Design and Implementation of a High-Speed RFID Data Filtering Engine★

Hyunsung Park, Jongdeok Kim

Dept. of Computer Science and Engineering, Pusan National University.
Geumjeong-gu, Busan 609-735, Korea
jesse@pusan.ac.kr, kimjd@pusan.ac.kr

Abstract. In this paper, we present a high-speed RFID data filtering engine designed to carry out filtering under the conditions of massive data and massive filters. We discovered that the high-speed RFID data filtering technique is very similar to the high-speed packet classification technique which is used in high-speed routers and firewall systems. Actually, our filtering engine is designed based on existing packet classification algorithms, Bit-Parallelism and Aggregated Bit Vector (ABV). In addition, we also discovered that there are strong temporal relations and redundancy in the RFID data filtering operations. We incorporated two kinds of caches, tag and filter caches, to make use of this characteristic to improve the efficiency of the filtering engine. The performance of the proposed engine has been examined by implementing a prototype system and testing it. Compared to the basic sequential filter comparison approach, our engine shows much better performance, and it gets better as the number of filters increases.

1. Introduction

RFID is an automatic identification system, which consists of tags attached to target objects and networked readers recognizing those tags. The RFID system does not require line of sight or contact between readers and tags and it is fast as well. With the significant advantages of RFID technology, RFID is being gradually adopted and deployed in a wide area of applications, including supply chain management, retail, anti-counterfeiting, and healthcare. For example, global retail giants, such as Wal-Mart and Tesco, are now pushing their suppliers to integrate the RFID technology into their supply chain. As RFID systems enable us to achieve many good things, such as more efficient inventory managing, greater visibility, easier product tracking and monitoring, we expect that nationwide, even globalwide RFID infrastructure would be built in the near future.

However, the amount of tag data to handle in real-time would get heavier as RFID systems get more widely deployed and used. Note that an RFID application concerns only a certain subset of the total captured data. Considering the limited computing and communication resources of RFID systems, delivering all the captured data to all the

★This work was supported by the Regional Research Centers Program (Research Center for Logistics Information Technology), granted by the Korean Ministry of Education & Human Resources Development.

applications above is neither proper nor feasible. It is inevitable to filter our unwanted data and deliver only wanted data to appropriate applications. As a result, RFID data filtering has emerged one of key challenges in RFID technology and it is now getting more attention. Leading RFID standardization bodies like EPCglobal have defined some basic specifications about RFID data filtering and some major RFID middleware providers incorporate filtering functions into their products.

EPCglobal, which is an industry consortium leading the development of industry-driven standards for the Electronic Product Code™ (EPC) to support the use of Radio Frequency Identification (RFID), has released "EPCglobal Reader Protocol" [1]. It describes filtering functions, such as ReadFilter, that can be carried out by a conforming RFID reader. The ReadFilter removes certain tag read events according to the bit-wise patterns negotiated through the reader protocol. Multiple filters can be used and each filter is specified to be either inclusive, meaning that only tags matching the filter shall be reported, or exclusive, meaning that a tag shall only be reported if it does not match the filter. While basic RFID filter specifications are addressed, how to carry out filtering function for massive data and massive filters are not addressed in related documents. Besides, we can hardly find any previous academic research on this massive RFID filtering problem in spite of its importance.

2. RFID Data filtering and Bit-Parallelism

In this section, we introduce the filtering operation defined by EPCglobal. Then we explain the similarity between packet classification and RFID data filtering.

2.1 RFID Data Filtering

RFID data filtering consists of simple logical operations based on bit-wise patterns. A filter is specified by using two hexadecimal strings, a filter value 'V' and a filter mask 'M'. In the filter mask (M), all bit positions where the value is important for the filtering are set to 1. In the filter value (V), the desired value of the bit positions defined as relevant in M can be set. A tag ID matches the filter if and only if the result of applying the filter mask on the filter value using a bit-wise AND operation is the same as when applying the filter mask on the tag ID:

If $(V\ bitand\ M) == (A\ bitand\ M)$ then $TagIDMatchesTheFilter()$

Example:
filter mask M = 1C (00011100),
filter value V = 10 (00010000)
Because of the setting of M, only the values of bit positions 4-6 are important.

