positive rate with compression. The number k of hash functions that minimizes the false positive ratio with compression is either 0 or infinity [111]. As we need at least one hash function, and it is impractical to have infinite number of hash functions, we use $k = 1$ as the optimal value.

When $k = 1$, the probability p for a bit to be '0' is,

$$p = e^{-kn_r/m} = e^{-n_r/m}, \tag{5–6}$$

and the false positive ratio of CBF is,

$$f_{CB} = (1-p)^k = 1-p. \tag{5–7}$$

Adopting (5–6) and (5–7) to $f_{CB} \le \alpha$, we have,

$$m \ge -\frac{n_r}{\ln(1-\alpha)}. \tag{5–8}$$

From $M = mH(p)$, we have,

$$\begin{aligned} M = mH(p) &= m[-p\log_2 p - (1-p)\log_2(1-p)] \\ &= \frac{ne^{-n/m}}{\ln 2} - m(1-e^{-n/m})\log_2(1-e^{-n/m}). \end{aligned} \tag{5–9}$$

From (5–9), we can numerically compute the minimum message size M, or we can take the following approximation form where m is minimized:

$$M = mH(p)|_{m=-n_r/\ln(1-\alpha)} = \frac{n_r}{\ln 2}\left[1-\alpha+\frac{\alpha\ln\alpha}{ln(1-\alpha)}\right]. \tag{5–10}$$

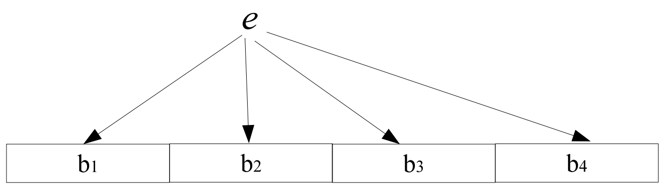

Figure 5-1. The partitioned Bloom filter with $k = 4$ segments.

5.4 Adaptive Bloom filter (ABF)

5.4.1 Motivation

From (5–3), we can see that the optimal false positive ratio of a Bloom filter depends only on m and n [3] . The Compressed Bloom filter (CBF) increases the actual size m of the Bloom filter to achieve lower false positive. This inspired us: Can we decrease the number n of elements instead? If we can, how can we achieve it?

First let's see what n really means. In the context of this chapter, the elements we are interested in are those in both sets. In other words, elements not belonging to the intersection all contribute to false positives. Let us name the elements in the intersection T as candidate elements, and those in $S_l \cup S_r - T$ as non-candidate elements. In both the Bloom filter approach and CBF approach, a portion of non-candidates are encoded in B_r or CB_r. As (5–3) suggests, the false positive ratio of a BF or CBF is bigger when the number of elements encoded increases. In a perfect scenario, we should let B_r or CB_r only encode elements in T, whose false positive ratio will be theoretically the lowest. But how can we achieve it in practice?

In order to reduce n, we first need to separate the membership bits of the elements. Our approach is inspired by the partitioned Bloom filter. In a partitioned Bloom filter, the bit array is divided into k equal-sized sub-arrays. Each element maps to k bits, one in each segment, as illustrated in Figure 5-1. In a partitioned Bloom filter, segments are

[3] In practice, the hash functions used can also affect the false positive ratio. Imperfect hash functions incurs higher false positive ratio than theoretical expectations [92].

independent: each segment independently provide some filtering power to eliminate non-members. If an element is eliminated with the first i segments, we don't need to check the $i+1$th segment. Thinking this reversely, if know that an element is not in T in the ith segment, we don't need to encode it to the segments followed. This leads to our solution.

5.4.2 Description

We first introduce a simplified version of our approach, then we generalize it to optimize the communication cost for arbitrary set sizes. Our solution works in iterations. Each iteration the local host constructs a segment of partitioned Bloom filter, i.e. a bit map, with its remaining elements. We denote it as b_{li}, where i is the iteration number. Let n_l be the number of elements in the local host. We use $\frac{n_l}{\ln 2}$ bits in the bit map so that each bit in b_{li} is '1' with a probability of $\frac{1}{2}$. The local host send b_{li} to the remote host. The remote host use b_{li} to filter its elements: if an element is mapped to '1' in b_{li}, it stays; otherwise it is eliminated from the intersection. The probability for a non-candidate to be eliminated is the same as the probability for a bit in b_{li} to be '0', which is $\frac{1}{2}$. After the elimination, the number of elements left in the remote host is roughly $n_c + \frac{n_r - n_c}{2}$, where $n_c = |T|$ is the number of common elements. The remote host encodes remaining elements into another bit map (using different hash seed) b_{ri} using $(n_c + \frac{n_r - n_c}{2}) / \ln 2$ bits so that b_{ri} has roughly the same number of '0's and '1's. Then the remote host sends b_{r_i} back to the local host. On receiving b_{ri}, the local host then perform the same elimination on its remaining elements. After $-\log_2 \alpha$ iterations, it will meet the required false positive ratio. Figure 5-2 shows an illustration of this approach.

However, when one set is much larger than the other set, the above solution is not optimal. We need bigger filtering power than $\frac{1}{2}$ to quickly rule out non-candidate elements from the large set.

Therefore, we propose a generalization of above iterative bit map approach, called the "adaptive Bloom filter (ABF)". Instead of exchanging bitmaps in each iteration, we

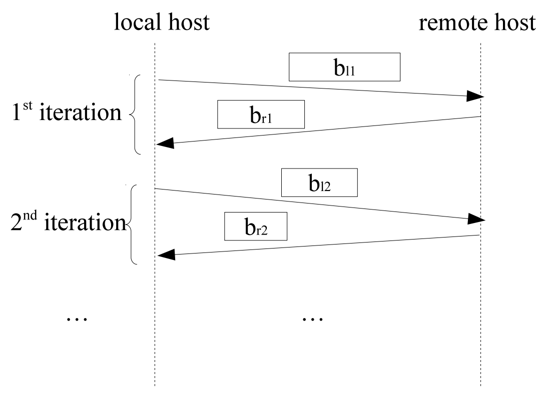

Figure 5-2. A simplified adaptive Bloom filter.

exchange Bloom filters. Note that we may also send multiple bit maps, but a Bloom filter with k hash functions is more powerful than k bitmaps with the same overall space.

5.4.3 Analysis

In an iterative bit map, all bit maps transmitted have the same property: a bit is ‘1’ with a probability of $\frac{1}{2}$. After each iteration, the local host receives a bit map from the remote host. The filtering power of the bit map is $\frac{1}{2}$. The probability for a non-candidate element in S_l to remain un-eliminated after the kth iteration is $(\frac{1}{2})^k$. Consider the false positive constraint, we have, $(\frac{1}{2})^k \le \alpha$, So,

$$k \ge \log_{\frac{1}{2}} \alpha = -\log_2 \alpha, \tag{5–11}$$

where k is the number of iterations.

In each iteration, about half of the non-candidate members can be eliminated. Let n_{li} be the number of non-candidate elements left in the local host after the ith iteration. Then $n_{li+1} = \frac{1}{2} n_{li}$. The number of bits that the local host sends in the ith iteration is $\frac{n_c + n_{li}}{\ln 2}$; As a result, the total number of bits that the local host sends in k iterations is,

$$M_{sent} = \sum_{i=1}^{k} \frac{1}{\ln 2}\left(n_c + \frac{n_l - n_c}{2^{i-1}}\right) = \frac{k n_c}{\ln 2} + \frac{2^k - 1}{2^{k-1} \ln 2}(n_l - n_c). \tag{5–12}$$

Similarly, let n_r be the number of elements in S_r. We can find the number of bits received by the local host, which is,

$$M_{received} = \sum_{i=1}^{k} \frac{1}{\ln 2}\left(n_c + \frac{n_r - n_c}{2^i}\right) = \frac{kn_c}{\ln 2} + \frac{2^k - 1}{2^k \ln 2}(n_r - n_c). \tag{5–13}$$

Therefore, the total message size is,

$$\begin{aligned} M &= M_{sent} + M_{received} \\ &= \frac{2kn_c}{\ln 2} + \frac{2^k - 1}{2^k \ln 2}(2n_l + n_r - 3n_c) \\ &\approx \frac{1}{\ln 2}\left[2n_l + n_r + (2k-3)n_c\right] \\ &= \frac{1}{\ln 2}\left[2n_l + n_r + (-2\log_2 \alpha - 3)n_c\right]. \end{aligned} \tag{5–14}$$

From the asymmetry in (5–14), we can see that the local hosts contribute more to the final message size. This is because the local host sends a filter out first, while the remote host is able to filter out half of its non-candidate before its first transmission. In practice, if $n_l > n_r$, we may ask the remote server to send its bitmap first. In that case, the larger set (S_l) will be reduced first and the overall message size is reduced. Another observation is that this approach is sensitive to the size of common elements. Intuitively, elements in the common set will never be eliminated. So they are encoded into the message in all the iterations. However, in a lot of applications n_c is very small comparing to n_l and n_r. One example is in the scanner detection the proportion of scanners is actually very small compared to the normal users. The other example is the search results of multiple key words are much fewer than that of each single one [23, 132].

Now, if Bloom filters are exchanged instead of bit maps, we have two series of new variables: k_{li} and k_{ri}, where k_{li} represents the number of hash functions for the Bloom filter that the local host send in the ith iteration, while k_{ri} stands for the number of hash functions for the Bloom filter that the remote host replies in the ith iteration. We have

$\sum_{i=1}^{t} k_{ri} = k$, where t is the number of iterations. Now, we have an optimization problem as follows:

Optimize M, with constraints:

$$\begin{cases} M = \frac{1}{\ln 2}\sum_{i=1}^{t}(n_{li} + n_c) \times k_{li} + \frac{1}{\ln 2}\sum_{i=1}^{t}(n_{ri} + n_c) \times k_{ri}; \\ n_{l1} = n_l; \\ n_{r0} = n_r; \\ n_{li+1} = n_{li} \times fp_{ri} + \hat{n_c}, \text{ for } 1 \le i < t; \\ n_{ri+1} = n_{ri} \times fp_{li+1} + \hat{n_c}, \text{ for } 0 \le i < t; \\ fp_{ri} = (\frac{1}{2})^{k_{ri}}; \\ fp_{li} = (\frac{1}{2})^{k_{li}}; \\ \sum_{i=1}^{t} k_{ri} = k; \\ k \ge -\log_2 \alpha; \\ k_{li} \ge 1, \text{ for } 1 \le i \le t; \\ k_{ri} \ge 1, \text{ for } 1 \le i \le t; \\ k_{l1} = 1; \end{cases}$$

The optimal setting k_{li} and k_{ri} depends on size n_c of the intersection set. However, it is not known as a prior. Therefore, the last condition in the constraints is there for estimating n_c. We may use the methods mentioned in [86] to compute an approximate intersection size with the AND of two bloom filters. As more Bloom filters are exchanged, the estimation can become more precise. If necessary, we can re-compute the optimal parameters on the way.

5.5 Summary

In this chapter, we study the distributed join problem, which is to find out common elements between two sets in distributed systems. We introduce the adaptive Bloom filter – a new family of Bloom filter variant which is designed for data sharing via network. The idea is to iteratively eliminate non-candidates. We transform the problem into an optimization problem.

CHAPTER 6
USING PARTITIONED BLOOM FILTERS FOR INFORMATION COLLECTION IN RFID SYSTEMS

6.1 From Space-Time Efficiency to Time-Energy Efficiency

Switching our perspective from space domain to time domain, and from time domain to energy domain, we found that space-time efficient data structure can also be used to design energy-time efficient RFID protocols. In a typical computer/network system, information is stored in memory as 0/1 bits. While in a wireless environment such as an RFID system, information is sent/received via continuous time slots while the protocol is executing. The length of overall time slots defines the execution time of the protocol. As a result, the space domain in computer/network systems becomes time domain in RFID systems. Meanwhile, RFID tags send/receive information to/from one or more time slots. The amount of data that a tag send/receive determines its energy expenditure. Therefore, the time domain in computer/network systems (accessing/updating the data structure) can be mapped to the energy domain of RFID tags (sending/receiving information to/from time slots). As a result, designing an time-energy efficient RFID protocol shares the same discipline as implementing a space-time data structure in a computer/network system.

6.2 RFID Background, Model, and Problem Definition

6.2.1 RFID System

Traditional barcodes can only be read in close ranges. RFID tags replace barcodes with electronic circuits that can transmit identification numbers wirelessly over a distance. The longer operational range makes them popular in automatic transportation payments, object tracking, and supply chain management [72, 119, 149]. A typical RFID system consists of one or multiple readers and numerous tags. Each tag carries a unique identifier (ID). Tags do not communicate amongst themselves; they communicate directly with the reader.

Passive tags are most widely used today. They are cheap, but do not have internal power sources. They rely on radio waves emitted from the reader for power, and have small operational ranges of a few meters, which seriously limit their applicability. For example, consider a large warehouse in a distribution center of a major retailer, where hundreds of thousands of tagged commercial products are stored. In such an indoor environment, if we use passive tags, hundreds of RFID readers may have to be installed in order to access tags in the whole area, which is not only costly but also causes interference when nearby readers communicate with their tags simultaneously. It is not a good solution to use a mobile reader and walk through the whole area whenever we need information from tags. To automate warehouse management in large scale, a much better choice is to use battery-powered active tags because of their long transmission ranges. The lifetime of these tags is determined by how their battery power is used. Energy conservation must be one of the top priorities in any protocol design that involves active tags.

With richer on-tag resources, active tags are likely to gain more popularity in the future, particularly when their prices drop over time as manufactural technologies are improved and markets are expanded. These tags can be integrated with miniaturized sensors [112, 119, 130, 136]. Not only will they report their IDs, but also they can report dynamic, real-time information about the operation status of the tags or the conditions of the environment.

We study a general problem of how to design efficient information collection protocols to collect information from a subset of tags in a large RFID system. For example, consider a large chilled food storage facility, where each food item is attached with a RFID tag that has a thermal sensor. A RFID reader periodically collects temperature readings from tags to check whether any area is too hot (or too cold), which may cause food spoil (or energy waste).[1] Because each area in the facility may be packed with many food items, the temperature readings from these close-by tags are highly redundant. Hence, it is not necessary for the reader to collect information from all tags in the system. The reader may select a subset of tags each time to collect temperature information. In another example, a RFID reader periodically accesses the residual energy levels of on-tag batteries to see if some tags (or their batteries) need to be replaced. If the reader has information about which tags are new and which ones are old, it may choose to only query the old tags. As we will see later, it costs less energy to query a smaller number of tags. On the other hand, it is a harder problem to collect information from a subset of tags than from all tags because the reader has to make sure that tags that are not under query do not transmit — their transmissions will interfere with the transmissions made by tags of interest, causing unnecessary energy waste.

Much existing research focused on designing ID-collection protocols that read IDs from tags in a RFID system [1, 6, 19, 78, 115, 137, 140, 154, 159, 169, 170, 174]. In recent years, some interest is shifted to other functions such as estimating the number of tags in a system [55, 71, 72, 89, 129, 154, 167, 172, 173], detecting the missing tags [83, 85, 100–102, 149] or misplaced tags [14], and tag authentication and privacy [42, 87, 95, 161]. The primary performance objective in most papers is to minimize the execution time it takes a protocol to read all tag IDs or perform other functions. Energy efficiency, particularly, how to reduce energy consumption by the tags, is an under-studied subject. There exists prior work on energy-efficient protocols for estimating the number of tags [88], or anti-collision protocols that minimize the energy consumption of a mobile reader [70, 116]. To the best of our knowledge, we were the

[1] If a tag reports an abnormal temperature, the reader may instruct the tag to keep transmitting beacons, which guide a mobile signal detector to locate the tag.

first to investigate energy-efficient protocols for collecting information from a subset of tags in a large RFID system [**?**].

In this chapter, we first show that the standard, straightforward polling design is not energy-efficient because each tag has to continuously monitor the wireless channel and receive $O(m)$ tag IDs, which is energy-consuming if the number m of tags that the reader needs to collect information from is large. We then show that a coded polling protocol (CP) is able to cut the amount of data each tag has to receive by half, which means that energy consumption per tag is also reduced by half. This is still far away from our objective of reducing energy consumption to $O(1)$. We propose a novel tag-ordering polling protocol (TOP) that can reduce per-tag energy consumption by more than an order of magnitude when comparing with the coded polling protocol. We also reveal an energy-time tradeoff in the protocol design: per-tag energy consumption can be reduced to $O(1)$ at the expense of longer protocol execution time. We then apply partitioned Bloom filters to enhance the performance of TOP, such that it can achieve much better energy efficiency without degradation in protocol execution time. Finally, we show how to configure the new protocols for time-constrained energy minimization.

6.2.2 System Model

We consider a large RFID system using active tags. Each tag carries a unique ID and one or more sensors. It also has the capability of performing certain computations as well as communicating with the RFID reader wirelessly. The reader and the tags transmit with sufficient power such that they can communicate over a long distance. We assume that the RFID reader knows the IDs of all tags in the system by executing an ID-collection protocol, and it has enough power supply.

6.2.3 Problem Definition

Let N be the set of tags in the system and $n = |N|$. Let M be a subset of tags, $m = |M|$, and $M \subseteq N$. Our objective is to design efficient information collection protocols that collect information from tags in M. An information collection protocol may be scheduled to execute periodically. M may change over time so that different subsets of tags are queried. We have two performance objectives. The primary performance objective is to achieve energy efficiency. We want to minimize the average amount of energy that a tag spends during one execution of an information collection protocol. The energy expenditure by a tag has two components: (1) energy for transmitting its information (e.g., 32 bits) to the reader, and (2) energy for receiving the polling request and other information from the reader. The former is a small, fixed amount of energy that must be spent. The latter is dependent on the protocol design as we will see shortly. It is a variable amount of energy that should be minimized. Simple protocol designs will result in all tags in the system, including those not in M, to be continuously active and unnecessarily receive a large amount of data from the reader for an extended period of time. How to minimize such energy cost is the focus of this chapter.

Our secondary performance objective is to reduce protocol execution time. RFID systems use low-rate communication channels. For example, in the Philips I-Code system, the rate from a reader to a tag is about 27Kbps and the rate from a tag to a reader is about 53Kbps. Low rates, coupled with a large number of tags, often cause long execution times for RFID protocols. To apply such protocols in a busy warehouse environment, it is desirable to reduce protocol execution time as much as possible.

Communication between the reader and tags is time-slotted. The reader's signal synchronizes the clocks of tags. Let t_{tag} be the length of a time slot during which the reader is able to broadcast a tag ID, and t_{inf} be the length of a time slot during which a tag is able to transmit its information.

6.2.4 Prior Art

Much existing work on RFID systems is to design anti-collision ID-collection protocols, which read IDs from all the tags in the system. They mainly fall into two categories. One is ALOHA-based [19, 78, 137, 140, 154, 170], and the other is *Tree-cased* [1, 6, 115, 174]. The *ALOHA-based protocols* work as follows: The reader broadcasts a query request. With a certain probability, each tag chooses a time slot in the current frame to transmit its ID. If there is a collision, the tag will continue participating in the next frame. This process repeats until all tags are identified successfully.

The tree-based protocols organize all IDs in a tree of ID prefixes [6, 115, 174]. Each in-tree prefix has two child nodes that have one additional bit, '0' or '1'. The tag IDs are leaves of the tree. The RFID reader walks through the tree. As it reaches an in-tree node, it queries for tags with the prefix represented by the node. When multiple tags match the prefix, they will all respond and cause collision. Then the reader moves to a child node by extending the prefix with one more bit. If zero or one tag responds (in the one-tag case, the reader receives an ID), it moves up in the tree and follows the next branch. Another type of tree-based protocols tries to balance the tree by letting the tags randomly pick which branches they belong to [1, 18, 115].

ID-collection protocols can be adopted to solve our problem, but they are inefficient comparing to our approaches. For ALOHA-based protocols, tags can send sensor information along with their IDs. But each tag need to continuously listen to the channel and may send its sensor information multiple times due to collision, which results in at lease $O(n)$ per-tag energy consumption. For tree-based protocols, reader can use an additional time slot to receive a tag's information when it becomes the sole leaf of a branch. However, all tags need to receive at least $O(m)$ prefixes, which is equivalent to the Basic Polling protocol design in Section 6.3.

Shen et al. proposed a data collection protocol in hybrid mobile networks consisting of wireless sensor nodes, RFID readers, and smart RFID tags [139]. Zheng and Li proposed a two-phase fast tag search protocol to find the subset of wanted tags in readers' interrogation zone [171]. Chen, Zhang and Xiao proposed efficient information collection protocols for sensor-augmented RFID networks [28]. Yue et al. designed efficient protocols to collect sensor information from RFID tags in multi-reader scenarios

[165]. A reader first detects which tags are located in its interrogation region. Then it broadcasts a distributively constructed Bloom Filter to tags and wait for tags' response. A follow up work [166] further analyzes the impact of cardinality estimation error and channel error to the execution time / overhead of the protocol. Min et al. proposed an iterative tag search protocol, which uses filtering vectors to iteratively filter out tags not of interest [24]. These methods however only improves time efficiency [24, 28, 139, 165, 166, 171] or infrastructure-cost efficiency [139]. Tag energy consumption are not considered. Our proposed methods on the other hand, minimizes tag-energy consumption while constraining execution time.

6.3 Basic Polling Protocol (BP)

In a standard, straightforward way of designing a polling protocol, we simply let the RFID reader broadcast the tag IDs in M one by one. After it transmits an ID, it waits for a time slot of t_{inf} during which the corresponding tag transmits its information. Each tag continuously listens to the wireless channel. Whenever it receives an ID from the reader, the tag compares the received ID with its own ID. If they match, the tag will transmit its information and then go to sleep until the next scheduled execution of the protocol.

In the above protocol, each tag in M will have to receive $\frac{m}{2}$ IDs on average from the reader before it transmits. Each tag not in M will have to receive all m IDs. The amount of energy spent by a tag in receiving such data grows linearly with respect to m. It takes a constant amount of energy for a tag to receive an ID and another constant amount of energy for it to transmit its information. The energy cost of the whole system is thus $O(nm)$. The protocol execution time is $m(t_{tag} + t_{inf})$.

We use a numerical example to explain the energy cost. Consider a military base that has a large warehouse storing 50,000 weapons, ammunition magazines, and other equipment, which are tagged with RFID sensors. Among them, there are 1,000 sensitive devices, from which a RFID reader needs to access information in order to make sure that they are in good conditions or simply to confirm their presence (against unauthorized removal). Let e_r be the amount of energy a tag spends in receiving an ID and e_s be the amount of energy a tag spends in transmitting its information. The total energy consumed by all tags for transmitting is $1,000e_s$, and the total energy consumed by all tags for receiving is about $50,000,000e_r$. Even though e_r may be smaller than e_s, the total amount of energy spent by tags in receiving can be much greater than the amount spent in transmitting.

6.4 Coded Polling Protocol

We show that a coded polling protocol (CP) [25] is able to reduce the amount of data each tag has to receive by half. The protocol assumes that each tag ID carries an identification number and a CRC (cyclic redundancy code) for error detection. This requirement is satisfied by the EPCglobal Gen-2 standard, where each 96-bit tag ID contains a CRC checksum. The CRC is computed based on the identification number and a generator. When a tag receives an ID from a wireless channel, it computes a CRC based on the received identification number and then compares the result with the received CRC. If they are the same, we say the ID contains a valid CRC.

CRC has the following property: If x and y are two tag IDs with valid CRCs, then $x \oplus y$ also has a valid CRC. The same property does not hold for $x \oplus \hat{y}$, where $\hat{y}$ contains the same bits in y but in the reverse order. For example, if $y = 10110$, then $\hat{y} = 01101$. We call $\hat{y}$ the reversal of y.

In the coded polling protocol, the RFID reader first arranges the IDs in M in pairs. Each pair consists of two IDs that are arbitrarily selected from M. Consider an arbitrary pair, x and y, which are called each other's paring ID. We define the polling code of the pair as $c = x \oplus \hat{y}$.

Instead of sending out the IDs in M one after another, the reader broadcasts the polling code of each pair one after another. After each broadcast of a polling code $c = x \oplus \hat{y}$, the reader waits for two time slots, during which tag x and tag y will transmit. More specifically, when an arbitrary tag z receives the polling code c, it first computes

$z \oplus c$, and checks whether the CRC in the reversal of $z \oplus c$ is valid. If it is, the tag will transmit its information. Otherwise, the tag computes $\hat{z} \oplus c$, and checks whether the CRC in $\hat{z} \oplus c$ is valid. Again, if it is valid, the tag will transmit. Otherwise, the tag will not transmit. We show that only tag x and tag y will transmit.

First, consider the case of $z = x$. The tag first computes $z \oplus c = x \oplus x \oplus \hat{y} = \hat{y}$. The reversal of $\hat{y}$ is y. The CRC in any tag ID (including y) is valid. Hence, tag x will transmit. Moreover, it now knows its pairing ID, y. If x is greater than y, the tag will transmit in the first slot after receiving the polling code; otherwise, it will transmit in the second slot.

Second, we consider the case of $z = y$. The tag first computes $y \oplus c = y \oplus x \oplus \hat{y}$. Its reversal is likely to have an invalid CRC; the chance for an arbitrary number to contain a valid CRC is very small. Then, the tag computes $\hat{z} \oplus c = \hat{y} \oplus x \oplus \hat{y} = x$, which contains a valid CRC. Consequently, y will transmit. Since it now knows its pairing ID, x, it also knows in which slot it should transmit.

Finally, consider the case of $z \neq x$ and $z \neq y$. The tag computes the reversal of $z \oplus c = z \oplus x \oplus \hat{y}$ and then computes $\hat{z} \oplus c = \hat{z} \oplus x \oplus \hat{y}$. Both of them are likely to have invalid CRCs.

A minor problem is that $y \oplus c$ in the second case and $z \oplus c$ or $\hat{z} \oplus c$ in the third case still have a small probability to contain a valid CRC. However, the reader can easily prevent this from happening. It knows all tag IDs. It can precompute all polling codes and check whether a valid CRC happens in the above cases by chance when it is not supposed to. If this is true for a pair of tags, x and y, the reader must break up the pair, and use them to form new pairs with other IDs in M. Such an approach is effective because the probability for this to happen is exceedingly small when CRC is sufficiently long.

Because each polling code represents two tag IDs, the number of polling codes in CP is $\frac{m}{2}$. Hence, when comparing with the basic polling protocol, CP reduces the number of broadcasts made by the reader by half, and it also reduces the amount of data that each tag has to receive by half. This not only saves energy for tags, but also reduces the protocol execution time to $\frac{m}{2}t_{tag} + mt_{inf}$.

6.5 Tag-Ordering Polling Protocol (TOP)

Although CP is more efficient, the expected amount of energy that each tag spends in receiving remains $O(m)$. In this section, we propose a new tag-ordering polling protocol that reduces such energy cost to $O(1)$.

6.5.1 Motivation

In the basic polling protocol, a RFID reader broadcasts m IDs in time slots of length t_{tag}. All tags must continuously monitor the wireless channel in order to know whether their own IDs are in the broadcast. In CP, the reader broadcasts $\frac{m}{2}$ polling codes also in time slots of length t_{tag}. Again, all tags must continuously monitor the wireless channel. They have to keep receiving and processing the polling codes. Each tag in the basic protocol has to receive up to m IDs. Even though CP is more efficient, a tag still has to receive up to $\frac{m}{2}$ codes.

We want to remove the necessity for any tag to keep monitoring the wireless channel. Ideally, a tag should stay in an energy-conserving standby mode for most of time, and only wake up at the right time slot to receive information about itself, such as whether it is polled and, if so, when it should transmit. To further reduce the amount of data that tags have to receive, we let the reader broadcast a so-called reporting-order vector V, instead of IDs in M. Each ID in M is mapped to a bit in V through a hash function; the bit is set as one to encode the ID in the vector. A tag only needs to check a specific bit in V at a location determined by the hash of its ID. This bit is called the representative bit of the tag. If its value is one, the tag is polled by the reader for reporting, i.e., the tag belongs to M; if its value is zero, the tag is not polled. The vector V also carries information about the order in which the polled tags will report their data. Each bit whose value is one in V represents a polled tag. If a tag finds that there are i ones in V preceding its representative bit, it knows that it should be the $(i + 1)$th tag in

M to report its information. With such an ordering, it becomes possible for tags in M to report at different times and avoid collision.

However, this basic idea has two problems. First, there should be at least m bits in V to encode m IDs in M. The energy cost of receiving V remains $O(m)$. How can a tag find out the number of ones in V preceding its representative bit without having to receive the whole vector? Second, hash collision causes two issues. If a tag not in M is hashed to the same bit in V as a tag in M does, it will find its representative bit to be one, causing false positive. If two tags in M are mapped to the same bit in V, they will transmit at the same time, causing report collision. In the rest of this section, we design a new tag-ordering polling protocol (TOP) to solve these problems. It consists of three phases: ordering phase, polling phase, and reporting phase. In the ordering phase, the reader broadcasts the vector V so that each tag knows whether it is polled and where it is located in the reporting order. The polling phase resolves the issues of false positive and report collision. Finally, in the reporting phase, tags in M report their information in the order defined by V without collision.

6.5.2 Protocol Description

6.5.2.1 Ordering Phase

The RFID reader does not broadcast any IDs or indices. It only broadcasts the reporting-order vector, V. If V cannot fit in one time slot of length t_{tag}, the reader breaks the vector into segments and broadcasts each segment in a time slot of t_{tag}. In addition, the reader also broadcasts the vector size v.

Knowing the vector size, a tag t is able to hash its ID and find out the location of its representative bit in V. Because the segment size is fixed, t also knows which segment its representative bit belongs to. This segment, denoted as V_t, is called the representative segment of tag t. A tag will stay in the standby mode and be active only when receiving its representative segment.

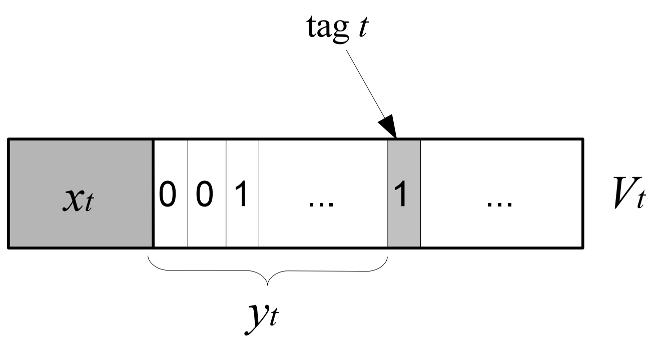

Figure 6-1. V_t is the representative segment of tag t, x_t is the total number of ones in all previous segments, and y_t is the number of ones in V_t that precede tag t's representative bit. l_t is the position of t in the reporting order. $l_t = x_t + y_t$.

If a tag finds that its representative bit is zero, it knows for sure that it is not a member in M. If a tag finds that its representative bit is one, it may be a member in M or a non-member that is mapped to a bit which a member in M is also mapped to. The latter case causes false positive. Because the reader knows all IDs in the system, it can pre-compute the set F of non-member tags that cause false positive.

When the reader broadcasts any segment of V, it includes in the same time slot the total number of ones in the previous segments. For an arbitrary tag t, let l_t be the number of ones in V preceding the representative bit of t. When tag t receives V_t, it can computes l_t as the sum of (a) the number of ones in the previous segments and (b) the number of ones in V_t before its representative bit. See Figure 6-1 for illustration. As we will see later, the value of l_t specifies when tag t will transmit during the reporting phase.

If two tags in M are mapped to the same bit in V, they will have the same l_t value and thus transmit at the same time during the reporting phase, causing collision. Because the reader has all IDs in M, it knows exactly which tags will be mapped to the same bit. This makes it easy to resolve collision. The reader simply removes all but one tag that are mapped to a bit, and puts them in a set C. These tags, together with tags in F, will not participate in the reporting phase. They are handled separately in the polling phase.

6.5.2.2 Polling Phase

In this phase, the reader issues two types of polling requests. For each tag in C, it sends a positive polling request. For each tag in F, it sends a negative polling request. To distinguish these two types, the reader must transmit a one-bit flag together with a tag ID in each request, specifying whether the polling is positive or negative and which tag is polled.

Tags that find their representative bits to be ones in the previous phase must continuously listen to the channel during the polling phase. After sending a positive request, the reader waits for a time slot to receive information. The tag that finds its ID in the request will transmit its information in this slot. This tag, which belongs to C, will not participate in the reporting phase. After sending a negative request, the reader does not wait before sending out the next request. The tag that finds its ID in a negative request knows that it must belong to F and hence should not further participate in the protocol execution.

The total number of polling requests is $|F| + |C|$. By choosing an appropriate size for the reporting-order vector, we can make sure that $|F| + |C| = O(1)$ (see Section 6.6). Note that only tags in M and F have to listen to the channel in this phase. Tags in $N - M - F$, which may contain the majority of tags in the system, have already known that they do not belong to M and thus do not need to participate in the protocol execution.

6.5.2.3 Reporting Phase

A tag participates in the reporting phase only if it satisfies the following two conditions: (1) it finds that its representative bit is one in the ordering phase, and (2) it does not find its ID in the requests of the polling phase.

The reporting phase consists of $m - |C|$ time slots. In each time slot, one tag in $M - C$ transmits its information. Recall that each tag in M learns its index in the reporting order during the ordering phase. The tag will transmit in the reporting phase at the time slot of the same index.

6.5.2.4 Timing

Before executing the protocol, the RFID reader uses its broadcasting signal to synchronize the clocks of the tags. The reader computes the vector V and breaks it into segments. Suppose each time slot of length t_{tag} can carry 96 bits. We may set the segment size to be 80 bits and use the remaining 16 bits to carry the total number of ones in the previous segments.[2] The reader is able to compute the execution time T_1 of the ordering phase, which is the number of segments multiplied by t_{tag}.

Since the reader knows all IDs in the system, it can precompute the set F of tags that cause false positive and the set C of tags that should not participate in the reporting phase in order to avoid collision. Based on F and C, the reader can compute the execution time T_2 of the polling phase, which is $|F| \times t_{tag} + |C| \times (t_{tag} + t_{inf})$.

Suppose all tags wake up at each scheduled execution of the protocol. The reader computes and broadcasts the values of T_1 and T_2 right before the ordering phase, so that the tags know when each phase of the protocol will begin. They will remain in the standby mode unless they have to receive their representative segments, participate in the polling phase, or transmit their information in the reporting phase.

If the system requires on-demand polling of tag information instead of periodic execution, there are two possible solutions to wake the tags up in the first place. The first one is ”pseudo-on-demand” polling, where tags still wake up periodically, but the reader only issues the polling request when needed. The second approach is to attach a wake-up circuit to each tag, and use the two-stage wake-up scheme proposed in [29]

[2] Using 16 bit to carry the number of ones in previous segments will limit the value of m to (0, 65,535]. To get rid of this limitation, we can use $\lceil \log_2 m \rceil$ bits instead and broadcast the value of $\lceil \log_2 m \rceil$ to tags at the beginning of protocol. However, for the sake of simplicity, we use 16 bits in the following to help demonstrate the main idea.

to activate the tags. In this approach, tags responde almost immediately to the polling event. However, the wake-up circuit requires the reader to be close enough so that the radio power is strong enough to trigger the wake-up event. As a result, we may have to deploy extra readers to cover all the tags.

6.6 Performance Analysis of TOP

6.6.1 Energy Cost

We show how to configure TOP such that the energy cost per tag is $O(1)$. The energy cost of a tag has four components: (1) receiving v, T_1 and T_2, (2) receiving a segment of V in the ordering phase, (3) listening to the channel during the polling phase, and (4) transmitting information in a slot at the reporting phase (or at the polling phase if the tag is in C). The first two components incur small, constant energy expenditure to every tag in the system. The fourth component also incurs small, constant energy cost, but only to the tags in M. The third component incurs energy cost only to tags in F and M. In the worse case, a tag has to listen to all $|C| + |F|$ polling requests from the reader. Suppose it takes one unit of energy to receive a polling request. The total energy cost of a tag, denoted as Ω, is

$$\Omega \leq |C| + |F| + O(1). \tag{6–1}$$

We treat $|C|$ and $|F|$ as random variables and derive their expected values. Recall that v be the number of bits in the reporting-order vector V. Let b_i be the value of the ith bit in V, $0 \leq i < v$. For each tag in M, the reader maps it to a random bit in V and sets the bit to one. After encoding all m tags in V, the probability for b_i to be one is

$$Prob\{b_i = 1\} = 1 - (1 - \frac{1}{v})^m \approx 1 - e^{-m/v}. \tag{6–2}$$

The bits, b_0, b_2, ..., b_{v-1}, are independent of each other. Thus, the expected number of ones in V is $\sum_{i=1}^{v} Prob\{b_i = 1\}$. The value of $|C|$ is equal to m subtracted by the number of ones in V. Hence, we have

$$E(|C|) = m - \sum_{i=1}^{v} Prob\{b_i = 1\} \approx m - v(1 - e^{-m/v}). \tag{6–3}$$

A tag not in M will cause false positive when its representative bit is one. The probability for this to happen is $Prob\{b_i = 1\}$. Hence,

$$E(|F|) = (n - m)Prob\{b_i = 1\} \approx (n - m)(1 - e^{-m/v}). \tag{6–4}$$

Both $E(|C|)$ and $E(|F|)$ are monotonically decreasing functions of v. We show that $E(|C|) = O(1)$ if v is sufficiently large. Let $v = \frac{m^2}{2}$. From Taylor expansion, we know that

$$\begin{aligned} 1 - e^{-m/v} &= \frac{m}{v} - \frac{1}{2!}(\frac{m}{v})^2 + \frac{1}{3!}(\frac{m}{v})^3 - \frac{1}{4!}(\frac{m}{v})^4 ... \\ &\geq \frac{m}{v} - \frac{1}{2!}(\frac{m}{v})^2. \end{aligned}$$

Applying it to (6–3), we have

$$E(|C|) = m - v(1 - e^{-m/v}) \leq \frac{1}{2!}\frac{m^2}{v} = 1. \tag{6–5}$$

Next we show that $E(|F|) = O(1)$ if v is sufficiently large. If $n = m$, $E(|F|) = 0$. Now assume $n > m$. Let $v = -\frac{m}{\ln(1-\frac{1}{n-m})}$. Applying it to (6–4), we have

$$E(|F|) = (n - m)(1 - e^{-m/v}) = 1. \tag{6–6}$$

Therefore, if we choose $v = \max\{\frac{m^2}{2}, -\frac{m}{\ln(1-\frac{1}{n-m})}\}$, we have

$$E(\Omega) \leq E(|C|) + E(|F|) + O(1) \leq 1 + 1 + O(1) = O(1).$$

We conclude that TOP can be configured such that the expected energy cost per tag is $O(1)$. As we will see shortly, the protocol execution time increases when v becomes too large. To strike a balance between energy cost and protocol execution time, we may choose a value of v much smaller than $\max\{\frac{m^2}{2}, -\frac{m}{\ln(1-\frac{1}{n-m})}\}$. In Section 6.10, we use simulations to study the performance of TOP under practical values of v. For example, when $v = 24m$, the amount of data that a tag receives in TOP is more than an order of magnitude smaller than what a tag has to receive in CP.

We characterize the energy cost in the polling phase by counting the amount of data (in Kilobits) that a tag has to receive. Numerical results are shown in the first plot of Figure 6-2, where $n = 50,000$ and $m = 5,000, 10,000$, or $25,000$, corresponding to three curves in the plot. Clearly, as v increases, the energy cost decreases.

6.6.2 Execution Time

The protocol execution time also consists of four components. To begin with, it takes the reader a small, constant time to broadcast v, T_1 and T_2. The time for the ordering phase is $\frac{v}{l} t_{tag}$, where l is the segment size. The time for the polling phase is $|F| \times t_{tag} + |C| \times (t_{tag} + t_{inf})$. The time for the reporting phase is $|M - C| \times t_{inf}$. Hence, the total execution time is

$$T = (\frac{v}{l} + |F| + |C|)t_{tag} + m \cdot t_{inf} + O(1). \tag{6–7}$$

From (6–3) and (6–4), the expected protocol execution time is

$$\begin{aligned} E(T) &= \left[\frac{v}{l} + (n - m)(1 - e^{-m/v}) + m - v(1 - e^{-m/v})\right] \cdot t_{tag} + m \cdot t_{inf} + O(1) \\ &\approx \left[\frac{v}{l} + \frac{(n - m)m}{v}\right] \cdot t_{tag} + m \cdot t_{inf} + O(1). \end{aligned} \tag{6–8}$$

The second plot of Figure 6-2 presents the protocol execution time (excluding the constant $O(1)$) when $n = 50,000$, $m = 5,000, 10,000$, or $25,000$, $t_{tag} = 3297\mu s$, and $t_{inf} = 906\mu s$; see Section 6.10 for how they are determined. Interestingly, as v increases, the execution time first decreases and then increases. We can find the optimal value of v that minimizes the execution time from $\frac{\delta E(T)}{\delta v} = 0$.

Combining the results in the first and second plots, we can figure out the tradeoff relation between energy cost and protocol execution time, which is presented in the third plot. As v becomes large, the energy cost decreases at the expense of increased execution time.

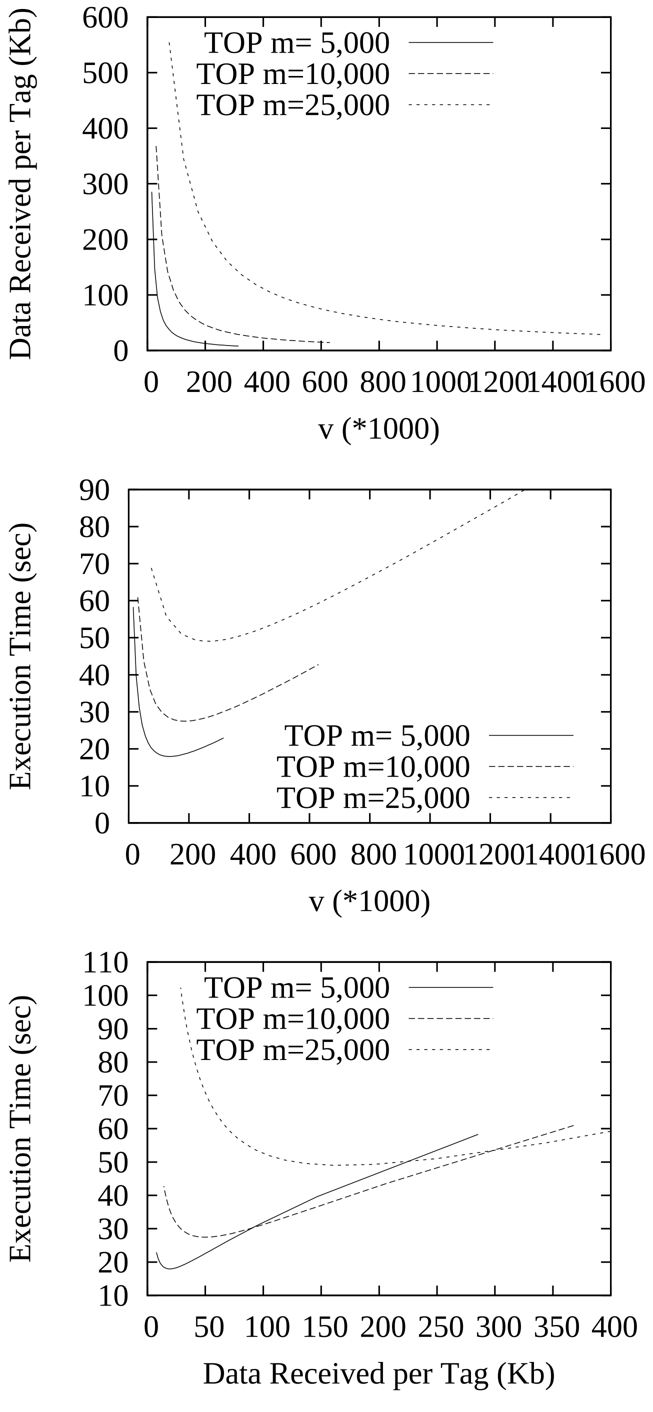

Figure 6-2. Energy, time, and energy-time trade-off of TOP. First plot: Energy cost per tag with respect to v. Second plot: Protocol execution time with respect to v. Third plot: Energy-time tradeoff controlled by v.