Actual tag ID data A = 55 (01010101 in binary)
V bitand M = 10 bitand 1C = 00010000 bitand 00011100 = 00010000 = 10
A bitand M = 55 bitand 1C = 01010101 bitand 00011100 = 00010100 = 14

The two values are different, so there is no match.

2.2 Packet Classification vs. RFID Data Filtering

An example of a real-life packet classifier in four dimensions is shown in Table 1. Table 2 shows an example of a GID-96 tag's RFID data filters which are compliant with Tag Data Structure (TDS) [2]. They are recorded with hexadecimal numbers.

Table 1. An example of Packet Classification

Rule	Destination	Source	Port	Protocol	Action
R1	164.125.34.164 255.255.255.0	164.125.34.164 255.255.255.255	eq www	*	Deny
R2	164.125.34.164 255.255.0.0	164.125.34.164 255.255.252.0	*	udp	Accept

Table 2. An example of RFID data filters

Filter		Header	General Manager Number	Object Class	Serial Number
F1	V	35	4AB8012	856001	9112ABC90
	M	FF	FFFFC00	FFE00	FFFC00000
F2	V	35	2300890	AB6001	0012134AF
	M	FF	FF80000	FFFFC0	FFFE00000

Let's examine these two examples on the point of bit-wise pattern filtering. Even though there is no action column in Table 2, we can say that packet classification and RFID data filtering have the same way of pattern filtering. Instead of an action column, an RFID data filter is specified to be either an inclusive or exclusive filter. "Inclusive" means that only tags matching the filter shall be reported whereas "exclusive" means that a tag shall only be reported if it does not match the filter (i.e., the tag does not match any of the exclusive patterns and matches at least one of the inclusive patterns). Therefore, we conclude that packet classification and RFID data filtering are very similar. Thus, the major concern of this paper is applying packet classification algorithms to RFID data filtering. After studying packet classification algorithms [3, 4, and 5] in previous studies, we applied them to RFID data filtering.

2.3 Bit Vector Generation from Bit-Parallelism

The Bit-Parallelism scheme is a method of divide-and-conquer which divides the whole bit stream into k dimensions and then combines the results. Table 3 is a simple example of this. Note that the wild card (*) performs the same role of filter mask in RFID data filtering.

Table 3. A simple example with 8 filters on two dimensions

Filter	D1	D2
F1	00*	11*
F2	10*	11*
F3	11*	10*
F4	0*	10*
F5	0*	01*
F6	0*	0*
F7	00*	00*
F8	1*	0*

Fig. 1 illustrates the trie construction of simple two dimensional example databases in Table 3. In Fig. 1, from the center point on the top, prefixes slide down to the left by bit 0 or to the right by bit 1 and set its bit position of the bit vector in the destination node. For example, the first filter F1 in Table 3 has 00* in the first dimension (D1); thus, the leftmost node in the trie corresponds to 00* (i.e., slides down to left and left and set 1st bit). By the same procedure, the D1 trie contains a node for all distinct prefixes in D1 of Table 3 such as 00*, 10*, 11*, 0*, and 1*. Note that the * affects the lower layers; for example, 1* of F8 in D1 slides down right and sets the 8th bit. Also, its right and left sub node's 8th bit have to be set (i.e., a recursive procedure). As a result, each node in the trie is labeled with an N-bit long vector (i.e., N is the total number of filters).

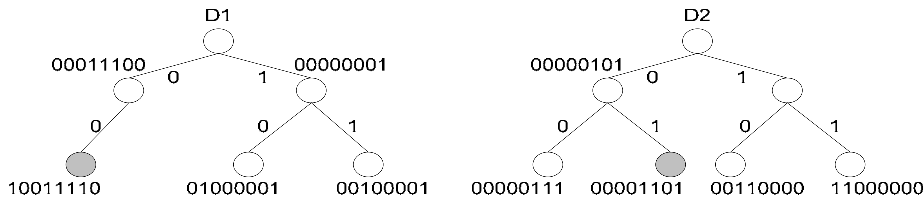

Fig. 1. Filtering by Bit-Parallelism Trie

Using two constructed tries in Fig. 1, let's examine an actual tag ID filtering. When a tag ID arrives with dimensions $D_1, ..., D_k$, we do a longest matching prefix lookup. For a simple example, tag ID of 8 bits long 00100111 is read in the reader. It consists of two dimensions; D1 having 4 bits and D2 having 4 bits. Those two 4 bit vectors slide down Fig. 1's tries, respectively. Grey circles are the final bit vector destinations of each dimension as a result of sliding down by bit 0 to the left and bit 1 to the right manner. Then we calculate the bit-wise AND operation with these two bit vectors:

10011110 & 00001101 = 00001100 = {F5, F6}

As a result, both F5 and F6 are matched filters for the actual tag ID (00100111). If we take priority into consideration, F5 has higher priority than F6 by convention. However, in this paper, we restrict RFID data filters to inclusive or exclusive without priority. Therefore, if the tag does not match any of the exclusive filters and matches at least one of the inclusive filters, it shall be reported to an upper layer.

2.4 Applying ABV

The ABV (Aggregated Bit Vector) [5] is made by simply aggregating each group of A bits (A is aggregate size) into a single bit (which represents the bit-wise OR operation). In this way, the main goal of an ABV algorithm is reducing the number of bit-wise operations and memory access. In order to do this, there is one big condition that the set bits in the bit vector of each node must be very sparse (i.e., with very few '1' bits in the whole bit vector, the others being '0' bits).

Given the above, by examining the aggregate bit vector with large N, we only examine the leaf bit map values for which the aggregate bits are set. In other words, the aggregate vectors allow us to quickly filter out bit positions where there is no match. The goal is to have a scheme that comes close to taking $O(\log_A N)$ memory

accesses instead of taking $O(N)$, even for large N. Therefore, it is important to decide the A value. A is a constant that can be tuned to optimize the performance of the aggregate scheme; a convenient value for A is W (the word size).

Fig. 2 illustrates the trie construction of ABV for the example database in Table 3. The parentheses are aggregated bit vectors using an aggregate size A = 3. So, the leftmost leaf node (10011110) produces (111) by OR operation (aggregated with 3 bits, 3 bits and the last 2 bits in order).

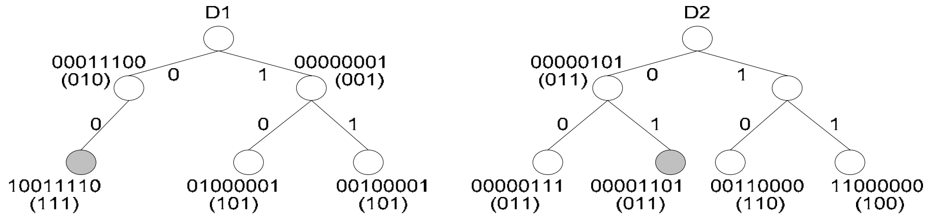

Fig. 2. Filtering by ABV Trie

For example, with the same actual tag ID in the previous example, 00100111 is read in the reader. Like the same procedure as in the previous example, grey circles are the final bit vector destinations of each dimension. Then, we can do the bit-wise AND operation with two aggregated bit vectors. Thus,

ABV (111) & ABV (011) = 011.

The AND operation on the two aggregate vectors yields 011, showing that a possible matching rule must be located only in the last 5 bits (3 bits plus 2 bits). Thus, it is not necessary to retrieve the first 3 bits for each field. Notice that the cost savings are small in this example, but in larger examples they show much bigger gains.

However, note that the ABV algorithm may wrongly assume that there might be a matching rule in the corresponding bit positions for all the set bits in the aggregate bit vector. This is because of a false match [5], a situation in which the result of an AND operation on an aggregate bit returns a set bit but there is no valid match in the real bit vector.

3. Cache Based Improvement

In this section, we integrate cache to RFID data filtering to achieve an additional improvement. Also, we describe their implementation.

3.1 Background

We suggest a cache concept to RFID data filtering because we believe they have the following characteristics, especially regarding the RFID system. We assume these two characteristics as the background of our cache-based improvement:

- The probability of the same tag detection is very high in the short period of time.
- The probability that the matched filter will be rematched is very high during the specified time.