6.6.3 Choosing v for Time-constrained Energy Minimization

Recall the performance objectives of TOP are energy efficiency and time efficiency. However, as shown in Figure 6-2, we may not be able to achieve the best performance in both metrics using one configuration. Below we study how to configure TOP for time-constrained energy minimization.

Consider a warehouse with a large number of RFID-tagged goods. Suppose the system administer wants to maximize the tags' battery lifetime, but there is a requirement on the execution time of a polling operation because excessively long execution time increases the chance of interfering with other scheduled tasks. From the previous analysis, we know that the protocol execution time is treated as a random variable. Let T be the execution time of TOP, B be a pre-defined time bound, and α be a probability value, $0 < \alpha < 1$. The time constraint can be specified in a probabilistic way,

$$Prob\{T \leq B\} \geq \alpha. \tag{6–9}$$

Our performance objective is to find the optimal value of v that minimizes the energy cost, subject to the above constraint.

As shown in the first plot of Figure 6-2, the energy cost decreases as the size of the reporting-order vector, v, increases. Hence, our goal becomes finding the largest v that satisfies (6–9). In the following, we derive $Prob\{T \leq B\}$ as a function of v. Based on this function, we will be able to compute the optimal value of v.

Let d be the total number of ones in V after encoding tags in M, $0 < d \leq m$. The probability that x bits are ones, expressed as $Prob\{d = x\}$, can be calculated by the "balls and bins algorithm", which will be given in the next subsection. For now we denote the function for computing $Prob\{d = x\}$ as $p_d(m, v, x)$.

After encoding tags in M, the reader removes colliding tags to C. The value of $|C|$ is equal to m subtracted by the number of ones in V. Hence,

$$Prob\{|C| = c\} = Prob\{d = m - c\} = p_d(m, v, m - c). \tag{6–10}$$

When a tag not in M is mapped to a bit that is one, false positive happens. The reader puts all false positive tags to F. When there are x bits that are ones in V, the conditional false positive probability is $\frac{x}{v}$. Thus,

$$Prob\{\text{false positive} \mid d = x\} = \frac{x}{v}.$$

Obviously, when $d = x$, the total number of false positive tags follows a binomial distribution $Bino(n-m, \frac{x}{v})$.

$$Prob\{|F| = f \mid d = x\} = \binom{n-m}{f}(\frac{x}{v})^f(1-\frac{x}{v})^{n-m-f}.$$

Let S be the union of C and F, so $|S| = |C| + |F|$. The probability distribution of $|S|$ is

$$\begin{aligned} Prob\{|S| = s\} &= \sum_{c=0}^{s} Prob\{|F| = s - c \mid |C| = c\} \cdot Prob\{|C| = c\} \\ &= \sum_{x=1}^{m} Prob\{|F| = s - m + x | d = x\} \cdot Prob\{d = x\} \\ &= \sum_{x=1}^{m} \binom{n-m}{s-m+x}(\frac{x}{v})^{s-m+x}(1-\frac{x}{v})^{n-s-x} p_d(m, v, x). \end{aligned} \tag{6–11}$$

Adopt (6–7) and ignore $O(1)$, a small constant time for the reader to broadcast v, T_1 and T_2, which is negligibly small when comparing with other components on the right side of (6–7). We have

$$Prob\{T \leq B\} = \sum_{s=0}^{s_{max}} Prob\{|S| = s\}, \tag{6–12}$$

where $s_{max} = \frac{B - mt_{inf}}{t_{tag}} - \frac{v}{l}$. We denote the right side of (6–12) as $P_t(v, B)$, which is the probability for the protocol execution time to be bounded by B under a certain value of v. It is computable as a function of v and B after (6–11) is applied and parameters m and n are given.

We want to find the largest value of v that satisfies the inequality, $P_t(v, B) \geq \alpha$. Our numerical computation shows that, given a fixed value of B, $P_t(v, B)$ is not a

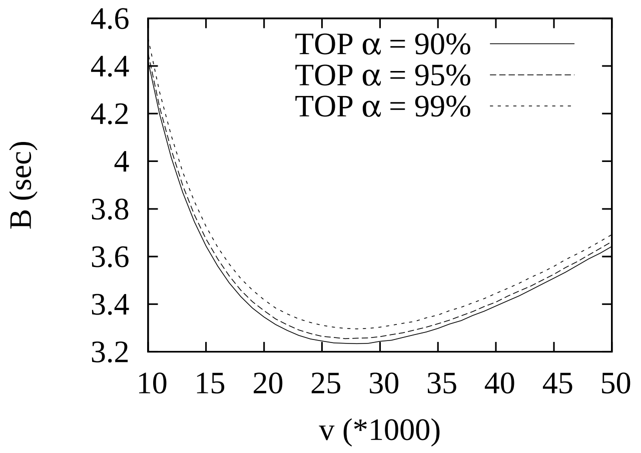

Figure 6-3. Bound B that satisfies $Prob\{T \leq B\} \geq \alpha$ with respect to v. Parameters: $n = 10,000$, $m = 1,000$.

monotonic function with respect to v. Hence, we cannot directly apply the bisection search method to find the largest v that satisfies $P_t(v, B) \geq \alpha$. We may use the False Position algorithm [16] to find the optimal value of v. The computation overhead is reasonable. For $n = 10,000$, $m = 1,000$, $B = 4$ seconds, and $\alpha = 99\%$, it takes an Apple macbook (2.4GHz CPU and 4GB memory) 3 seconds to find the optimal $v = 60,160$. And for $n = 10,000$, $m = 1,000$, $B = 3$ seconds, and $\alpha = 99\%$, it takes the same computer 16 seconds to find that no v can satisfy the requirement, because $B = 3$ seconds is smaller than the minimum execution time that TOP can achieve.

As a related problem, if v and α are given, we can also use $P_t(v, B)$ to compute the time bound that TOP can achieve. More specifically, given a value of v, we are able to find the smallest B that satisfies $P_t(v, B) \geq \alpha$ through bi-section search: Recall that $P_t(v, B)$ is the formula for $Prob\{T \leq B\}$, the probability for the protocol execution time to be bounded by B. Clearly, it is an increasing function of B with $P_t(v, 0) = 0$ and $P_t(v, +\infty) = 1$. We choose a small value B_1 (e.g., 0) such that $P_t(v, B_1) < \alpha$ and a large value B_2 such that $P_t(v, B_2) \geq \alpha$. Let $B_3 = \lceil \frac{B_1+B_2}{2} \rceil$. If $P_t(v, B_3) < \alpha$, assign B_3 to B_1; otherwise, assign B_3 to B_3. Hence, the search range $[B_1, B_2]$ is cut by half. Repeat the above process until $B_1 = B_2$, which gives the smallest bound B that satisfies

$P_t(v, B) \geq \alpha$. Let $n = 10,000$ and $m = 1,000$. Figure 6-3 shows the smallest bound B with respect to v when $\alpha = 90\%, 95\%$ and 99%, which correspond to the three curves in the figure.

6.6.4 Computing $p_d(m, v, x)$ — the Balls And Bins Algorithm

Problem: Suppose we throw m balls into v empty bins. Each ball is thrown to a random bin, and each bin can hold unlimited number of balls. We want to find the probability that after m balls are thrown, x bins are not empty, denoted as $p_d(m, v, x)$.

Solution: There are many solutions to this problem. We now provide a recursive one. Assume after we throw m balls, there are x non-empty bins, $1 \leq x \leq m$. When $x > 1$, there are two possibilities of where the m^{th} ball goes: (1) If the m^{th} ball is placed to a previously empty bin, there should be $x - 1$ non-empty bins after $m - 1$ balls were thrown, and the possibility for this to happen is $\frac{v-x+1}{v}$; (2) Otherwise if the m^{th} ball goes to a previously non-empty bin, there must be x non-empty bins after $m - 1$ balls were thrown, and the possibility of this option is $\frac{x}{v}$. Thus,

$$p_d(m, v, x) = \begin{cases} 1; x = m = 1. \\ \frac{x}{v} p_d(m-1, v, x) + \frac{v-x+1}{v} p_d(m-1, v, x-1); \\ \qquad 1 \leq x \leq m \text{ and } x \leq v. \\ 0; \text{ all other cases.} \end{cases}$$

$p_d(m, v, x)$ can be calculated from simple dynamic programing.

6.7 Enhanced Tag-Ordering Polling Protocol (ETOP)

6.7.1 Motivation

If we do not want to significantly increase execution time, we cannot choose a large value for v. In this case, we must find other means to lower energy cost. The key is to reduce the number of IDs that have to be transmitted in the polling phase. Namely, we should reduce the number of tags in F and C. Let us first focus our discussion on false positive. Consider an arbitrary tag $t \notin M$. Its representative segment is V_t. Let q be the number of tags in M that are also mapped to V_t. False positive occurs if t and one of those q tags have the same representative bit. The probability for this to happen is $1-(1-\frac{1}{l})^q$, where l is the number of bits in V_t.

To further reduce the false-positive probability, we can implement each segment of V as a Bloom filter [7, 12]. The reader uses multiple hash functions to map each tag to $k(>1)$ representative bits in V, instead of just one in TOP. More specifically, for each member $t' \in M$, the reader first maps it to a representative segment $V_{t'}$ through a hash function whose range is $[0, \frac{v}{l})$. Then the reader further maps t' to k representative bits in $V_{t'}$ and set them to ones.

After all members in M are encoded in the segments of V, the reader broadcasts the segments in the ordering phase. A tag t only listens for its representative segment V_t and then checks its representative bits. If any representative bit is zero, the tag can not be in M. If all representative bits are ones, the tag may be a member in M or a false positive. In the case of false positive, even though the tag does not belong to M, every one of its representative bits is set because it is also a representative bit of a member tag in M. The probability for this to happen is $(1-(1-\frac{1}{l})^{kq})^k$, where q is the number of tags in M whose representative segments are also V_t. For example, if $l=80$, $k=3$, and $q=2$, the false-positive probability is just 3.8×10^{-4}, much lower than $1-(1-\frac{1}{l})^q = 2.5\times10^{-2}$ in TOP under the same parameters.

Bloom filters can reduce the false-positive probability. But it is more difficult to use them to carry the reporting order, based on which the tags will take turn to transmit during the reporting phase. In TOP, we use the number of ones that precede the representative bit of a tag to determine the tag's position in the reporting order. Bloom filters use multiple representative bits to encode each member. The representative bits of different members may overlap in an arbitrary way. Hence, we cannot simply use all bits whose values are ones to represent tags in M because there is no one-to-one mapping between them.

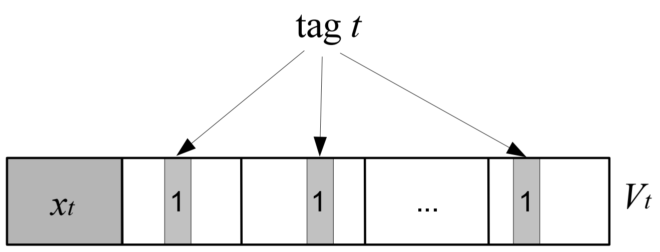

Figure 6-4. V_t is the representative segment of tag t. V_t is evenly divided into k partitions, each having $\lfloor \frac{l}{k} \rfloor$ bits. Tag t has one representative bit in every partition.

In the following, we design an enhanced tag-ordering polling protocol (ETOP) to solve the above problem. ETOP uses partitioned Bloom filters, which not only reduce false positive and encode the reporting order, but also reduce $|C|$ as well as overall execution time of the protocol.

6.7.2 Protocol Description

The main difference between ETOP and TOP is that ETOP implements each segment of V as a partitioned Bloom filter instead of a simple bit array. When we describe the protocol of ETOP, we focuses on the difference while omitting the details that it shares in common with TOP.

In a partitioned Bloom filter, the l bits of a segment are evenly divided into k partitions. Each partition has $\lfloor \frac{l}{k} \rfloor$ bits. See Figure 6-4 for illustration. For every member tag t in M, the reader applies a hash function on its ID to obtain a number of hash bits. The reader uses $\lceil \log_2 v \rceil$ hash bits to map t to a representative segment V_t, and then uses $k \lceil \log_2 \frac{l}{k} \rceil$ hash bits to further map t to one representative bit in every partition of the segment. Like a classical Bloom filter, the partitioned Bloom filter sets k representative bits for each encoded member; unlike a classical Bloom filter, a partitioned Bloom filter spreads the k representative bits in k different partitions.

After receiving its representative segment, a tag checks the k representative bits to determine if it is a member in M. False positive cases are handled by the reader in the polling phase as usual.

How does a tag t know its position in the reporting order? First we consider the reporting order among tags that are encoded in the same segment V_t. Since every tag has exactly one representative bit in each partition of V_t, we may be able to use one of the partitions to carry the order information. In other words, if there is a partition P^* whose number of ones is equal to the number of tags encoded in V_t, we know that there must be a one-to-one mapping between these tags and the '1' bits in P^*. We can use the order of '1' bits in P^* as the reporting order of the corresponding tags. We will explain later how the reader makes sure that such a partition exists. When the reader sends out V_t, in the same time slot it also sends the total number x_t of tags that are encoded in all previous segments of V. The position of tag t in the reporting order can be computed from x_t and the information in P^*, which we will further explain shortly.

How to make sure that any segment of V always has a partition whose number of ones is equal to the number of tags encoded in the segment? The reader has to do some extra work. After encoding all tags in M, the reader examines the partitions one by one for each segment. If there is not such a partition, the reader removes an encoded tag and places it in the set C, which will be explicitly polled in the polling phase. The reader keeps removing tags until it finds a partition that satisfies the above requirement. Note that the requirement is always satisfied when the number of tags encoded in a segment is one.

After receiving its representative segment V_t, a tag $t \in M$ computes its position in the reporting order as follows: It finds out a partition P^* in V_t that has the largest number of ones. This partition must have a one-to-one mapping between '1' bits and encoded tags. Let y_t be the number of ones in P^* that precedes the representative bit of t. The tag computes its position in the reporting order as $y_t + x_t$. Recall that x_t is the number of tags that are encoded in the previous segments. It is received together with V_t in the same time slot.

The polling phase and the reporting phase of ETOP are identical to their counterparts in TOP.

6.8 Performance Analysis of ETOP

6.8.1 Energy Cost

We show that ETOP can be configured such that the energy cost per tag is $O(1)$. ETOP has the same upper bound formula for per-tag energy cost as TOP does, which is shown in (6–1), but it has different values of $|C|$ and $|F|$. In the following, we derive $|C|$ and $|F|$ for ETOP. Let m_i be the number of tags in M that are encoded in the ith segment, $0 \leq i < \frac{v}{l}$. Each tag in M has a probability of $\frac{l}{v}$ to be mapped to the ith segment. Hence, m_i follows a binomial distribution $Bino(m, \frac{l}{v})$.

$$Prob\{m_i = x\} = \binom{m}{x}(\frac{l}{v})^x(1-\frac{l}{v})^{m-x}. \tag{6–13}$$

Let C_i be a subset of C, containing the tags that are removed from the ith segment. We know the following facts: (1) When $m_i = 0$, $|C_i| = 0$. (2) When $m_i = 1$, $|C_i| = 0$. (3) When $m_i \geq 1$, $|C_i| \leq m_i - 1$. Hence, we must have

$$\begin{aligned} E(|C_i|) &< (m_i - 1)\cdot\Big(1 - Prob\{m_i = 0\} - Prob\{m_i = 1\}\Big) \\ &= (m_i - 1)\cdot(1 - (1-\frac{l}{v})^m - \frac{ml}{v}(1-\frac{l}{v})^{m-1}). \end{aligned}$$

Since $(1-\frac{l}{v})^m > 1 - \frac{ml}{v}$, we have

$$E(|C_i|) < \frac{m_i(m-1)^2l^2}{v^2} < \frac{m_i m^2 l^2}{v^2}.$$

$|C|$ is the sum of all $|C_i|$s, $0 \leq i < \frac{v}{l}$. We know $\sum_{i=1}^{v/l} m_i = m$. So,

$$E(|C|) = \sum_{i=1}^{v/l} E(|C_i|) < m\frac{m^2l^2}{v^2} = \frac{m^3l^2}{v^2}.$$

If we let $v = \sqrt{m^3l^2}$, $E(|C|) < 1$.

Consider an arbitrary tag not in M. Without loss of generality, suppose it is mapped to the ith segment. In any partition of the segment, the probability for it to share a representative bit with a tag in M is $1-(1-\frac{k}{l})^{m_i}$. The probability for that to happen in all partitions is $[1-(1-\frac{k}{l})^{m_i}]^k$. Hence, the probability for the tag to cause false positive, denoted as p_f is

$$\begin{aligned} p_f &= \sum_{q=0}^{m} Prob\{m_i = q\}\big[1-(1-\frac{k}{l})^q\big]^k \\ &< (1-Prob\{m_i=0\})\big[1-(1-\frac{k}{l})^m\big]^k \\ &\approx (1-e^{-lm/v})(1-e^{-km/l}). \end{aligned}$$

The expected valus of $|F|$ is

$$E(|F|) = (n-m)\cdot p_f < (n-m)(1-e^{-lm/v})(1-e^{-km/l}). \tag{6–14}$$

If we let $v = -\frac{ml}{\ln(1-\frac{1}{(n-m)(1-e^{-km/l})})}$ and apply it to (6–14), we have $E(|F|) < 1$. Now, if we choose $v = \max\{\sqrt{m^3l^2}, -\frac{ml}{\ln(1-\frac{1}{(n-m)(1-e^{-km/l})})}\}$, the expected energy cost $E(\Omega) \leq E(|C|)+E(|F|)+O(1) < 1+1+O(1) = O(1)$. Therefore, ETOP can also be configured such that the energy cost per tag is $O(1)$.

6.8.2 Execution Time

Following the same analysis as in Section 6.6.2, it is easy to see that ETOP has the same formula for protocol execution time as TOP does: $T = (\frac{v}{l}+|F|+|C|)t_{tag}+m\times t_{inf}+O(1)$, but the values of $|C|$ and $|F|$ are different. Our simulation results in Section 6.10 show that ETOP has smaller execution time than TOP.

6.8.3 Choosing v for Time-constrained Energy Minimization

Following the same reasoning in Section 6.6.3, we define the time bound for ETOP to be

$$Prob\{T \leq B\} \geq \alpha, \tag{6–15}$$

where T is the execution time of ETOP, B is a pre-defined time bound, and α is a probability value, $0 < \alpha < 1$. The objective is to find the largest value v that minimizes the energy cost, subject to the constraint (6–15). In the following, we derive a computable formula for $Prob\{T \leq B\}$, which can be found in (6–23) and (6–24). Based on the formula, we will be able to find the optimal value v.

Let m_i be the number of tags in M that are encoded in the ith segment, denoted as V_i, $0 \leq i < \frac{v}{l}$. Obviously, m_i follows a binomial distribution $Bino(m, \frac{l}{v})$,

$$Prob\{m_i = x\} = \binom{m}{x}(\frac{l}{v})^x(1 - \frac{l}{v})^{m-x}.$$

Let n_i be the number of tags not in M that are mapped to the ith segment, $0 \leq n_i \leq n - m$. Obviously, n_i follows the binomial distribution $Bino(n - m, \frac{l}{v})$.

$$Prob\{n_i = z\} = \binom{n-m}{z}(\frac{l}{v})^z(1 - \frac{l}{v})^{n-m-z}.$$

Let C_i be a subset of C, containing the tags that are removed from V_i; Let F_i be a subset of F, consisting the false positive tags that are mapped to V_i; Let S_i be the union of C_i and F_i, thus $|S_i| = |C_i| + |F_i|$, and,

$$\begin{aligned} &Prob\{|S_i| = s\big|m_i = x, n_i = z\} \\ &\quad = \sum_{c=0}^{s} Prob\{|F_i| = s - c\Big||C_i| = c, m_i = x, n_i = z\} \cdot Prob\{|C_i| = c\big|m_i = x\}. \end{aligned} \quad (6\text{–}16)$$

Firstly, we show how to calculate $Prob\{|C_i| = c\big|m_i = x\}$. After encoding m_i tags in V_i, let d_{ij} be the number of ones in the jth partition, $1 \leq j \leq k$. As a tag in M has exactly 1 representative bit in each partition, $0 \leq d_{ij} \leq \min\{m_i, \frac{l}{k}\}$. The reader removes a tag to C_i only if it shares a representative bit with another tag in the partition that contains the largest number of ones. As a result, $|C_i| = m_i - \max_{j\in[1,k]} d_{ij}$. When $y \geq 1$, we have

$$
\begin{aligned}
&Prob\{\max_{j\in[1,k]} d_{ij} = y \big| m_i = x\} \\
&\quad = \prod_{j=1}^{k} Prob\{d_{ij} \le y \big| m_i = x\} - \prod_{j=1}^{k} Prob\{d_{ij} \le y-1 \big| m_i = x\} \\
&\quad = \Big(\sum_{d=0}^{y} Prob\{d_{ij} = d \big| m_i = x\}\Big)^k - \Big(\sum_{d=0}^{y-1} Prob\{d_{ij} = d \big| m_i = x\}\Big)^k \\
&\quad = \Big(\sum_{d=0}^{y} p_d(x, \frac{l}{k}, d)\Big)^k - \Big(\sum_{d=0}^{y-1} p_d(x, \frac{l}{k}, d)\Big)^k, \qquad (6\text{–}17)
\end{aligned}
$$

where $p_d(x, \frac{l}{k}, d) = Prob\{d_{ij} = d \big| m_i = x\}$ is the conditional probability that a partition containing $m_i = x$ tags happens to have d ones. The calculation of $p_d(\cdot)$ can be found in Section 6.6.4. Hence, the conditional distribution of $|C_i|$ is,

$$
Prob\{|C_i| = c \big| m_i = x\} = \Big(\sum_{d=0}^{x-c} p_d(x, \frac{l}{k}, d)\Big)^k - \Big(\sum_{d=0}^{x-c-1} p_d(x, \frac{l}{k}, d)\Big)^k. \qquad (6\text{–}18)
$$

Secondly, we derive $Prob\{|F_i| = s-c \big| |C_i| = c, m_i = x, n_i = z\}$. A tag not in M maps itself to k partitions and choose one bit randomly from each partition. If all these bits are ones, false positive happens. The conditional false positive probability is,

$$
Prob\{\text{false positive in } V_i \big| m_i = x\} = \Big(\sum_{d=0}^{x} \frac{kd}{l} p_d(x, \frac{l}{k}, d)\Big)^k. \qquad (6\text{–}19)
$$

When $|C_i| = c$, $\max_{j\in[1,k]} d_{ij} = m_i - c$, hence,

$$
\begin{aligned}
&Prob\{\text{false positive in } V_i \Big| |C_i| = c, m_i = x\} \\
&= \frac{1}{p_d(x, \frac{l}{k}, x-c)} \Big[\Big(\sum_{d=0}^{x-c} \frac{kd}{l} p_d(x, \frac{l}{k}, d)\Big)^k - \Big(\sum_{d=0}^{x-c-1} \frac{kd}{l} p_d(x, \frac{l}{k}, d)\Big)^k\Big], \qquad (6\text{–}20)
\end{aligned}
$$

denoted as p_{fc}, which represents the false positive probability when $m_i = x$ tags are encoded in the ith segments and $|C_i| = c$ tags are moved to the collision set C. When n_i tags in $N - M$ are mapped to V_i, the conditional distribution of $|F_i|$ follows the binomial distribution $Bino(n_i, p_{fc})$, thus,

$$Prob\{|F_i| = s - c \big| |C_i| = c, m_i = x, n_i = z\} = \binom{z}{s-c} p_{fc}^{s-c}(1-p_{fc})^{z-s+c}. \tag{6–21}$$

From (6–18) and (6–21), we can derive $Prob\{|S_i| = s \big| m_i = x, n_i = z\}$. Thus, the probability distribution of $|S_i|$ is,

$$\begin{aligned}
Prob\{|S_i| = s\} &= \sum_{x=0}^{m}\sum_{z=0}^{n-m}\sum_{c=0}^{s} Prob\{|S_i| = s \big| m_i = x, n_i = z\} \cdot Prob\{m_i = x\} \cdot Prob\{n_i = z\} \\
&= \sum_{x=0}^{m}\sum_{z=0}^{n-m}\sum_{c=0}^{s} \Big[\big(\sum_{d=0}^{x-c} p_d(x, \frac{l}{k}, d)\big)^k - \big(\sum_{d=0}^{x-c-1} p_d(x, \frac{l}{k}, d)\big)^k\Big] \\
&\quad \cdot \binom{z}{s-c} p_{fc}^{s-c}(1-p_{fc})^{z-s+c} \cdot \binom{m}{x}(\frac{l}{v})^x(1-\frac{l}{v})^{m-x} \\
&\quad \cdot \binom{n-m}{z}(\frac{l}{v})^z(1-\frac{l}{v})^{n-m-z}.
\end{aligned} \tag{6–22}$$

Let S be the union of C and F. We have $|S| = |C| + |F|$ and $|S| = \sum_{i=1}^{v/l} |S_i|$. As S_1, S_2, ..., $S_{v/l}$ are independent of each other, the probability distribution of $|S|$ is the convolution of $|S_i|$. Hence,

$$Prob\{|S| = s\} = Prob\{|S_1| = s\} * ... * Prob\{|S_{\frac{v}{l}}| = s\},$$

where $*$ is the convolution operator. With the help of Fourier Transform, we have

$$Prob\{|S| = s\} = F\hat{F}T\Big[\Big(FFT\big(Prob\{|S_i| = s\}\big)\Big)^{v/l}\Big], \tag{6–23}$$

where FFT is the Fast Fourier Transform, and $F\hat{F}T$ is the inverse Fast Fourier Transform. Adopting (6–7), we have

$$Prob\{T \leq B\} = \sum_{s=0}^{s_{max}} Prob\{|S| = s\}, \tag{6–24}$$

where $s_{max} = \frac{B - mt_{inf}}{t_{tag}} - \frac{v}{l}$. The right side is denoted as $P'_t(v, B)$, which is the probability for the protocol execution time to be bounded by B under a certain value of v. It is computable as a function of v and B after (6–22) is applied and parameters m and n are

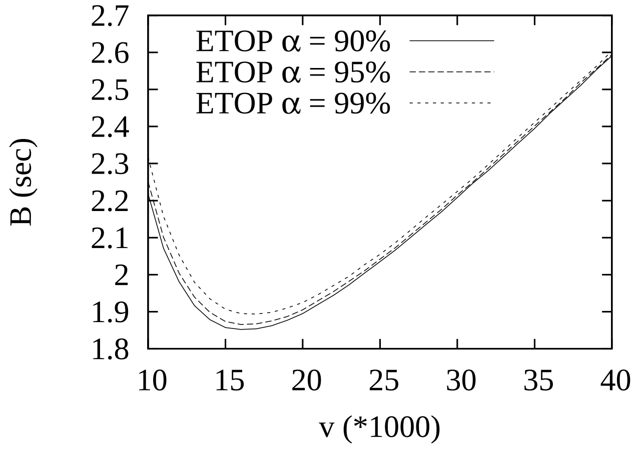

Figure 6-5. Bound B that satisfies $Prob\{T \leq B\} \geq \alpha$ with respect to v. Parameters: $n = 10,000$, $m = 1,000$.

given. Given a value of B, we can find the largest v that satisfies $P'_t(v, B) < \alpha$ using the False Position algorithm [16]. For example, when $n = 10,000$, $m = 1,000$, $B = 2$ seconds, and $\alpha = 99\%$, the optimal value of v is 23,200.

As a related problem, if v and α are given, we can use $P'_t(v, B)$ to compute the time bound that ETOP can achieve. More specifically, given a value of v, we are able to find the smallest B that satisfies $P'_t(v, B) \geq \alpha$ through bi-section search as described in Section 6.6.3. Figure 6-5 shows the time bound of ETOP with respect to v when $\alpha = 90\%, 95\%$ and 99%, which correspond to the three curves in the figure.

6.9 Channel Error

Channel error may corrupt the data exchanged between the reader and tags. For example, if a negative polling request is corrupted, the tag that is not supposed to participate in the reporting phase will transmit and cause collision in the reporting phase. A segment of V sent from the reader may be corrupted so that tags encoded in this segment will not report their information. There exists other scenarios of corruption in the execution of TOP or ETOP. They cause two effects: 1) A tag in M does not transmit its information in the slot when it is supposed to transmit, and 2) it transmits but collides with another tag that is not supposed to transmit in the slot. To detect these cases, when a tag transmits, we require it to include a CRC checksum that is computed from the concatenation of the information bits and the tag's ID. When the reader expects information from a tag in a time slot, if the slot turns out to be empty or the data received in the slot do not carry a correct CRC, the reader knows that information from the tag is not correctly received. At the end of the protocol, all missed information can be retrieved by polling the tags directly.

6.10 Simulation Results

In this section, we evaluate the performance of our new protocols, the tag ordering polling protocol (TOP) and the enhanced tag ordering polling protocol (ETOP). We compare them with the basic polling protocol (BP) and the coded polling protocol (CP). Our evaluation uses two performance metrics: (1) the average number of bits that each tag has to receive during the protocol execution, and (2) the overall execution time.

We only consider energy consumption of tags in receiving information for two reasons. First, this is the major, variable portion of the energy cost per tag. As we will see shortly, each tag may have to receive hundreds of thousands of bits during protocol execution, whereas it only sends a small, fixed amount, e.g., 32 bits. Second, the energy cost for tags in M to transmit their information is the same for all protocols. Omitting them does not affect the comparison.

We use the following parameters to configure the simulation: each tag ID is 96 bits long, information reported from a tag to the reader is 32 bits long, and each segment in ETOP is 80 bits long and divided into 4 partitions, i.e. $k = 4$. The transmission time is based on the parameters of the Philips I-Code specification [138]. The rate from a tag to the reader is 53Kb/sec; it takes 18.88μs for a tag to transmit one bit. Any two consecutive transmissions (from the reader to tags or vice versa) are separated by a waiting time of 302μs. The value of t_{inf} is calculated as the sum of a waiting time and the time for transmitting the information, which is 18.88μs multiplied by the length of the information. For 32-bit information, $t_{inf} = 906\mu s$. The transmission rate from the reader

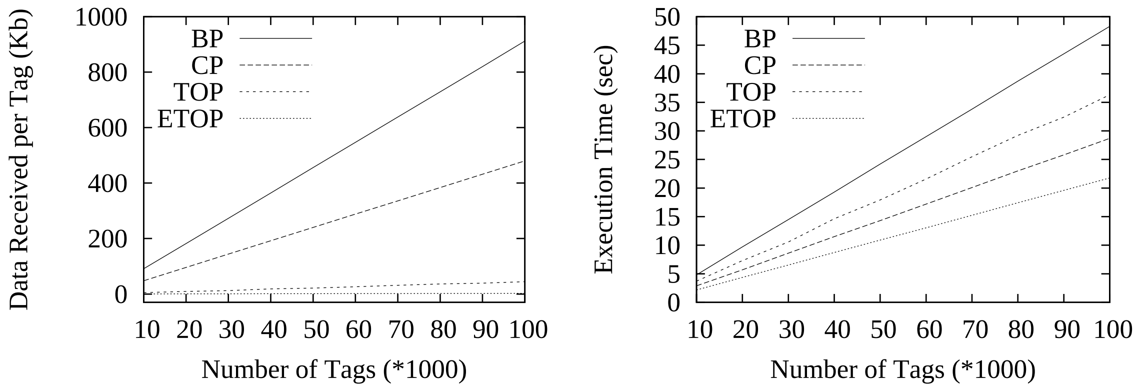

Figure 6-6. Energy and time comparison between BP, CP, TOP, and ETOP. Parameters: $m = 0.1n$, $v = 24m$ for TOP and ETOP. Note that the horizontal '0' line is not at the bottom in order to make the ETOP curve visible.

to tags is 26.5Kb/sec; it takes 37.76μs for the reader to transmit one bit. The value of t_{tag} is calculated as the sum of a waiting time and the time for transmitting a 96-bit ID. The result is 3927μs.

6.10.1 Varying number n of tags

We first vary the number n of tags in the system from 10,000 to 100,000. We set $v = 24m$ and $m = 0.1n$, i.e., 10% of all tags are selected by the reader to report information. Figure 6-6 compares four protocols in terms of energy cost and protocol execution time. The left plot shows energy costs. TOP and ETOP reduce energy consumption by one or multiple orders of magnitude. For example, when $n = 100,000$, per-tag energy cost in TOP is 9.4% of the cost in CP, and 5.0% of the cost in BP. Per-tag energy cost in ETOP is just 0.52% of the cost in CP, and 0.28% of the cost in BP. The right plot shows the execution time comparison. TOP requires 25% less time than BP, but 27% more time than CP. ETOP requires 55% less time than BP and 24% less time than CP.

In summary, CP reduces both energy cost and execution time nearly by half when comparing with BP. TOP makes great improvement over CP in terms of energy cost, but has modestly higher execution time. ETOP considerably outperforms CP in terms of both energy cost and execution time.

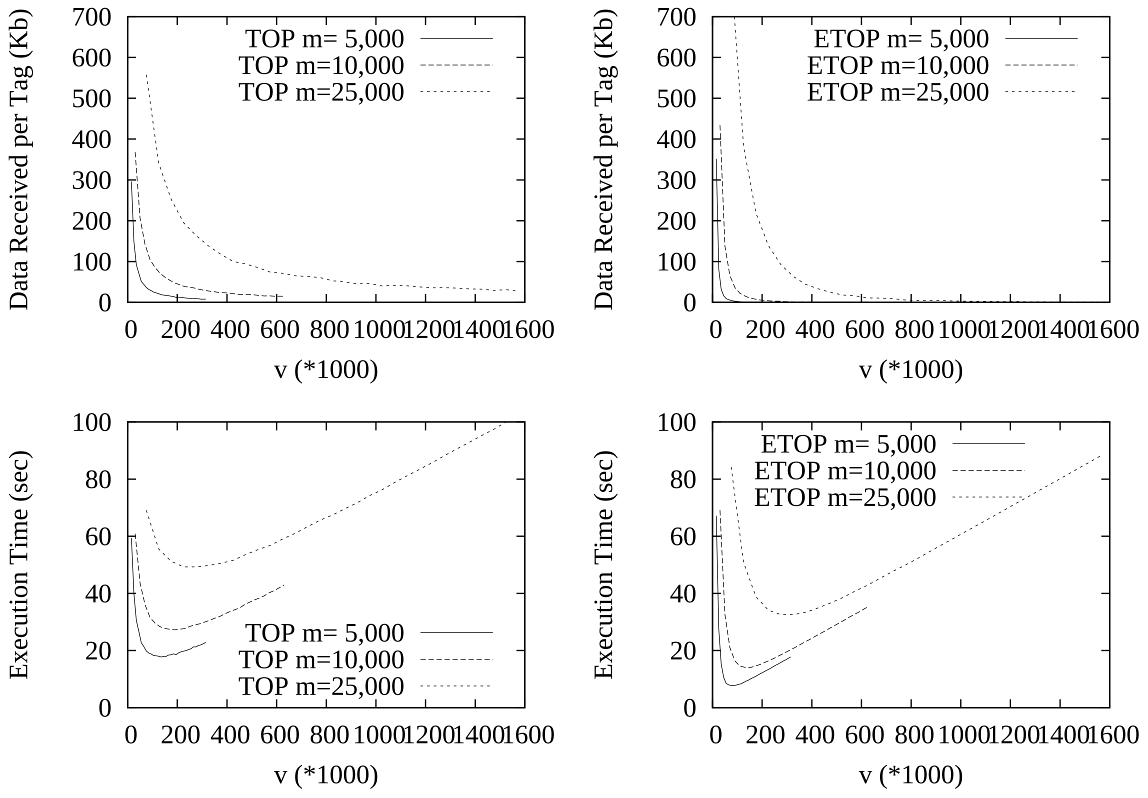

Figure 6-7. Energy and execution time comparision of TOP and ETOP. Top left plot: Energy cost of TOP with respect to v. Top right plot: Energy cost of ETOP with respect to v. Bottom left plot: Execution time of TOP with respect to v. Bottom right plot: Execution time of ETOP with respect to v.

6.10.2 Varying Size v of Reporting-order Vector

Next, we show how the value of v influences the performance of TOP and ETOP. We set $n = 50,000$ and $m = 5,000$, $10,000$, or $25,000$. We vary v from $4m$ to $64m$ and use simulation to find energy cost per tag and protocol execution time. Figure 6-7 shows the simulation results. The top left and top right plots present the average amount of data each tag receives in TOP and ETOP, respectively. The curves match the theoretical results we have given in Section 6.6. When v is reasonably large, e.g., $v \geq 7m$, ETOP consumes less energy than TOP. The bottom left and bottom right plots present the protocol execution time of TOP and ETOP, respectively. ETOP also requires less time than TOP when $v \geq 7m$.

6.11 Summary

In this chapter, we introduce two energy-efficient information collection protocols, TOP and ETOP, for large-scale RFID systems. These protocols are designed to collect real-time information from a subset of tags in the system. Our primary objective is to lower energy consumption by tags in order to extend their lifetime. TOP shares similarity with the Bloom-1 filter we discussed in Chapter 3, while ETOP can be treated as an extension of it adopting the idea of partitioned Bloom filter. The new protocols can be configured to achieve $O(1)$ energy cost per tag. Performance tradeoff between energy cost and execution time can be made by controlling the size of the reporting-order vector. Simulation results show that the new protocols are able to cut energy cost by more than an order of magnitude, when comparing with other protocols.

CHAPTER 7
SET SIZE ESTIMATION IN MOBILE VEHICULAR P2P NETWORKS

7.1 Background, System Model, and Problem Definition

7.1.1 Mobile Vehicular P2P Networks

RFID system is not the only cyber-physical system that requires time and energy efficiency. Future automobiles may be equipped with wireless devices that enable vehicular peer-to-peer (VP2P) networks to make our transportation systems more efficient. As vehicles move close or pass by each other, data may be exchanged to inform road conditions ahead, and statistical information or system-wide measurement may be generated collaboratively [151]. In a VP2P network, moving cars communicate using short-range wireless technologies such as IEEE 802.11 and Dedicated Short Range Communications (DSRC). Vehicles in these networks can be highly mobile. Constantly unstable network topologies may make it unsuitable to use some popular routing protocols such as DSDV [125], AODV [126] and DSR [64], which are designed for traditional mobile ad-hoc networks (whose topologies should be relatively stable enough for these protocols to work). If the vehicles are GPS-capable, geographical routing protocols [9, 67] can be used to route data over multiple hops.

New functions have to be developed in the context of VP2P to support emerging applications. This paper investigates one basic function: estimating the cardinality of a mobile VP2P network, i.e., the number of vehicles in the network. In literature, this problem is also called “the vehicle density estimation problem”. Such a function is useful when a transportation department needs to know information about traffic volume, i.e., number of moving cars,[1] in each district of a city, where cars within the boundary of each district forms a VP2P network with the assistance of GPS and

[1] Parked cars do not contribute to traffic. They are mostly powered off and therefore do not participate in the network.

wireless communication devices. Each vehicle in a VP2P network only has knowledge about its immediate neighborhood. Without a central entity that has access to all vehicles, cardinality estimation is a distributed process carried out among peer nodes, as vehicles enter and depart from the network dynamically.

The function of cardinality estimation can be easily implemented in a static wireless network where all nodes are stationary. A query node broadcasts a request message into the network. Each node transmits the message once when it receives the message for the first time. The node also remembers its predecessor, which is the one from which it receives the message for the first time. This simple broadcast protocol is not an efficient one, comparing with others [60, 91, 105]. But it quickly establishes a routing tree if each node treats its predecessor as the parent node. In this tree, each node learns the numbers of nodes in the subtrees rooted at its children, sum them up for the number of nodes in the subtree rooted at itself, and then reports that information to its parent. As this distributed computation is recursively carried out from the leaf nodes toward the root, the query node — which is located at the root of the tree — will learn the number of nodes in the whole network. However, any node failure will break the tree. To solve this problem, the synopsis diffusion approach [117] is proposed to collect information based on a more robust DAG(Directed Acyclic Graph [32]) routing structure. Each node aggregates data from its upstream neighbors, integrates its own information, and broadcasts the aggregated information downstream. To ensure each node only transmits once, it must receive data from all upstream neighbors before its own transmission. This requires a static topology such as the type of sensor networks investigated in [117]. Other more efficient approaches have been invented to estimate the number of nodes in a stationary sensor network [15, 33, 34, 81] or estimate the number of tags in a large RFID system [55, 71, 72, 129, 173], assuming a centralized communication model where all tags directly communicate with a RFID reader.[2] The above solutions cannot be applied to the multi-hop mobile network model in the context of chapter.

In a mobile VP2P network, because nodes (i.e., vehicles) are moving, there is no stable routing structure for collecting information. Moreover, the number of nodes may change as new nodes join and existing nodes depart. This requires the operation of determining the number of nodes to be carried out frequently if applications demand an evolving picture of network size over time. On one hand, it is not efficient to rely on a flooding (or broadcasting) approach for such an operation due to high overhead. On the other hand, many applications may not require the exact count of the nodes, which varies over time anyway. An estimated number can already be very useful if a certain accuracy requirement is met. For example, the transportation department may not have to know the exact number of vehicles that are present in a district. If an estimated number, say within $\pm 10\%$ of the exact number, meets their need, then we can adopt more efficient approaches that work well for mobile networks.