As a simple example of the first assumption, let's think about the big warehouse. We assume there are RFID readers which read more than 1000 tags of goods per second. The goods which lay currently within the reader's RF field are detected according to the sensing interval. Therefore, until those goods are moved beyond the RFID reader's RF coverage, the same tag IDs are detected continuously. As a simple example of the second assumption, we assume a situation of putting goods into the warehouse or taking goods out of the warehouse. Usually, the same kinds of goods are delivered at the same time, which means only their serial numbers are different (i.e., often the serial number is increased by one). Therefore, the probability that a matched filter will be rematched is very high during the specified time.

In summary, all processes based on the above assumptions are illustrated with the flow diagram in Fig.3. First, one sensed tag ID goes through a tag cache. If the tag cache contains that tag ID, the tag cache is able to know whether it was a matched filter or not by extracting its value. If the tag ID doesn't exist in the tag cache (i.e., new tag ID), the process moves to the filter cache. In the filter cache, filters of the highest probability should be applied first. If there is no matched filter in the filter cache, the filtering process is performed last.

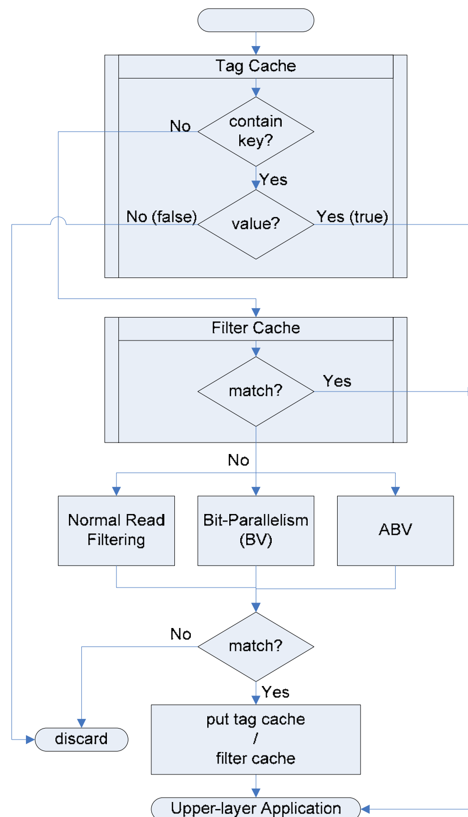

Fig. 3. Diagram of filtering flow including tag cache and filter cache.

3.2 Using a Tag Cache

As an implementation of the tag cache, we have decided to use a combination of a hash table and a hash map with a linked list. As a hash function, we simply performed parity checking. We counted how many set bit exists in each bit vector. For example, Table 4 shows the parity checking of the source data for a tag cache. The first tag ID (00100011) can be divided into three bit vectors; 3-3-2 (001-000-11). The first bit vector (001) has an odd number of bit 1, so the parity value is '1'. Whereas the last bit vector (11) has even number of bit 1, the parity value is '0'. Thus, we can get a parity bit vector (100). With the parity column of Table 4, we classified them into the hash buckets of Table 5. This will evenly distribute tag IDs into the hash buckets.

In Table 5, we illustrated that each hash buckets refer to a linked list of records as a collision resolution strategy of a hash table. Furthermore, we gave each linked list of records a limit to store (i.e., a queue). Therefore, a bucket which overflows the prefixed queue size should delete its older entry; i.e., the Least Recently Used (LRU) cache. This is true because older data has a lower probability of being sensed again than newly sensed tags (with RFID characteristics we assumed in the background section of this chapter). In our implementation, we decided to use the doubly linked list as a queue. Determining the queue size is another important factor in creating better filtering performance.

Table 4. Source data for tag cache

No.	Tag IDs	Match?	Parity
1	00100011	T	100
2	11110011	T	110
3	10101100	T	000
4	10111011	F	000
5	00011111	T	010
6	10001101	T	101
7	11101011	T	110
8	01010111	T	100
9	10011001	F	101
10	10100101	T	011

Table 5. Tag Cache using a hash table

Hash Buckets	key (value)
000	10101100(T)→10111011(F)
001	-
010	00011111(T)
011	10100101(T)
100	00100011(T)→01010111(T)
101	10001101(T)→10011001(F)
110	11110011(T)→11101011(T)
111	-

We consider a duplication problem in Table 5, because duplication makes the cache even useless. Actually, to prevent duplicated tag IDs from appending into the queue of each bucket, we used a LinkedHashMap Class of Java API which implements Map interface. Map interface is an object that maps keys to values. A map cannot contain duplicate keys (i.e., tag ID); each key can map to at most one value [6]. Therefore, we didn't need to worry about duplication, but had to consider the operation overhead of the LinkedHashMap class in the Java API. In fact, we put off the specific verification of this to future.