This chapter proposes two statistical methods for estimating the cardinality of a mobile VP2P network. The first method is called the circled random walk (CRW). It estimates the number of nodes with an accuracy that is tunable. It makes tradeoff between communication overhead and estimation error. The error can be made arbitrarily small at the expense of increasing overhead. One problem of CRW is that it has an idealized assumption for randomized neighboring relationship among the nodes, which approximates the case of a relatively small network of highly mobile nodes. This assumption is however not generally applicable. Our second method, called the tokened random walk (TRW), removes such an assumption. With the assistance

[2] Normally, RFID tags cannot communicate with each other to form a multi-hop wireless network.

of randomly distributed tokens, TRW provides a statistical estimation on the number of nodes in an arbitrarily connected VP2P network. Our extensive simulations demonstrate that CRW works well in networks of high mobility, whereas TRW works well with high or low mobility.

There are other types of mobile P2P networks [157, 160], where communication infrastructure does not exist or cannot be accessed due to technical or economical reasons. Mobile P2P networks have many civilian and military applications. For instance, soldiers in a battlefield may carry small wireless devices and form a mobile P2P network among them. In another example, smart phones or PDAs may carry applications that benefit from peer-to-peer networks for information discovery or social networking functions. Solutions developed for VP2P networks, including the methods for cardinality estimation in this chapter, will be able to find their applicability in other types of mobile P2P networks as well.

7.1.2 Network Model

Consider a large vehicular peer-to-peer system where we abstract vehicles as wireless nodes. Wireless links are established between nearby nodes that are within the communication range of each other. All links form a connected graph. Two nodes are neighbors of each other if there is a wireless link between them. Because the nodes move, the neighboring relationship changes over time.

7.1.3 Problem Definition

The problem is to estimate the number n of nodes in a mobile VP2P network, i.e., the cardinality of the network. Our goal is to develop statistical methods that estimate the value of n without flooding the network or requiring the participation of all nodes (for overhead reduction). Let $\hat{n}$ be the estimated number of nodes. The goal is that the probability for n to fall in the range $[(1-\beta)\hat{n}, (1+\beta)\hat{n}]$ is at least α, where $\alpha(<1)$ and $\beta(<1)$ are the parameters of the accuracy requirement.

We assume that there exist end-to-end routing protocols [9, 67] that are able to route a message to a certain node.

Consider a large vehicular peer-to-peer system where we abstract vehicles as wireless nodes. Wireless links are established between nearby nodes that are within the communication range of each other. All links form a connected graph. Two nodes are neighbors of each other if there is a wireless link between them. Because the nodes move, the neighboring relationship changes over time.

The problem is to estimate the number n of nodes in a mobile VP2P network, i.e., the cardinality of the network. Our goal is to develop statistical methods that estimate the value of n without flooding the network or requiring the participation of all nodes (for overhead reduction). Let $\hat{n}$ be the estimated number of nodes. By sampling only a subset of nodes, our methods ensure that the probability for n to fall in the range $[(1-\beta)\hat{n}, (1+\beta)\hat{n}]$ is at least α, where $\alpha(<1)$ and $\beta(<1)$ are the parameters of the accuracy requirement.

We assume that there exist end-to-end routing protocols [9, 67] that are able to route a message to a certain node.

For mobile VP2P networks, we may approximately consider that the neighbors of a node are randomly selected from the network if the movement of a node allows it to frequently change neighbors and have chance of neighboring with many other nodes over time. In practice, the random-neighbor assumption cannot really be met. However, our simulation reveals that even though this assumption is not accurate, our method based on the assumption still provides good cardinality estimation. We also observe that for static networks or networks of low mobility, the method based on the random-neighbor assumption does not work well.

We develop cardinality estimation methods for mobile VP2P networks. Our first method uses the random neighbor assumption. It is simpler and easy to implement. Removing the random neighbor assumption, our second method is able to work for network of high or low mobility. It is more robust but requires additional assistance in its execution.

7.1.4 Prior Art

The most related topic in vehicular networks is the vehicle density estimation problem, which aims to calculate the number of vehicles per road length in an area. Our estimated cardinality can be directly applied to calculate vehicle density by dividing it with the total length of the road in the area. Most related work estimate the traffic density on specific road or section. Yet our approach focuses on a relative large district, for example, the whole downtown area, whose actually density changes much slower over time comparing to street segments.

Vehicle density estimation methods can be divided into two general categories: infrastructure-assisted approaches and infrastructure-free approaches.

Infrastructure-assisted approaches [4, 36, 51] rely on detecting devices such as inductive loop detectors [146], roadside radar, sensors, or cameras to be installed for collecting traffic data. These devices may collect the vehicle count, the vehicle speed, or the instance at which each vehicle moves past them, which are then used to estimate the vehicle density. However, these approaches suffer from limited coverage, low reliability, and high deployment and maintenance costs [80].

Infrastructure-free density estimation approaches do not rely on any pre-installed detecting devices. Existing works in this category can be further divided into two sub-categories: speed-based approaches and message-based approaches.

Speed-based approaches [5, 141, 153] assume a speed-density relationship between vehicle's speed and vehicle density on the road. For example, low speed implies dense traffic. However, these approaches could give inaccurate densities since the speed is not always an indication of the vehicles density on the road. An example is when vehicles stop at sparse intersections with low speed and low density.

The following works, together with our approach, belong to message-based category. Jerbi et al. proposed a fully distributed grouping approach for density estimation [63]. In this approach, a road segment is divided into multiple fixed-size cells. Group leader, which is a designated vehicle in a cell, computes vehicle density and spreads this information among other members in the cell and group leaders of other cells. The average density on the road can be obtained.

Panichpapiboon and Pattara-atikom proposed a neighbor-based density estimation [123]. Local density is first calculated based on the number of one-hop or two-hop local neighbors. Then, the global density of the road is estimated assuming the inter-vehicle spacing is exponentially distributed. However, in sparse network, this approach become unreliable because the probe vehicle is not able to find a sufficient number of neighbors to make an accurate estimation. Also, their assumption of exponential distributed inter-vehicle spacing might not be the case for all possible traffic scenarios.

Garelli et al. proposed a method called MobSampling to calculate the density of a specific target area [50]. It selects a vehicle called "sampler", which broadcast a "POLL" message with the sampler position and the radius of the target area at random intervals. When a vehicle in the target area receives the message, it replies after waiting for random time to avoid flooding the receiver. The sampler counts the received messages to estimate the local density. When a sampler leaves the target area, a new vehicle is chosen as sampler. As sampler and sampler position randomly changes, continuous measurements of local density gives an estimation of density of the whole area. However, this method needs relatively long time due to the random waiting and sampler handing-over.

In [2], Akhtar et al. adapted mechanisms from system size estimation in peer-to-peer system [62, 73, 108] for vehicular density estimation. They adapted and implemented three fully distributed algorithms, namely Sample & Collide, Hop Sampling and Gossip-based Aggregation. Sample & Collide algorithm tries to sample the nodes from the network, and then estimating the system size depending on how many samples of the nodes are collected, before an already sampled node is re-selected. This algorithm is similar to our CRW approach in terms variable used for estimation, but random sampling is achieve differently. In their algorithm, an initiator node sends a message with a pre-defined value T to one of its neighboring nodes ($T > 0$). Upon receiving a sampling message, a node decrements T by a random amount, and send the new value to one of its neighbors selected uniformly at random. When a node finds $T \leq 0$, it is sampled and replies to the initiator. The initiator keeps sending sampling message until a already sampled node replies. This algorithm has three disadvantages: First, it does not take advantage of node mobility, which itself shows randomness in neighboring relationship. Second, even though it sends out several probe messages, it essentially has only one measurement of C (equivalent to X in our approach). It is not efficient. Third, precision is not bounded in their algorithm.

The Hop Sampling algorithm works as followsThe initiator node broadcasts a message to all the vehicles in the network using gossiping. The vehicles reply back to the initiator with a probability, which is inversely proportional to the distance (hop count) from the initiator. Based on the replies that the initiator gets from other vehicles at different distances, it estimates the size of the network. However, the overhead of this approach is too much. Even though nodes reply probabilistically, gossiping message traverses every edge in the network at least once. Moreover, as network topology dynamically changes, a vehicle's probability of replying is larger than expected, because its stored hop count never goes up even if its real distance has increased. As a consequence, the algorithm has over-estimation problem, which is indicated by their simulation results.

In Gossip-based Aggregation algorithm, the initiator samples K vehicles at random and assign a value 1 to them (Random sampling is achieved using the method mentioned in Sample \$ Collide algorithm). Other vehicles are assigned 0. Neighbors periodically exchanges information with one of its neighbours selected at random, and update their value to the average of the two. In steady state, the value in each vehicle will be K/N, where N is the number of vehicles in the network. However, this algorithm needs time to converge. Also, if a node leaves the network during the initial phase of the algorithm after receiving the gossip message, the accuracy of the algorithm decreases significantly [2].

There are also works that estimate local density to dynamically change transmission power [110, 152]. For example, Noureddine and Samira propose an efficient local density estimation method using segmented roads and extended beacons to estimate the number of neighbors around a vehicle [120]. These algorithms can be used collaboratively with our approach to prevent MAC level collision.

7.2 Circled Random Walk Protocol (CRW)

In this section, we describe our first statistical method for estimating the number of nodes in a large VP2P network.

7.2.1 Description of the Protocol

A query node a sends out a number of probe messages. Each probe independently performs a circled random walk (CRW). The probe carries a globally unique identifier and a hop count. The identifier consists of node a's location and a sequence number. The hop count is initialized to be zero. When a node receives a probe, it records the probe's identifier and increases the hop count in the probe by one. If the node receives the probe for the first time, it forwards the probe to a randomly selected neighbor except for the one from which the probe was just received. If the node receives the probe for the second time, it discards the probe and sends the probe's hop count back to node a. The random walk of a probe terminates once its traversing path forms a circle. An illustration is given in Figure 7-1.

Let X be the length of a circled random walk, which is the number of hops that the probe traverses before it reaches a node for the second time. X is a random number. In

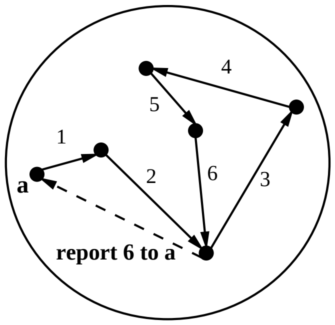

Figure 7-1. Illustration of CRW.

the next subsection, we establish the following mathematical formula that links $E(X)$ to n:

$$E(X) = f(n) = \sum_{i=3}^{n} i(\prod_{j=2}^{i-1}(1 - \frac{j-2}{n-2}))\frac{i-2}{n-2},$$
$$n = f^{-1}(E(X)). \tag{7–1}$$

Using the formula, we can estimate n after we measure $E(X)$ by performing a number of circled random walks and taking the average $\bar{X}$ of the received hop counts. Let $\hat{n}$ be the estimated value of n.

$$\hat{n} = f^{-1}(\bar{X}) \tag{7–2}$$

The pseudo code of the algorithm we use to numerically compute $\hat{n}$ numerically from $\bar{X}$ based on (7–2) is given below. Note that $f(n)$ is a monotonically increasing function.

```
1.    pick a small value $n_1$ and a large value $n_2$ such
      that $f(n_1) < \bar{X}$ and $f(n_2) > \bar{X}$
2.    WHILE $(n_1 \neq n_2)$ DO
3.        let $n_3 = \lfloor (n_1 + n_2)/2 \rfloor$
4.        IF $f(n_3) < \bar{X}$ THEN $n_1 = n_3$ ELSE $n_2 = n_3$
5.    END WHILE
6.    RETURN $n_1$
```

7.2.2 Linking $E(X)$ to n

In this subsection, we derive Equation (7–1) and the variance of X. We also give a way for quick estimation of n. It is less accurate than what is computed by Equation (7–2) but is easier to calculate.

Let $q(i)$ be the probability of not visiting any node twice after a probe moves for i hops in its random walk. The node a that initializes the random walk is a visited node by default. It is obvious that $q(1) = q(2) = 100\%$. Consider the ith hop of the random walk, $\forall i > 2$. The previous $(i-1)$ hops visited i nodes, including node a. The next hop can be any node except for the current node and the previous node of the random walk. The probability for the ith hop to be an unvisited node is $1 - \frac{i-2}{n-2}$. Hence,

$$q(i) = q(i-1)(1 - \frac{i-2}{n-2}).$$

Here, we have used the random neighbor assumption in Section 7.1. It does not work well for low mobility as we will demonstrate through simulations. This assumption will be removed in the next section.

Recursively applying the above equation, we have

$$q(i) = q(1)\prod_{j=2}^{i}(1 - \frac{j-2}{n-2}) = \prod_{j=2}^{i}(1 - \frac{j-2}{n-2}).$$

X is a random variable with a distribution from 3 to n. We know that $q(i-1)$ is the probability of not visiting any node twice after the first $(i-1)$ hops, and $\frac{i-2}{n-2}$ is the conditional probability for the ith hop to reach a node that has been visited previously. Clearly, $q(i-1)\frac{i-2}{n-2}$ is the probability of visiting a node twice after i hops. Hence, the expected value of X is

$$\begin{aligned} E(X) &= \sum_{i=3}^{n} iq(i-1)\frac{i-2}{n-2} \\ &= \sum_{i=3}^{n} i(\prod_{j=2}^{i-1}(1 - \frac{j-2}{n-2}))\frac{i-2}{n-2} \end{aligned}$$

The variance of X is

$$V(X) = \sum_{i=3}^{n}(i - E(X))^2(\prod_{j=2}^{i-1}(1 - \frac{j-2}{n-2}))\frac{i-2}{n-2}.$$

Figure 7-2 shows $E(X)$ with respect to n in log scale. In the same plot, two additional curves, $1.31\sqrt{n}$ and $1.25\sqrt{n}$, are shown to closely overlap with $E(X)$. In fact, numerical computation shows that they are upper and lower bounds of $E(x)$ for a wide range of $n \in [10^3, 10^7]$. This indicates that $E(X) = O(\sqrt{n})$ in this range.

$$1.25\sqrt{n} < E(X) < 1.31\sqrt{n}$$

$$n < 0.64(E(X))^2 < 1.1n$$

Therefore, if the network size is in the range of $[10^3, 10^7]$, then $0.64\bar{X}^2$, which approximates $0.64(E(X))^2$, can be used as a quick but less accurate estimation of n. For accurate estimation, we use the estimator (7–2), which is based on (7–1). We analyze its accuracy below.

7.2.3 Determining the Number of Probes

The number m of probes initially sent from the query node controls the tradeoff between the communication overhead and the estimation accuracy. A wireless transmission is required when a node sends a probe to the next hop. We also know that $E(X) = O(\sqrt{n})$. Hence, the expected overhead of CRW in terms of the number of transmissions made by all nodes is $O(m\sqrt{n})$, which is linear in m.

Our goal is to determine the minimum number of probes that are needed to ensure that the probability for n to fall in the range $[(1-\beta)\hat{n}, (1+\beta)\hat{n}]$ is at least α, where α and β are the parameters of the accuracy requirement. In other words, the estimation has to achieve an α confidence interval that is bounded by $[(1-\beta)\hat{n}, (1+\beta)\hat{n}]$.

First, we prove that $\bar{X}$ is an unbiased estimate of $E(X)$, which is the mean hop count of a circled random walk. Let X_i be the hop count of the circled random walk by the i^{th} probe. As $\bar{X}$ is the average hop counts of probes, we have,

$$\bar{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$$
$$E(\bar{X}) = \frac{1}{m}E(\sum_{i=1}^{m} X_i) = \frac{1}{m}\sum_{i=1}^{m} E(X_i) \tag{7–3}$$

Each X_i is an independent random sample of X. Therefore $E(X_i) = E(X)$. From (7–3), we have $E(\bar{X}) = E(X)$.

Next, we determine the value of m such that the probability for $E(X)$ to fall in the range $[(1-\beta')\bar{X}, (1+\beta')\bar{X}]$ is at least α, where β' is a constant whose value will be determined shortly. The α confidence interval of $E(X)$ is $\bar{X} \pm st^*/\sqrt{m}$, where s is the standard deviation calculated from the received hop counts and t^* is the upper $\frac{1+\alpha}{2}$ point of the t distribution with $(m-1)$ degree of freedom. The value of m is calculated as follows:

$$st^*/\sqrt{m} = \beta'\bar{X}$$
$$m = \frac{(st^*)^2}{(\beta'\bar{X})^2} \tag{7–4}$$

Then, we determine an appropriate value for β'. We know that if m is chosen based on (7–4), then the following inequality is satisfied with probability α:

$$(1-\beta')\bar{X} \le E(X) \le (1+\beta')\bar{X}$$
$$f^{-1}((1-\beta')\bar{X}) \le f^{-1}(E(X)) \le f^{-1}((1+\beta')\bar{X})$$
$$f^{-1}((1-\beta')\bar{X}) \le n \le f^{-1}((1+\beta')\bar{X})$$

Our estimation accuracy requirement is to satisfy $(1-\beta)\hat{n} \le n \le (1+\beta)\hat{n}$ with probability α. To meet this requirement, we select the maximum value of β' that meets the following conditions:

$$f^{-1}((1+\beta')\bar{X}) \le (1+\beta)\hat{n}$$
$$f^{-1}((1-\beta')\bar{X}) \ge (1-\beta)\hat{n}, \tag{7–5}$$

where β is a constant specified in the accuracy requirement, $\bar{X}$ and $\hat{n}$ are results of the CRW execution, and we know how to solve f^{-1} based on the algorithm in Section 7.2.1. Hence, β' can be computed numerically from the above inequalities.

Finally, based on (7–4) and (7–5), we design an iterative process for determining the number of probes that is required to meet a given accuracy requirement specified by α and β. The query node a begins with a certain number (e.g., 50) of probes. After it receives the hop counts of the probes and computes $\bar{X}$ and $\hat{n}$, node a determines β' from (7–5) and then the value of m from (7–4). If the total number m' of probes that have already been sent is equal to or greater than m, it returns the current value of $\hat{n}$ as the estimation of n. Otherwise, if m' is less than m, node a send $(m - m')$ more probes and use the returned hop counts to refine the value of m. This process continues until the total number of probes that have been sent is equal to or greater than m.

7.3 Tokened Random Walk Protocol (TRW)

In this section, we describe our second statistical method for estimating the number of nodes in a large mobile P2P network.

7.3.1 Description of the Protocol

We randomly distribute a number T of tokens to nodes in the network. The problem of how to randomly distribute tokens will be discussed in Section 7.3.3. The nodes that hold tokens are called the tokened "nodes".

A query node a sends out a number of probe messages. Each probe independently performs a tokened random walk (TRW) and counts the number of hops that it has traversed. When a node that does not hold a token receives a probe, it increases the hop count in the probe by one and forwards the probe to one of its neighbors. When a tokened node receives a probe, it discards the probe and sends the probe's hop count back to node a. An illustration of TRW is shown in Fig 7-3.[3]

Let Y be the length of a tokened random walk, which is the number of hops that a probe traverses before it reaches a tokened node. In the next subsection, we establish a mathematical formula that links $E(Y)$ to n:

$$\begin{aligned} E(Y) = g(n) = & \sum_{i=1}^{n-T+1} i \frac{(n-i+1)!(n-T)!}{n!(n-i-T+1)!} \frac{T}{n-i+1}, \\ n = g^{-1}(E(Y)). & \end{aligned} \tag{7–6}$$

Using the formula, we can estimate n after we measure $E(Y)$ by performing a number of tokened random walks and taking the average $\bar{Y}$ of the received hop counts. The estimated number $\hat{n}$ of nodes is calculated as

$$\hat{n} = g^{-1}(\bar{Y}). \tag{7–7}$$

7.3.2 Linking $E(Y)$ to n

Let $p(i)$ be the probability of not reaching a tokened node after a probe has visited i nodes. This happens when and only when the set of T tokened nodes does not overlap with the set of i visited nodes. Recall that the tokened nodes are randomly chosen from the network. $\binom{T}{n}$ is the number of different ways for picking the set of tokened nodes. $\binom{T}{n-i}$ is the number of different ways for picking them from the nodes that are not visited. Hence,

$$p(i) = \frac{\binom{T}{n-i}}{\binom{T}{n}} = \frac{(n-i)!(n-T)!}{n!(n-i-T)!}$$

Y is a random variable with a distribution from 1 to $n-T+1$. We know that $p(i-1)$ is the probability of not reaching a tokened node after the first $(i-1)$ hops. The conditional

[3] In practice, we may allow the probes to make some random walks before starting to count the hops. Our simulation shows that it increases the randomness and returns better estimation results.

probability for the ith visited node to have a token is $\frac{T}{n-2}$. Clearly, $p(i-1) \cdot \frac{T}{n-2}$ is the probability of reaching the first tokened node at the ith hop. The expected value of Y is

$$\begin{aligned} E(Y) &= \sum_{i=1}^{n-T+1} i \cdot p(i-1) \cdot \frac{T}{n-2} \\ &= \sum_{i=1}^{n-T+1} i \frac{(n-i+1)!(n-T)!}{n!(n-i-T+1)!} \frac{T}{n-2}. \end{aligned}$$

The variance of Y is

$$V(Y) = \sum_{i=1}^{n-T+1} (i - E(Y))^2 \frac{(n-i+1)!(n-T)!}{n!(n-i-T+1)!} \frac{T}{n-2}.$$

7.3.3 Random Token Distribution

The process of deriving (7–6) in the previous subsection does not require the assumption of a randomly-connected network. Instead, it requires that the tokens are randomly distributed. Essentially, we shift the random topology requirement of CRW to a random token distribution requirement, which can be implemented by the following distributed algorithm: Every token independently moves around in the network. When a node receives a token, it holds the token for a period that is inversely proportional to its current number of neighbors, i.e., the node's degree. After that period, it forwards the token to a neighbor selected uniformly at random. The transmission of a token may be piggybacked in the periodic hello exchange between the neighbors [135]. In order to support communications in a mobile network, each node has to periodically broadcast a hello packet (or called a beacon), which allows the node to learn its new neighbors. One bit in the hello packet can be used to encode a token transmission. '0' means there is no token carried in the hello packet, and '1' means there is.

We prove that, in the steady state, the rate at which a node receives tokens is proportional to the node's degree. Because the holding time after each receipt of a token is inversely proportional to the node's degree, the aggregate holding time at every node is about the same, which ensures the uniform random distribution of tokens.

Consider an arbitrary token. The movement of the token in the network is modeled as a discrete-time finite-state Markov chain. The current state is i if node i holds the token. The set of states is $\{1, ..., n\}$ for the n nodes. Let N_i be the set of neighbors of node i. Let $n_i = |N_i|$. The transition probability from state i to state j is

$$p_{ij} = \begin{cases} \frac{1}{n_i} & \text{if } j \in N_i \\ 0 & \text{if } j \notin N_i \end{cases}$$

The matrix of transition probabilities is $P = (p_{ij}, i, j \in [1, n])$. Let $\pi = (\pi_1, \pi_2, ..., \pi_n)$ be the stationary distribution of the Markov chain, which satisfies $\pi P = \pi$ and $\sum_{i=1}^{n} \pi_i = 1$, where π_i represents how often node i has (or receives) the token in the steady state of the stochastic token movement process. $\pi P = \pi$ can be rewritten as

$$\sum_{j \in N_i} \frac{\pi_j}{n_j} = \pi_i, \quad i \in [1, n].$$

It can be easily verified that the solution is

$$\pi_i = \frac{n_i}{\sum_{j=1}^{n} n_j}, \quad i \in [1, n].$$

Therefore, the rate at which a node receives a particular token is proportional to its degree. Because all tokens move independently, the rate at which a node receives tokens is also proportional to its degree. Our goal is for each node to have an equal chance of being a tokened node. To compensate the rate difference, when a node receives a token, the holding time should be inversely proportional to the degree. In particular, we may randomly pick a holding time from an exponential distribution whose mean is inversely proportional to the degree. If a node's degree changes during this period, we use the degree when it first receives the token.

In a mobile environment, a node may depart from the network or be powered down. To support the TRW protocol, we require that if a node has a token, it must transmit the token to a neighbor before it departs or is powered down. There may be rare cases where a node crashes to cause a permanent token loss. One solution is to release a new set of tokens periodically. Each release is identified by a sequence number s. The sequence number is initialized to be one and increased by one for every subsequent release. Each token must be associated with the sequence number that identifies which release it belongs to. When a node transmits a token to a neighbor piggybacked in a hello packet, instead of using one bit, the packet carries a sequence number. If the number is zero, it means no token; otherwise, it means a token of a certain release.

Each query is made with a sequence number, and each probe of the query carries that sequence number. When a node receives a probe, only if it has a token of the same sequence number, it will discard the probe and send the probe's hop count back to the query node.

Let D be the period between two consecutive token releases. Suppose D is chosen to be sufficiently large, such that tokens will be randomly distributed after they are released for a time period of D. Now, after tokens of sequence number s are released and before tokens of sequence number $s + 1$ are released, all queries can be made with sequence number $s - 1$. Moreover, all previous tokens (with sequence numbers $s - 2$ or smaller) are no longer useful. To remove outdated tokens, when a node receives a token with a new sequence number s, it knows that it should remove any token with sequence number $s - 2$ or less that it may receive in the future. Hence, as tokens with a new sequence number start to travel in the network, nodes begin to remove old, useless tokens.

7.3.4 Determining the Number of Probes

We now determine the minimum number of probes that are needed to ensure that the probability for n to fall in the range $[(1 - \beta)\hat{n}, (1 + \beta)\hat{n}]$ is at least α.

First, we prove that $\bar{Y}$ is an unbiased estimate of $E(Y)$, which is the mean hop count of a tokened random walk. Let Y_i be the hop count of the tokened random walk by the i^{th} probe. As $\bar{Y}$ is the average hop count of the probes, we have,

$$\begin{aligned} \bar{Y} &= \frac{1}{m}\sum_{i=1}^{m} Y_i \\ E(\bar{Y}) &= \frac{1}{m}E(\sum_{i=1}^{m} Y_i) = \frac{1}{m}\sum_{i=1}^{m} E(Y_i) \end{aligned} \tag{7–8}$$

Each Y_i is an independent random sample of Y. Therefore $E(Y_i) = E(Y)$. From (7–8), we have $E(\bar{Y}) = E(Y)$.

Next, based on the same process as Section 7.2.3, it can be shown that if the number of probes is

$$m = \frac{(st^*)^2}{(\beta'\bar{Y})^2}, \tag{7–9}$$

then the probability for $E(Y)$ to fall in $[(1-\beta')\bar{Y}, (1+\beta')\bar{Y}]$ is at least α, where β' is a given value, s is the standard deviation calculated from the received hop counts, and t^* is the upper $\frac{1+\alpha}{2}$ point of the t distribution with $(m-1)$ degree of freedom. We have the following inequalities:

$$\begin{gathered} (1-\beta')\bar{Y} \le E(Y) \le (1+\beta')\bar{Y} \\ g^{-1}((1-\beta')\bar{Y}) \le g^{-1}(E(Y)) \le g^{-1}((1+\beta')\bar{Y}) \\ g^{-1}((1-\beta')\bar{Y}) \le n \le g^{-1}((1+\beta')\bar{Y}). \end{gathered}$$

The accuracy requirement is $(1-\beta)\hat{n} \le n \le (1+\beta)\hat{n}$ with probability α. To meet this requirement, we select the maximum value of β' that meets the following conditions:

$$\begin{gathered} g^{-1}((1+\beta')\bar{Y} \le (1+\beta)\hat{n} \\ g^{-1}((1-\beta')\bar{Y} \ge (1-\beta)\hat{n}. \end{gathered} \tag{7–10}$$

Given an accuracy requirement specified by α and β, the query node a begins with a certain number of probes. After it receives the hop counts of the probes, node a determines β' from (7–10) and then the value of m from (7–6). If the total number m' of probes that have already been sent is equal to or greater than m, it returns the current value of $\hat{n}$ as the estimation of n. Otherwise, if m' is less than m, node a send $(m-m')$

more probes and use the returned hop counts to refine the value of m. This process continues until the total number of probes that have been sent is equal to or greater than m.

7.4 Simulation

In this section, we use simulations to evaluate the performance of our cardinality estimation methods in terms of accuracy and overhead. Our simulations are performed based on two models: (1) street model, which estimates the number of moving vehicles in the streets of a city district; (2) random waypoint model [10, 61, 64, 118], which is used to demonstrate the performance of our methods for other mobile P2P systems that are not restricted to streets.

7.4.1 Simulation Setup for Street Model

Suppose future cars are equipped with wireless devices that not only support user applications but also assist transportation management functions such as information gathering. One such function may be estimating the number of moving vehicles. When a vehicle is powered down, we can consider it to be in parking status (in most cases). When a vehicle is powered up with its wireless device running, we can consider it be to in moving status (in most cases).

Consider a square area of 1000 meters by 1000 meters. Let the time unit be a second. Suppose the streets and avenues are arranged in a grid pattern and the distance between two adjacent streets (or avenues) is 100 meters. The number n of moving vehicles ranges from 1,000 to 5,000. The transmission range of a vehicle is 100 meters. Each vehicle is capable of forwarding messages to any other vehicle in its transmission range. Each moving vehicle selects an arbitrary intersection and a velocity in the range of $[20, 40]$ miles/hour, i.e., $[32, 64]$ kilometers/hour. Then it moves first along a street and then along an avenue or vice versa towards the destination at the selected speed. For simplicity, we do not implement variable speed and stops at signal lights.

We use CRW and TRW to estimate the number n of vehicles in the area. Both methods send probe messages to the network. For CRW, when a vehicle receives a probe message, it holds the message for 10 seconds before forwarding it. This holding period allows each vehicle to have chance to change neighbors. TRW does not need the random-neighbor assumption, and therefore the holding period for probe messages is not necessary. For TRW, if a vehicle receives a token, it keeps the token for 10 to 100 seconds before passing it to another vehicle. The actual time that it holds the token is inversely proportional to the number of its neighbors. We set the number of tokens in the network to be 100. In order to increase randomness, we let probe messages to make some random walks before starting to count the hops. Assume each vehicle knows its geographic location (possibly through GPS). A vehicle also knows approximately the geographic locations of its neighbors, which are piggybacked in the hello exchanges. We implement geographic routing [67] for the hop count result to be sent back to the query node whenever a probe message completes its random walk. The location of the query node is carried by the probe message.

7.4.2 Accuracy and Overhead under Street Model

Figure 7-4 presents the estimations made by CRW. Each point in the figure represents one simulation result; the x coordinate of the point is the actual number n of vehicles, and the y coordinate is the estimated number $\hat{n}$. An estimation is more accurate if the point is closer to the equality line, $y = x$. In the leftmost plot, we let $\alpha = 95\%$ and $\beta = 0.2$, which require that 95% of estimated numbers should fall within $\pm$ 20% of the true numbers. In the middle plot, $\alpha = 99\%$ and $\beta = 0.2$. In the rightmost plot, $\alpha = 95\%$ and $\beta = 0.1$. Our statistical measurement based on the data points in the figure confirms that the accuracy requirement is indeed met. Figure 7-5 presents the estimations made by TRW. Again, the accuracy requirement is met.

Next, we investigate the overhead of the two methods. For CRW, the overhead is caused by probe message transmissions. For TRW, the overhead comes from both

Table 7-1. Average completion time of Circled Random Walk (CRW)

n	1000	2000	3000	4000	5000
CRW Time	6’41”	9’18”	11’35”	13’32”	14’59”

probe transmissions and token transmissions. A probe transmission is a separate packet with headers at MAC, network, and application (CRW/TRW) layers, as well as physical-layer cost. A token transmission is piggybacked in the hello exchanges, which incurs far smaller overhead than a probe. Hence, we only count the probe message transmissions in overhead comparison. Figure 7-6 compares CRW and TRW in terms of the total number of message transmissions. For each data point, we make 100 simulation runs, average the overhead results, and present the mean overhead and its standard deviation. TRW outperforms CRW. Its communication overhead is only a fraction of CRW’s overhead.

For a circled random walk, the time for a probe to travel each hop has two components: the holding period and the transmission time from one node to the next. When comparing the holding time (10 seconds in this simulation), the transmission time is negligible. Hence, our simulation only considers the holding time. The average completion time of CRW (including all its probes) is shown in Table 7-1. Because of the long holding period, the time for CRW is significant. However, it is probably acceptable in practice to spend six to fifteen minutes for the task of measuring the number of vehicles in a city distinct. Because TRW does not need any holding period, its completion is much quicker. When $n = 1000$ and the number of tokens is 100, each tokened random walk has an average of just 10 hops, and only the transmission time is needed.

7.4.3 Simulation Setup for Random Waypoint Model

We study how our methods will perform for arbitrary mobile P2P networks under the random waypoint model. Consider a square area of 1000×1000 units of length. The number n of wireless nodes ranges from 1,000 to 5,000. The transmission range of the nodes is 100 units of length. Each node is capable of forwarding messages to any other node in its transmission range. We use the random waypoint model to simulate the mobility of nodes: Each node randomly selects a position within the area as destination and a velocity in the range of $[5, 30]$. Then it moves towards the destination at the selected speed. After it arrives at the destination, it moves again with a new destination and a new velocity. Other parameters are similar to those in Section 7.4.1 with "second" being replaced with "unit of time", in correspondence with the abstract term of "unit of length" above.

7.4.4 Accuracy and Overhead under the Random Waypoint Model

We first present the estimation accuracy of the two methods. We vary the number n of wireless nodes from 1000 to 5000. We run the simulation under three different accuracy requirements: $\alpha = 95\%, \beta = 20\%$; $\alpha = 99\%, \beta = 20\%$; and $\alpha = 95\%, \beta = 10\%$.

Figures 7-7 and 7-8 compare the actual number of nodes, n, and the estimated number, $\hat{n}$. Each point in the figures represents one simulation result; the x coordinate of the point is the actual number n of nodes, and the y coordinate is the estimated number $\hat{n}$. In the leftmost plot, the requirement specified by α and β is that the estimated number $\hat{n}$ should have a 95% chance to fall within $\pm$ 20% of the actual number n. Our statistical measurement based on the data points in the figure shows that this requirement is met. The same is true for other plots in Figures 7-7 and 7-8 with different combinations of α and β values.

Figure 7-9 compares CRW and TRW in terms of the total number of message transmissions. From the figure, we observe that TRW is much more efficient, only requiring 20% to 30% of the overhead incurred by CRW.

7.4.5 Performance under Low Mobility

As we have shown in the previous simulations, CRW produces good estimation when the nodes move relatively fast. Even though the random neighbor assumption is not accurate, it approximates well for the highly mobile scenarios. However, our next set of simulations show that CRW does not work well for networks of low mobility. On the contrary, TRW remains accurate for those networks.

The simulation setup is the same as described in Section 7.4.3 except that the velocity of each node is randomly selected from $[3, 10]$, instead of $[5, 30]$, when the node moves from one location to another. Figure 7-10 shows that CRW consistently under-estimates the number of nodes in a network of low mobility. That is because nodes have relatively stable neighbors and as they forward probe messages, the messages tend to make short cycles within small neighborhood, which leads to the under-estimation.

Figure 7-11 shows that TRW remains accurate for networks of low mobility because its design does not rely on the random neighbor assumption.

Figure 7-12 compares the overhead of the two methods. Again, TRW outperforms CRW significantly.

Overall, our simulation results are consistent across the street model and the random waypoint model. In summary, TRW is a better method in terms of communication overhead, and it is also better in terms of accuracy for low-mobility networks. CRW is able to produce results that meet the accuracy requirement in VP2P or other mobile networks with relatively high mobility. Its advantage is simplicity (without the assistance of tokens).

7.5 Summary

This chapter investigates the problem of how to estimate the number of nodes in a large VP2P network. We propose two statistical methods, called the circled random walk and the tokened random walk, respectively. The proposed methods are adjustable for estimation accuracy and communication overhead. Their estimation process involves only a subset of the nodes, and the estimation errors can be made arbitrarily small. While the circled random walk method requires a randomized neighboring relationship among the nodes and it works well in high-mobility networks, the tokened random walk is more practical for all types of VP2P or other mobile networks because it removes the random-neighbor requirement through a randomized token distribution process.

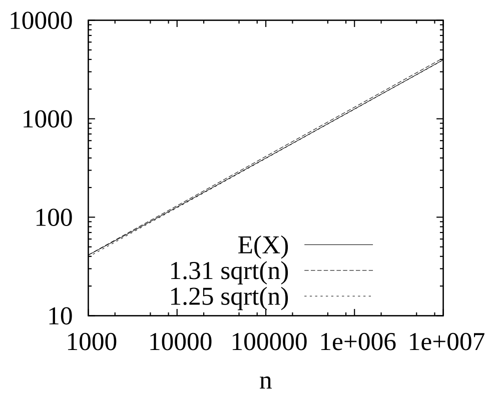

Figure 7-2. $E(X)$ as a function of n.

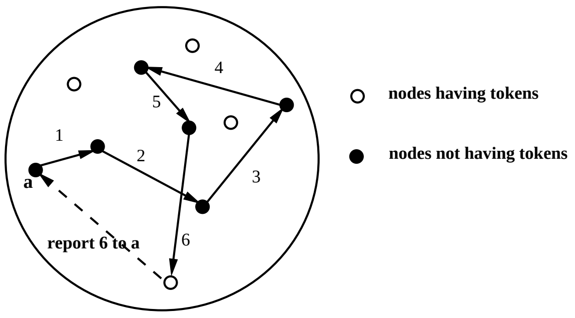

Figure 7-3. Illustration of TRW.

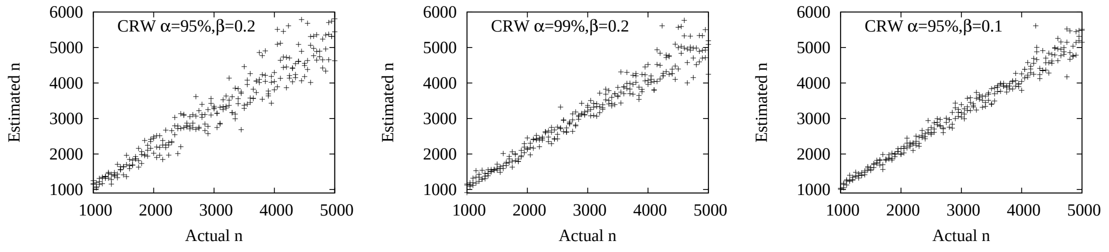

Figure 7-4. CRW's estimation accuracy under the street model.

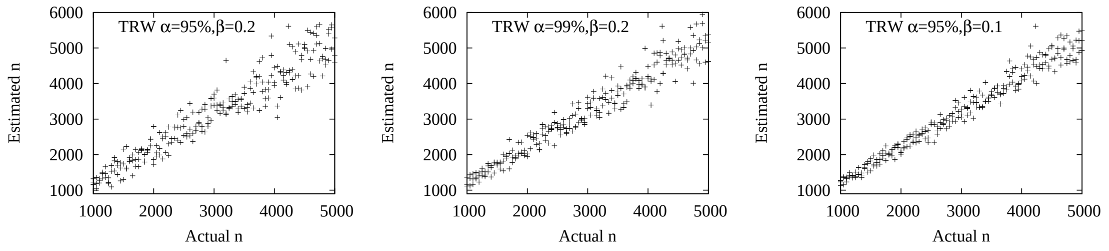

Figure 7-5. TRW's estimation accuracy under the street model.

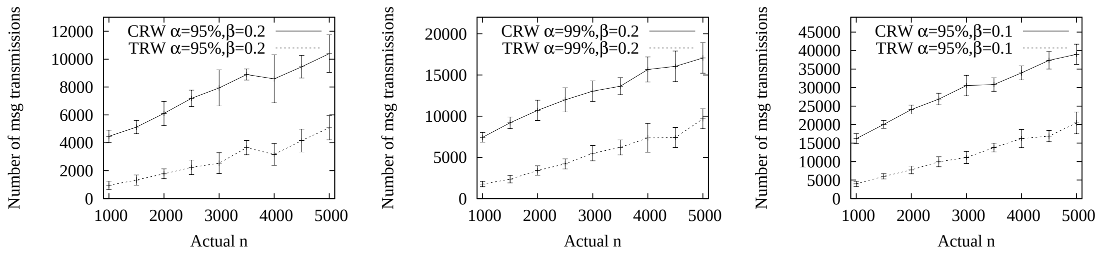

Figure 7-6. Number of message transmissions for each estimation, under the street model.

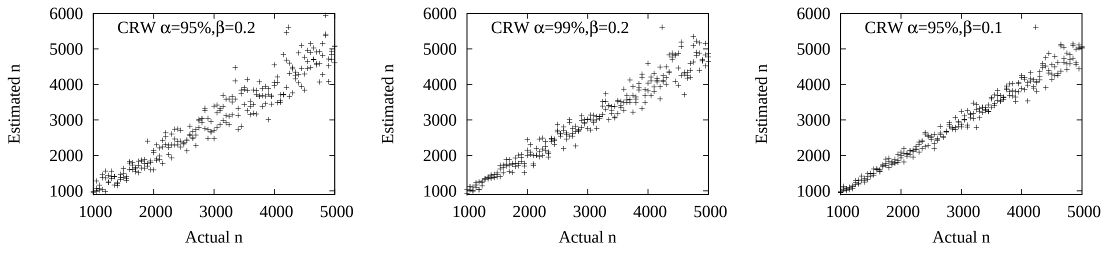

Figure 7-7. CRW's estimation accuracy under the random waypoint model with nodal moving velocity in range of $[5, 30]$.

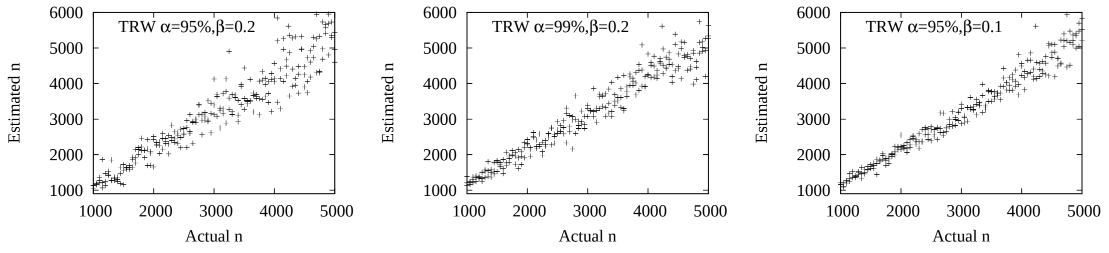

Figure 7-8. TRW's estimation accuracy under the random waypoint model with nodal moving velocity in range of $[5, 30]$.

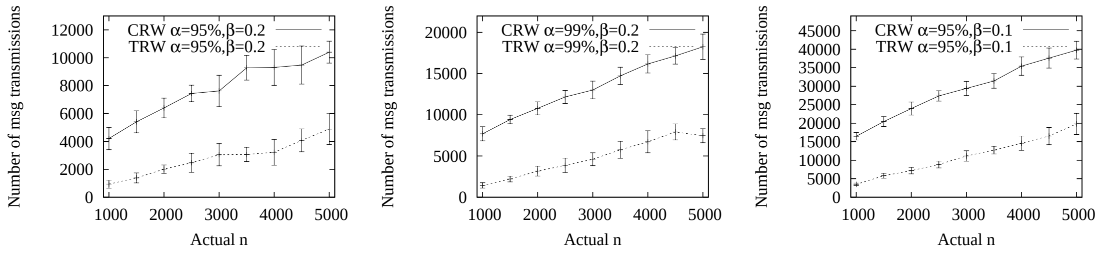

Figure 7-9. Number of message transmissions for each estimation, under the random waypoint model with nodal moving velocity in range of $[5, 30]$.

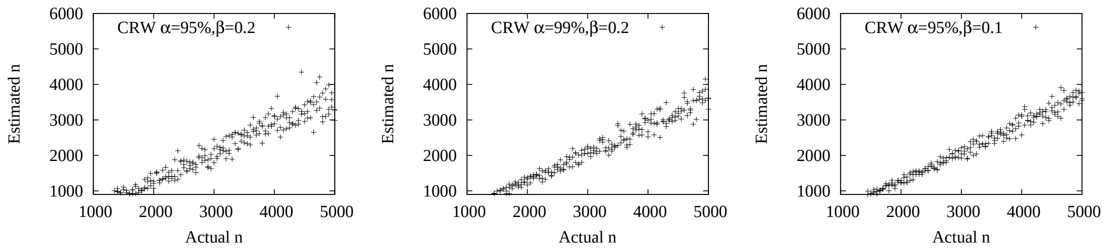

Figure 7-10. CRW's estimation accuracy under the random waypoint model with nodal moving velocity in range of $[3, 10]$.

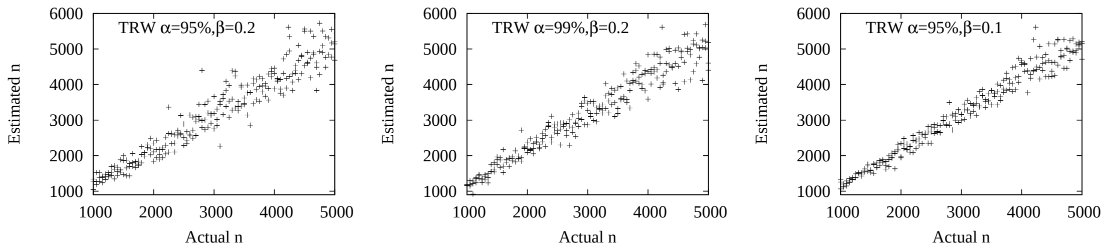

Figure 7-11. TRW's estimation accuracy under the random waypoint model with nodal moving velocity in range of $[3, 10]$.

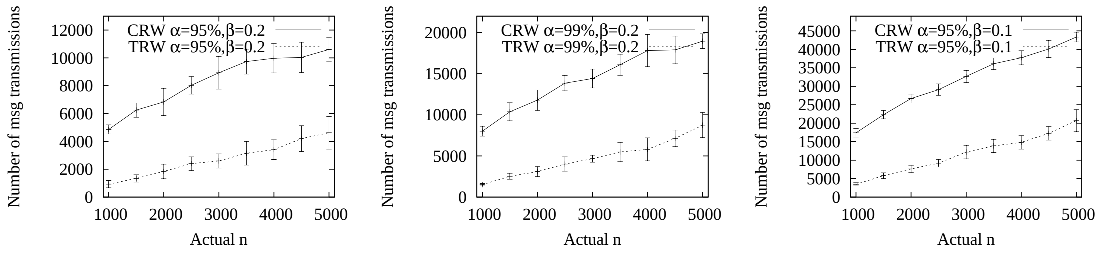

Figure 7-12. Number of message transmissions for each estimation, under the random waypoint model with nodal moving velocity in range of $[3, 10]$.

CHAPTER 8
CONCLUSION AND FUTURE WORK

In this work, we propose several space-time efficient data structures protocols and place them into various application scenarios. We first study a new family of space-time efficient Bloom filters that provide the same compactness as the Bloom filter, but requires much fewer memory access for each lookup. We open new design space with trade-offs among space usage, false positive ratio, memory access, and number of hash bits. We show the design trade-offs through theoretical analysis and experiments. Our proposed family of Bloom filter variant – the Bloom-g filter constantly uses less number of memory access and less hash bits than the Bloom filter when k is the same. If we allow the Bloom-g filter to use as many hash bits as standard Bloom filter, it will produce even better result in terms of the false positive ratio. When a small k is used, the Bloom-1 filter makes only one memory access and has comparable false positive ratio as standard Bloom filter. When the optimal k is used and the load factor is small, Bloom-1 introduces much more false positives than standard Bloom filter due to unbalanced load among different words. As a compensation, the Bloom-2 filter and Bloom-3 filter makes 2 and 3 memory accesses respectively, yet produce comparable false positive ratio as standard Bloom filter. As a conclusion, we suggest to use Bloom-1 as a substitute of the Bloom filter when k is small, and use Bloom-2 or Bloom-3 to replace the Bloom filter with optimal k.

In order to allow more design trade-offs, we propose another Bloom filter variant – the Bloom-α filter, which makes $1.x$ memory access on average, and produces false positive ratios that are between Bloom-1 and Bloom-2.

Then, we further propose an idea of representing multi-set, where a set ID is associated with each member element. Our idea is to separate membership encoding and set ID storage in two data structures, called “index encoder” and “set-id table”, respectively. The index encoder adopts ideas from the Bloom-g filter, which takes

g memory accesses to check the membership of a given element and to locate the index where set-id is stored. The set-id table is a multi-hashing table employing a novel load-to-left, candidate-to-right policy for element placement. These techniques together allow the proposed multi-set membership lookup function to work in very compact memory, take only a few memory accesses and hash operations for each lookup, and have much lower error probabilities when comparing with alternative data structures.

Next, we design another Bloom filter variant called adaptive Bloom filter for joining two sets stored distributively in the network. The idea is to iteratively eliminate non-candidates using small Bloom filters. We transform the problem into an optimization problem. Our design outperforms the Bloom filter and compressed Bloom filter in terms of exchanged message size.

After that, we apply the Bloom-1 filter idea together with the partitioned Bloom filter in the design of efficient information collection protocols for large-scale RFID systems. Our primary objective is to lower energy consumption by tags in order to extend their lifetime. TOP shares similarity with the Bloom-1 filter, where tags only listen to one vector that the reader broadcasts for necessary information. ETOP is an extension of TOP, adopting the idea of partitioned Bloom filter. The new protocols can be configured to achieve $O(1)$ energy cost per tag. Performance tradeoff between energy cost and execution time can be made by controlling the size of the reporting-order vector. Simulation results show that the new protocols are able to cut energy cost by more than an order of magnitude, when comparing with other protocols.

Lastly, we design two efficient protocols for estimating the network size of mobile vehicular peer-to-peer networks. Existing solution either requires flooding the network, or takes too long to converge. We present two novel statistical methods, called the circled random walk and the tokened random walk, to address this interesting problem. While the circled random walk method requires a randomized neighboring relationship among the nodes and it works well in high-mobility networks, the tokened random walk is more practical for all types of VP2P or other mobile networks because it removes the random-neighbor requirement through a randomized token distribution process. Both methods provide cardinality estimation by involving only a small subset of the nodes (vehicles). They make tradeoff between overhead and estimation accuracy. The estimation error can be made arbitrarily small at the expense of larger overhead.

Future research will go on. We will further explore the design space of the adaptive Bloom filter, for example, when it is used for multi-party information sharing. We will test it using real network traces. Also, for the multi-set membership checking problem, we proposed idea for element update/deletion. In practice, there will be practical issues, such as the introduction of false negatives. How to reduce the down side as much as possible whiling utilizing its advantage, remains an open problem.

REFERENCES

[1] "Information Technology – Radio Frequency Identification for Item Management Air Interface – Part 6: Parameters for Air Interface Communications at 860-960 MHz." *Final Draft International Standard ISO 18000-6* (2003).

[2] Akhtar, N., Ergen, S., and Ozkasap, O. "Analysis of Distributed Algorithms for Density Estimation in VANETs (Poster)." *Vehicular Networking Conference* (2012): 157–164.

[3] Almeida, P., Baquero, C., Preguica, N., and Hutchison, D. "Scalable Bloom Filters." *Information Processing Letters* 101 (2007).6: 255–261.

[4] Alvarez-Icaza, L., Munoz, L., Sun, X., and Horowitz, R. "Adaptive Observer for Traffic Density Estimation." *Proc. of American Control Conference* 3 (2004): 2705–2710.

[5] Artimy, M. "Local Density Estimation And Dynamic Transmission-Range Assignment In Vehicular Ad Hoc Networks." *IEEE Transactions on Intelligent Transportation Systems* 8 (2007).3: 400–412.

[6] Bhandari, N., Sahoo, A., and Iyer, S. "Intelligent Query Tree (IQT) Protocol to Improve RFID Tag Read Efficiency." *Proc. of IEEE ICIT* (2006).

[7] Bloom, B. H. "Space/Time Trade-offs in Hash Coding with Allowable Errors." *Communications of the ACM* 13 (1970).7: 422–426.

[8] Bonomi, F., Mitzenmacher, M., Panigrah, R., Singh, S., and Varghese, G. "Beyond Bloom Filters: from Approximate Membership Checks to Approximate State Machines." *ACM SIGCOMM Computer Communication Review* 36 (2006).4: 315–326.

[9] Bose, P., Morin, P., Stojmenovic, I., and Urrutia, J. "Routing with Guaranteed Delivery in Ad Hoc Wireless Networks." *Proc. of 3rd Int'l Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DialM)* (1999).

[10] Broch, J., Maltz, D., Johnson, D., Hu, Y., and Jetcheva, J. "Multi-Hop Wireless Ad Hoc Network Routing Protocols." *Proc. of ACM Mobicom* (1998): 85–97.

[11] Broder, A. and Karlin, A. "Multilevel Adaptive Hashing." *Proc. of ACM-SIAM SODA* (1990).

[12] Broder, A. and Mitzenmacher, M. "Network Applications of Bloom Filters: A Survey." *Internet Mathematics* 1 (2002).4: 485–509.

[13] Brodkin, J. "A Wireless Router That Tracks User Activity – But for a Good Reason." *Ars Technica* (2013).

URL http://arstechnica.com/gadgets/2013/01/a-wireless-router-that-tracks-user-activity-but-for-a-good-reason/

[14] Bu, K., Xiao, B., Xiao, Q., and Chen, S. “Efficient Pinpointing of Misplaced Tags in Large RFID Systems.” *IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)* (2011): 287–295.

[15] Budianu, C., David, S., and Tong, L. “Estimation of the Number of Operating Sensors in Large-Scale Sensor Networks with Mobile Access.” *IEEE Transcations on Signal Processing* 54 (2006).5: 1703–1715.

[16] Burden, R.L. and Faires, J.D. “Numerical Analysis.” *Brooks/Cole* 6 (2001).

[17] Canim, M., Mihaila, G.A., Bhattacharhee, B., Lang, C.A., and Ross, K.A. “Buffered Bloom Filters on Solid State Storage.” *VLDB ADMS Workshop* (2010).

[18] Capetenakis, J. I. “Tree Algorithms for Packet Broadcast Channels.” *IEEE Transactions on Information Theory* 25 (1979).5.

[19] Cha, J. R. and Kim, J. H. “Dynamic Framed Slotted ALOHA Algorithms using Fast Tag Estimation Method for RFID Systems.” *Proc. of IEEE CCNC* (2006): 768 – 772.

[20] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E. “Bigtable: A Distributed Storage System for Structured Data.” *ACM Transactions on Computer Systems (TOCS)* 26 (2008).2: 4.

[21] Chang, F., Feng, W., and Li, K. “Approximate Caches for Packet Classification.” *Proc. of IEEE Infocom* 4 (2004): 2196–2207.

[22] Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. “The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables.” *Proc. of ACM SODA* (2004): 30–39.

[23] Chen, H., Jin, H., Chen, L., Liu, Y., and Ni, L. “Optimizing Bloom Filter Settings in Peer-to-peer Multikeyword Searching.” *IEEE Transactions on Knowledge and Data Engineering* 24 (2012).4: 692–706.

[24] Chen, M., Luo, W., Mo, Z., Chen, S., and Fang, Y. “An Efficient Tag Search Protocol in Large-Scale RFID Systems.” *Proc. of IEEE INFOCOM* (2013): 899–907.

[25] Chen, S., Li, J., and Zhang, M. “Polling Protocols in RFID Systems.” *Internal Technical Report, Network Information Center, University of Shanghai for Science and Technology, Shanghai, China* (2010).

[26] Chen, S. and Nahrstedt, K. "Maxmin Fair Routing in Connection-oriented Networks." *Proc. of Euro-Parallel and Distributed Systems Conference (Euro-PDS98)* (1998): 163–168.

[27] Chen, S. and Shavitt, Y. "SoMR: A Scalable Distributed Qos Multicast Routing Protocol." *Journal of Parallel and Distributed Computing* 68 (2008).2: 137–149.

[28] Chen, S., Zhang, M., and Xiao, B. "Efficient Information Collection Protocols for Sensor-augmented RFID Networks." *Proc. of IEEE INFOCOM* (2011): 3101–3109.

[29] Chen, W., Che, W., Wang, X., Huang, C., Yan, N., Min, H., and Tan, J. "A Two-stage Wake-up Circuit for Semi-Passive RFID Tag." *IEEE International Conference on ASIC* (2009): 553–556.

[30] Chen, Y., Kumar, A., and Xu, J. "A New Design of Bloom Filter for Packet Inspection Speedup." *Proc. of IEEE GLOBECOM* (2007): 1–5.

[31] Christensen, K., Roginsky, A., and Jimeno, M. "A New Analysis Of The False Positive Rate Of A Bloom Filter." *Information Processing Letters* 110 (2010).21: 944–949.

[32] Christofides, N. *Graph Theory: An Algorithmic Approach*, vol. 8. Academic press New York, 1975.

[33] Cicho, J., Lemiesz, J., and Zawada, M. "On Cardinality Estimation Protocols for Wireless Sensor Networks." *Ad-Hoc, Mobile, and Wireless Networks* (2011): 322–331.

[34] Cichon, J., Lemiesz, J., Szpankowski, W., and Zawada, M. "Two-Phase Cardinality Estimation Protocols For Sensor Networks With Provable Precision." *Wireless Communications and Networking Conference (WCNC)* (2012): 2009–2013.

[35] Cohen, S. and Matias, Y. "Spectral Bloom Filters." *Proc. of ACM SIGMOD* (2003): 241–252.

[36] Coifman, B. "Estimating Density and Lane Inflow on a Freeway Segment." *Transportation Research Part A: Policy and Practice* 37 (2003).8: 689–701.

[37] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems: Concepts and Design*. Addison-Wesley, ISBN 0-201-18059-6, 2011.

[38] Debnath, B., Sengupta, S., Li, J., Lilja, D.J., and Du, D.H.C. "BloomFlash: Bloom Filter on Flash-based Storage." *Proc. of ICDCS* (2011): 635–644.

[39] Deng, F. and Rafiei, D. "Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters." *Proc. of ACM SIGMOD* (2006): 25–36.

[40] Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. “Longest Prefix Matching Using Bloom Filters.” *Proc. of ACM SIGCOMM* (2003): 201–212.

[41] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Heide, F., Rohnert, H., and Tarjan, R. “Dynamic Perfect Hashing: Upper and Lower Bounds.” *SIAM Journal on Computing* 23 (1994).4: 738–761.

[42] Dimitriou, T. “A Secure and Efficient RFID Protocol that could make Big Brother (partially) Obsolete.” *Proc. of IEEE PerCom* (2006): 6–pp.

[43] Dimitropoulos, X., Hurley, P., and Kind, A. “Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters.” *ACM SIGCOMM Computer Communication Review* 38 (2008).1: 7–16.

[44] Ding, N., Bi, X., and Zhang, D. “An Efficient Hybrid Hierarchical Trie Packet Classification Algorithm Based on No Prefix Relationship.” *Journal of Computational Information Systems* 9 (2013).22: 9193–9202.

[45] Estan, C. and Varghese, G. “New Directions in Traffic Measurement and Accounting.” *Proc. of ACM SIGCOMM* 32 (2002).4: 323–336.

[46] F. Hao, M. Kodialam and Lakshman, T.V. “Building High Accuracy Bloom Filters Using Partitioned Hashing.” *Proc. of ACM SIGMETRICS* 35 (2007).1: 277–288.

[47] Firewall, Cisco IOS. “Context-Based Access Control (CBAC): Introduction and Configuration.” (2008).

URL `http://www.cisco.com/en/US/products/sw/secursw/ps1018/products_tech_note09186a0080094e8b.shtml`

[48] Fredkin, E. “Trie Memory.” *Communications of the ACM* 3 (1960).9: 490–499.

[49] Gardner, W. David. “Researchers Transmit Optical Data At 16.4 Tbps.” *Information Week* (2008).

[50] Garelli, L., Casetti, C., Chiasserini, C., and Fiore, M. “Mobsampling: V2V Communications for Traffic Density Estimation.” *Vehicular Technology Conference* (2011): 1–5.

[51] Gazis, D. C. and Knapp, C. H. “On-Line Estimation of Traffic Densities from Time-Series of Flow and Speed Data.” *Transportation Science* 5 (1971).3: 283–301.

[52] Goodrich, M. and Mitzenmacher, M. “Invertible Bloom Lookup Tables.” *Allerton Conference on Communication, Control, and Computing (Allerton)* (2011): 792–799.

[53] Guo, D., Liu, Y., Li, X., and Yang, P. "False Negative Problem of Counting Bloom Filter." *IEEE Transactions on Knowledge and Data Engineering* 22 (2010).5: 651–664.

[54] Guo, D., Wu, J., Chen, H., Yuan, Y., and Luo, X. "The Dynamic Bloom Filters." *IEEE Transactions on Knowledge and Data Engineering* 22 (2010).1: 120–133.

[55] Han, H., Sheng, B., Tan, C., Li, Q., Mao, W., and Lu, S. "Counting RFID Tags Efficiently and Anonymously." *Proc. of IEEE INFOCOM* (2010): 1–9.

[56] Hao, F., Kodialam, M., and Lakshman, TV. "Incremental Bloom Filters." *Proc. of IEEE INFOCOM* (2008): 1067–1075.

[57] Hao, F., Kodialam, M. S., Lakshman, T. V., and Song, H. "Fast Multiset Membership Testing Using Combinatorial Bloom Filters." *Proc. of IEEE INFOCOM* (2009).

[58] Horowitz, E., Sahni, S., and Rajasekaran, S. *Computer Algorithms C++ (Chapter 3.2)*. WH Freeman, 1996.

[59] Hua, Y. and Xiao, B. "A Multi-attribute Data Structure With Parallel Bloom Filters for Network Services." *High Performance Computing (HiPC)* (2006): 277–288.

[60] Huang, S.C.H., Wan, P.J., Jia, X., Du, H., and Shang, W. "Minimum-Latency Broadcast Scheduling in Wireless Ad Hoc Networks." *Proc. of IEEE INFOCOM* (2007): 733–739.

[61] Hyytia, E., Lassila, P., and Virtamo, J. "Spatial Node Distribution of the Random Waypoint Mobility Model with Applications." *IEEE Transactions on Mobile Computing* 5 (2006).6: 680–694.

[62] Jelasity, M. and Montresor, A. "Epidemic-style Proactive Aggregation in Large Overlay Networks." *Proc. of Distributed Computing Systems* (2004): 102–109.

[63] Jerbi, M., Senouci, S., Rasheed, T., and Ghamri-Doudane, Y. "An Infrastructure-Free Traffic Information System for Vehicular Networks." *Vehicular Technology Conference* (2007): 2086–2090.

[64] Johnson, D. and Maltz, D. "Dynamic Source Routing in Ad Hoc Wireless Networks." *Mobile Computing* (1996): 153–181.

[65] Kamiyama, N. and Mori, T. "Simple and Accurate Identification of High-rate Flows by Packet Sampling." *Proc. of IEEE INFOCOM* (2006).

[66] Kanizo, Y., Hay, D., and Keslassy, I. "Optimal Fast Hashing." *Proc. of IEEE INFOCOM* (2009): 2500–2508.

[67] Karp, B. and Kung, H. "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks." *Proc. of ACM MobiCom* (2000): 243–254.

[68] Kim, D., Oh, D., and Ro, W. “Design of Power-Efficient Parallel Pipelined Bloom Filter.” *Electronics letters* 48 (2012).7: 367–369.

[69] Kirsch., A. and Mitzenmacher, M. “Simple Summaries for Hashing with Choices.” *Networking, IEEE/ACM Transactions on* 16 (2008).1: 218–231.

[70] Klair, D., Chin, K., and Raad, R. “On the Energy Consumption of Pure and Slotted Aloha based RFID Anti-collision Protocols.” *Computer Communications* 32 (2009).5: 961–973.

[71] Kodialam, M. and Nandagopal, T. “Fast and Reliable Estimation Schemes in RFID Systems.” *Proc. of ACM MOBICOM* (2006): 322–333.

[72] Kodialam, M., Nandagopal, T., and Lau, W. “Anonymous Tracking using RFID tags.” *Proc. of IEEE INFOCOM* (2007).

[73] Kostoulas, D., Psaltoulis, D., Gupta, I., Birman, K., and Demers, A. “Decentralized Schemes for Size Estimation in Large and Dynamic Groups.” *Network Computing and Applications* (2005): 41–48.

[74] Krishnamurthy, B., Sen, S., Zhang, Y., and Chen, Y. “Sketch-Based Change Detection: Methods, Evaluation, and Applications.” *Proc. of ACM SIGCOMM Internet Measurement Conference* (2003): 234–247.

[75] Kubiatowicz, J., Bindel, D., Chen, Y., et al. “Oceanstore: An Architecture for Global-Scale Persistent Storage.” *ACM Sigplan Notices* 35 (2000).11: 190–201.

[76] Kumar, A., Xu, J., Wang, J., Spatschek, O., and Li, L. “Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement.” *IEEE Journal on Selected Areas in Communications* 24 (2006).12: 2327–2339.

[77] Kumar, A., Xu, J., and Zegura, E.W. “Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks.” *Proc. of IEEE INFOCOM* 2 (2005): 1162–1173.

[78] Lee, S., Joo, S., and Lee, C. “An Enhanced Dynamic Framed Slotted ALOHA Algorithm for RFID Tag Identification.” *Proc. of IEEE MOBIQUITOUS* (2005): 166–172.

[79] Lee, T., Kim, K., and Kim, H. “Join Processing using Bloom Filter in MapReduce.” *Proc. of ACM Research in Applied Computation Symposium* (2012): 100–105.

[80] Leow, W., Ni, D., and Pishro-Nik, H. “A Sampling Theorem Approach to Traffic Sensor Optimization.” *IEEE Transactions on Intelligent Transportation Systems* 9 (2008).2: 369–374.

[81] Leshem, A. and Tong, L. “Estimating Sensor Population via Probabilistic Sequential Polling.” *IEEE Signal Processing Letters* 12 (2005).5: 395–398.

[82] Li, F., Cao, P., Almeida, J., and Broder, A. "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol." *IEEE/ACM Transactions on Networking* 8 (2000).3: 281–293.

[83] Li, T., Chen, S., and Ling, Y. "Identifying the Missing Tags in a Large RFID System." *Proc. of ACM Mobihoc* (2010): 1–10.

[84] ———. "Fast and Compact Per-Flow Traffic Measurement through Randomized Counter Sharing." *IEEE INFOCOM* (2011): 1799–1807.

[85] ———. "Efficient Protocols for Identifying the Missing Tags in a Large RFID System." *to appear in IEEE/ACM Transactions on Networking* (2014).

[86] Li, T., Chen, S., and Qiao, Y. "Origin-destination Flow measurement in High-speed Networks." *Proc. of INFOCOM* (2012): 2526–2530.

[87] Li, T., Luo, W., Mo, Z., and Chen, S. "Privacy-preserving RFID Authentication Based on Cryptographical Encoding." *Proc. of IEEE INFOCOM* (2012): 2174–2182.

[88] Li, T., Wu, S., Chen, S., and Yang, M. "Energy Efficient Algorithms for the RFID Estimation Problem." *Proc. IEEE INFOCOM* (2010): 1–9.

[89] ———. "Generalized Energy-Efficient Algorithms for the RFID Estimation Problem." *IEEE/ACM Transactions on Networking* 20 (2012).6: 1978–1990.

[90] Li, X., Bian, F., Crovella, M., Diot, C., Govindan, R., Iannaccone, G., and Lakhina, A. "Detection and Identification of Network Anomalies Using Sketch Subspaces." *Proc. of ACM SIGCOMM Internet Measurement Conference* (2006): 147–152.

[91] Liang, W. "Constructing Minimum-energy Broadcast Trees in Wireless Ad Hoc Networks." *Proc. of ACM MobiHoc* (2002): 112–122.

[92] Lim, H., Lee, N., Lee, J., and Yim, C. "Reducing False Positives of a Bloom Filter using Cross-Checking Bloom Filters." *Appl. Math* 8 (2014).4: 1865–1877.

[93] Lovett, S. and Porat, E. "A Lower Bound for Dynamic Approximate Membership Data Structures." *Proc. of Foundations of Computer Science (FOCS)* (2010): 797–804.

[94] Lu, G., Debnath, B., and Du, D. "A Forest-structured Bloom Filter with flash Memory." *IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (2011): 1–6.

[95] Lu, L., Liu, Y., and Li, X. "Refresh: Weak Privacy Model for RFID Systems." *Proc. of IEEE INFOCOM* (2010): 1–9.

[96] Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., and Kabbani, A. “Counter Braids: A Novel Counter Architecture for Per-Flow Measurement.” *Proc. of ACM SIGMETRICS* 36 (2008).1: 121–132.

[97] Lu, Y. and Prabhakar, B. “Robust Counting Via Counter Braids: An Error-Resilient Network Measurement Architecture.” *Proc. of IEEE INFOCOM* (2009): 522–530.

[98] Lu, Y., Prabhakar, B., and Bonomi, F. “Bloom Filters: Design Innovations and Novel Applications.” *Proc. of Allerton Conference* (2005).

[99] Lumetta, S. and Mitzenmacher, M. “Using the Power of Two Choices to Improve Bloom Filters.” *Internet Mathematics* 4 (2007).1: 17–33.

[100] Luo, W., Chen, S., Li, T., and Chen, S. “Efficient Missing Tag Detection in RFID Systems.” *Proc. of IEEE INFOCOM* (2011): 356–360.

[101] Luo, W., Chen, S., Li, T., and Qiao, Y. “Probabilistic Missing-Tag Detection and Energy-Time Tradeoff In Large-Scale RFID Systems.” *Proc. of Mobile Ad Hoc Networking and Computing (MobiHoc)* (2012): 95–104.

[102] Luo, W., Chen, S., Qiao, Y., and Li, T. “Missing-Tag Detection and Energy–Time Tradeoff in Large-Scale RFID Systems With Unreliable Channels.” *to appear in IEEE/ACM Transactions on Networking* (2014).

[103] Maccari, L., Fantacci, R., Neira, P., and Gasca, R. “Mesh Network Firewalling with Bloom Filters.” *IEEE International Conference on Communications* (2007): 1546–1551.

[104] Mackert, L. and Lohman, G. “R* Optimizer Validation and Performance Evaluation for Local Queries.” *Proc. of ACM SIGMOD international conference on Management of data* 15 (1986).2.

[105] Mahiourian, R., Chen, F., Tiwari, R., Thai, M.T., Zhai, H., and Fang, Y. “An Approximation Algorithm for Conflict-Aware Broadcast Scheduling in Wireless Ad Hoc Networks.” *Proc. of ACM MobiHoc* (2008): 331–340.

[106] Malde, K. and OSullivan, B. “Using Bloom Filters for Large Scale Gene Sequence Analysis in Haskell.” *Practical Aspects of Declarative Languages* (2009): 183–194.

[107] Manna, P. K., Chen, S., and Ranka, S. “Inside the Permutation-Scanning Worms: Propagation Modeling and Analysis.” *IEEE/ACM Transactions on Networking* 18 (2010).3: 858–870.

[108] Massoulie, L., Merrer, E. Le, Kermarrec, A., and Ganesh, A. “Peer Counting and Sampling in Overlay Networks: Random Walk Methods.” *Proc. of ACM Symposium on Principles of Distributed Computing* (2006): 123–132.

[109] Michael, L., Nejdl, W., Papapetrou, O., and Siberski, W. “Improving distributed join efficiency with extended bloom filter operations.” *International Conference on Advanced Information Networking and Applications (AINA)* (2007): 187–194.

[110] Mittag, J., Schmidt-Eisenlohr, F., Killat, M., Härri, J., and Hartenstein, H. “Analysis and Design of Effective and Low-overhead Transmission Power Control for VANETs.” *Proc. of ACM international workshop on VehiculAr Inter-NETworking* (2008): 39–48.

[111] Mitzenmacher, M. “Compressed Bloom Filters.” *IEEE/ACM Transactions on Networking* 10 (2002).5: 604–612.

[112] Miura, M., Ito, S., Takatsuka, R., Sugihara, T., and Kunifuji, S. “An Empirical Study of an RFID Mat Sensor System in a Group Home.” *Journal of Networks* 4 (2009).2.

[113] Moffat, A., Neal, R., and Witten, I. “Arithmetic Coding Revisited.” *ACM Transactions on Information Systems (TOIS)* 16 (1998).3: 256–294.

[114] Mullin, J.K. “Optimal Semijoins for Distributed Database Systems.” *IEEE Transactions on Software Engineering* 16 (1990).5: 558–560.

[115] Myung, J. and Lee, W. “Adaptive Splitting Protocols for RFID Tag Collision Arbitration.” *Proc. of ACM MOBIHOC* (2006): 202–213.

[116] Namboodiri, V. and Gao, L. “Energy-Aware Tag Anti-collision Protocols for RFID Systems.” *IEEE Transactions on Mobile Computing* 9 (2010).1: 44–59.

[117] Nath, S., Gibbons, P.B., Seshan, S., and Anderson, Z.R. “Synopsis Diffusion for Robust Aggregation in Sensor Networks.” *Proc. of SenSys* (2004): 250–262.

[118] Navidi, W. and Camp, T. “Stationary Distributions for the Random Waypoint Mobility Model.” *IEEE Transactions on Mobile Computing* 3 (2004).1: 99–108.

[119] Ni, L. M., Liu, Y., Lau, Y. C., and Patil, A. “LANDMARC: Indoor Location Sensing using Active RFID.” *ACM Wireless Networks (WINET)* 10 (2004).6: 701–710.

[120] Noureddine, H. and Samira, M. “Efficient Local Density Estimation Strategy for VANETs.” *arXiv preprint arXiv:1402.4508* (2014).

[121] Pagh, A., Pagh, R., and Rao, S.S. “An Optimal Bloom Filter Replacement.” *Proc. of ACM-SIAM symposium on Discrete algorithms* (2005): 823–829.

[122] Pagiamtzis, K. and Sheikholeslami, A. “Content-addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey.” *IEEE Journal of Solid-State Circuits* 41 (2006).3: 712–727.

[123] Panichpapiboon, S. and Pattara-atikom, W. “Evaluation of a Neighbor-Based Vehicle Density Estimation Scheme.” *ITS Telecommunications* (2008): 294–298.

[124] Pearson, M. “QDRTM-III: Next Generation SRAM for Networking.” *http://www.qdrconsortium. org/presentation/QDR-III-SRAM.pdf* (2009).

[125] Perkins, C. and Bhagwat, P. “Highly Dynamic Destination-Sequenced Distance Vector Routing (DSDV) for Mobile Computers.” *ACM SIGCOMM Computer Communication Review* 24 (1994).4: 234–244.

[126] Perkins, C. E. and Royer, E. M. “Ad Hoc On-demand Distance Vector Routing.” *Proc. of IEEE Workshop on Mobile Computing Systems and Applications* (1999): 90–100.

[127] Phan, T., d’Orazio, L., and Rigaux, P. “Toward Intersection Filter-based Optimization for Joins in MapReduce.” *Proc. of International Workshop on Cloud Intelligence* (2013): 2.

[128] Porat, E. “An Optimal Bloom Filter Replacement Based on Matrix Solving.” *Computer Science-Theory and Applications* (2009): 263–273.

[129] Qian, C., Ngan, H., Liu, Y., and Ni, L. “Cardinality Estimation for Large-scale RFID Systems.” *IEEE Transactions on Parallel and Distributed Systems* 22 (2011).9: 1441–1454.

[130] Qiao, Y., Chen, S., and Li, T. *RFID as an Infrastructure*. Springer, 2012.

[] Qiao, Y., Chen, S., Li, T., and Chen, S. “Energy-efficient Polling Protocols in RFID Systems.” *Proc. of ACM MobiHoc* (2011).

[] Qiao, Y., Li, T., and Chen, S. “One Memory Access Bloom Filters and Their Generalization.” *Proc. of IEEE INFOCOM* (2011): 1745 – 1753.

[] ———. “Fast Bloom Filters and Their Generalization.” *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25 (2014).1: 93 – 103.

[131] Ramesh, S., Papapetrou, O., and Siberski, W. “Optimizing Distributed Joins with Bloom Filters.” *Distributed Computing and Internet Technology* (2009): 145–156.

[132] Reynolds, P. and Vahdat, A. “Efficient Peer-to-peer Keyword Searching.” *Proc. of the ACM/IFIP/USENIX International Conference on Middleware* (2003): 21–40.

[133] Rothenberg, C., Macapuna, C., Verdi, F., and Magalhaes, M. “The Deletable Bloom Filter: a New Member of the Bloom Family.” *IEEE Communications Letters* 14 (2010).6.

[134] Rottenstreich, O., Kanizo, Y., and Keslassy, I. “The Variable-Increment Counting Bloom Filter.” *Proc. of IEEE INFOCOM* (2012): 1880–1888.

[135] Royer, E. M. and Toh, C. “A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks.” *IEEE Personal Communications* 6 (1999).2: 46–55.

[136] Ruhanen, A., Hanhikorpi, M., Bertuccelli, F., Colonna, A., Malik, W., Ranasinghe, D., Lopez, T. S., Yan, N., and Tavilampi, M. “Sensor-enabled RFID Tag Handbook.” *BRIDGE, IST-2005-033546* (2008).

[137] Sarangan, V., Devarapalli, M. R., and Radhakrishnan, S. “A Framework for Fast RFID Tag Reading in Static and Mobile Environments.” *The International Journal of Computer and Telecommunications Networking* 52 (2008).5.

[138] Semiconductors, Philips. “I-CODE Smart Label RFID Tags.” *http://www.nxp.com/documents/data_sheet/ SL092030.pdf* (2004).

[139] Shen, H., Li, Z., Yu, L., and Qiu, C. “Efficient Data Collection for Large-Scale Mobile Monitoring Applications.” *IEEE Transactions on Parallel and Distributed Systems (TPDS)* PP (2013).99.

[140] Sheng, B., Li, Q., and Mao, W. “Efficient Continuous Scanning in RFID Systems.” *Proc. of IEEE INFOCOM* (2010): 1–9.

[141] Shirani, R., Hendessi, F., and Gulliver, T.A. “Store-Carry-Forward Message Dissemination in Vehicular Ad-Hoc Networks with Local Density Estimation.” *Vehicular Technology Conference* (2009): 1–6.

[142] Song, H., Dharmapurikar, S., Turner, J., and Lockwood, J. “Fast Hash Table Lookup using Extended Bloom Filter: An Aid to Network Processing.” *ACM SIGCOMM Computer Communication Review* 35 (2005).4: 181–192.

[143] Song, H., Hao, F., Kodialam, M., and Lakshman, T. “IPv6 Lookups Using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards.” *Proc. of IEEE INFOCOM* (2009).

[144] Spitzner, L. “The Honeynet Project: Trapping The Hackers.” *IEEE Security & Privacy* 1 (2003).2: 15–23.

[145] Staniford, S., Hoagland, J., and McAlerney, J. “Practical Automated Detection of Stealthy Portscans.” *Journal of Computer Security* 10 (2002): 105 – 136.

[146] Sun, C. and Ritchie, S. “Individual Vehicle Speed Estimation Using Single Loop Inductive Waveforms.” *Journal of Transportation Engineering* 125 (1999).6: 531–538.

[147] Suresh, D., Guo, Z., Buyukkurt, B., and Najjar, W. “Automatic Compilation Framework for Bloom Filter Based Intrusion Detection.” *Reconfigurable Computing: Architectures and Applications* 3985 (2006): 413–418.

[148] Tabataba, F. and Hashemi, M. “Improving False Positive in Bloom Filter.” *Electrical Engineering (ICEE)* (2011): 1–5.

[149] Tan, C., Sheng, B., and Li, Q. “How to Monitor for Missing RFID Tags.” *Proc. of IEEE ICDCS* (2008): 295–302.

[150] Tarkoma, S., Rothenberg, C., and Lagerspetz, E. “Theory and Practice of Bloom Filters for Distributed Systems.” *Communications Surveys & Tutorials, IEEE* (2012).99: 1–25.

[151] Toh, C. “Future Application Scenarios For Manet-Based Intelligent Transportation Systems.” *Future Generation Communication And Networking* 2 (2007): 414–417.

[152] Torrent-Moreno, M., Mittag, J., Santi, P., and Hartenstein, H. “Vehicle-To-Vehicle Communication: Fair Transmit Power Control for Safety-Critical Information.” *IEEE Transactions on Vehicular Technology* 58 (2009).7: 3684–3703.

[153] Umer, T., Ding, Z., Honary, B., and Ahmad, H. “Implementation of Microscopic Parameters for Density Estimation of Heterogeneous Traffic Flow for VANET.” *Communication Systems Networks and Digital Signal Processing (CSNDSP)* (2010): 66–70.

[154] Vogt, H. “Efficient Object Identification with Passive RFID Tags.” *Proc. of IEEE PerCom* (2002): 98–113.

[155] Wang, W., Jiang, H., Lu, H., and Yu, J.X. “Bloom Histogram: Path Selectivity Estimation for XML Data with Updates.” *Proc. of Very Large Data Bases (VLDB)* (2004): 240–251.

[156] Witten, I., Moffat, A., and Bell, T. *Managing Gigabytes: Compressing And Indexing Documents and Images (2nd Edition)*. Morgan Kaufmann, 1999.

[157] Wolfson, O., Xu, B., Yin, H., and Cao, H. “Search-and-Discover in Mobile P2P Network Databases.” *Proc. of IEEE ICDCS* (2006): 65–65.

[158] Xiao, B. and Hua, Y. “Using Parallel Bloom filters for Multi-attribute Representation on Network Services.” *IEEE Transactions on Parallel and Distributed Systems* 21 (2010).1: 20–32.

[159] Xiao, Q., Xiao, B., and Chen, S. “Differential Estimation in Dynamic RFID Systems.” *Proc. of IEEE INFOCOM* (2013): 295–299.

[160] Xu, B. and Wolfson, O. “Data Management in Mobile Peer-to-peer Networks.” *International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)* (2005): 1–15.

[161] Yang, L., Han, J., Qi, Y., and Liu, Y. “Identification-free Batch Authentication for RFID Tags.” *Proc. of IEEE ICNP* (2010): 154–163.

[162] Yoon, M. “Aging Bloom Filter with Two Active Buffers for Dynamic Sets.” *IEEE Transactions on Knowledge and Data Engineering* 22 (2010).1: 134–138.

[163] Yoon, M., Li, T., Chen, S., and Peir, J. “Fit a Spread Estimator in Small Memory.” *Proc. of IEEE INFOCOM* (2009): 504–512.

[164] Yuan, Z., Miao, J., Jia, Y., and Wang, L. "Counting Data Stream Based on Improved Counting Bloom filter." *The 9th International Conference on Web-Age Information Management (WAIM)* (2008): 512–519.

[165] Yue, H., Zhang, C., Pan, M., Fang, Y., and Chen, S. "A Time-efficient Information Collection Protocol for Large-scale RFID Systems." *IEEE Proc. of INFOCOM* (2012): 2158–2166.

[166] ———. "Unknown-Target Information Collection in Sensor-Enabled RFID Systems." *to appear in IEEE/ACM Transactions on Networking* (2014).

[167] Zhai, J. and Wang, G. N. "An Anti-collision Algorithm using Two-functioned Estimation for RFID Tags." *Proc. of ICCSA* (2005): 702–711.

[168] Zhang, F., Wu, D., Ao, N., Wang, G., Liu, X., and Liu, J. "Fast Lists Intersection with Bloom Filter using Graphics Processing Units." *Proc. of ACM Symposium on Applied Computing* (2011): 825–826.

[169] Zhang, M., Li, T., Chen, S., and Li, B. "Using Analog Network Coding to Improve the RFID Reading Throughput." *International Conference on Distributed Computing Systems (ICDCS)* (2010): 547–556.

[170] Zhen, B., Kobayashi, M., and Shimizu, M. "Framed ALOHA for Multiple RFID Objects Identification." *IEICE Transactions on Communications* 88 (2005).3: 991–999.

[171] Zheng, Y. and Li, M. "Fast Tag Searching Protocol for Large-scale RFID Systems." *IEEE International Conference on Network Protocols (ICNP)* (2011): 363–372.

[172] ———. "ZOE: Fast Cardinality Estimation for Large-Scale RFID Systems." *Proc. of IEEE INFOCOM* (2013): 908–916.

[173] Zheng, Y., Li, M., and Qian, C. "PET: Probabilistic Estimating Tree for Large-scale RFID Estimation." *International Conference on Distributed Computing Systems (ICDCS)* (2011): 37–46.

[174] Zhou, F., Chen, C., Jin, D., Huang, C., and Min, H. "Evaluating and Optimizing Power Consumption of Anti-collision Protocols for Applications in RFID Systems." *Proc. of ISLPED* (2004): 357–362.

[175] Zhu, Y., Jiang, H., and Wang, J. "Hierarchical Bloom Filter Arrays (HBA): a Novel, Scalable Metadata Management System for Large Cluster-based Storage." *IEEE International Conference on Cluster Computing* (2004): 165–174.

BIOGRAPHICAL SKETCH

Yan Qiao received her Ph.D. in computer engineering from the University of Florida in the spring of 2014. She received her B.S. degree in computer science and technology in Shanghai Jiao Tong University, China in 2009. Her research interests include network measurement, network algorithms and RFID protocols.