Note that Map interface enables us to map keys to values. Therefore, stored key contains a boolean value which represents that true is matched tag and false is not matched tag. Thus, when a tag ID enters a tag cache, if the tag cache contains the tag

ID (i.e., key), the tag cache is able to know whether it was a matched filter or not by extracting its value without any further processes.

3.3 Using a Filter Cache

As a filter cache, we decided to use a single queue. Therefore, a filter cache is not so different from one array of tag caches. But the elements of the filter cache are filters, not tag ID. Further, if the appropriate queue size is applied, a filter cache complements a tag cache nicely, and vice versa.

3.4 Pseudo Codes for ABV with Cache

In this section, we are going to introduce only the ABV based filtering pseudo codes which includes cache related codes. Because the ABV pseudo codes also imply the principle of Bit-Parallelism, you can refer these pseudo codes without aggregation concept to the pure Bit-Parallelism pseudo codes with cache functions [5].

```
1   Get TagID(D_1,..., D_k);
2   HashVal ← HashFunction(TagID);
3   if tagCache[HashVal].containsKey(TagID) then
4     if tagCache[HashVal].getValue(TagID) then
5       postToUpperApp(TagID);
6       return;
7     else
8       return;
9   else if filterCacheFunc(TagID) then
10    postToUpperApp(TagID);
11    return;
12  for i ← 1 to k do
13    N_i ← longestPrefixMatchNode(Trie_i, D_i);
14  Aggregate ← 11...1;
15  for i ← 1 to k do
16    Aggregate ← Aggregate ∩ N_i.aggregate;
17  for i ← 1 to sizeof(Aggregate) − 1 do
18    if Aggregate[i] == 1 then
19      for j ← 0 to A − 1 do
20        if ∩_{l=1}^{k} N_l.bitVect[i × A + j] == 1 then
21          tagCache[HashVal].put(TagID, true);
22          filterCache.put(i × A + j);
23          postToUpperApp(TagID);
24          return;
25  tagCache[HashVal].put(TagID, false);
26  return;
```

4. Evaluation

In this section, we perform the evaluation in two ways. The first method is to measure filtering speed to show better performances when Bit-Parallelism based algorithms are used. The other method is measuring CPU usage which can express indirectly how much the filtering operation burden RFID middleware.

4.1 Preparations

First, we introduce the Virtual Reader which generates RFID pseudo tags synthetically. Note that we need synthetically created pseudo tags and a filter database to test the scalability of our scheme because real-life RFID systems are quite small. We have prepared the following items for the evaluation:

- Virtual Reader: It generates RFID pseudo tags synthetically. In addition, it can store pseudo tags in a file or transfer them to any other host via TCP/IP. We designed this to behave like an Alien RFID Reader [7].
- EdgeManager: Originally, we designed this in order to control various kinds of commercial readers with one particular manager application. For a simulation, we embedded our high-speed filtering algorithm into the EdgeManager. Therefore it is able to perform RFID data filtering, along with Virtual Readers [8].

Given this, we'd like to emphasize that the pseudo tag generation patterns were a very important factor in our evaluation. As a source of our evaluation, tag generation patterns affected the whole algorithm performance. The following are our tag generation patterns and we combined these patterns to imitate real-life tag IDs more closely. As well, we allowed duplicated tags to be generated by the options to satisfy the RFID characteristics we assumed in the previous chapter.

- Random Pattern Generation: The uniform random generation
- Common Prefix Pattern Generation: This pattern makes all the pseudo tags include one of the prearranged prefixes. (e.g., Virtual Reader opens the file which contains common prefixes, and then all the generated pseudo tags have to include one of the common prefixes.) The rest of the bits can be made by Random or Sequential Pattern Generation to complete the whole 96 bits tags length.
- Sequential Pattern Generation: This option enables the pseudo tags to increase their tag IDs by one (i.e., the Serial Number field of the tag is increased by one).

4.2 Filtering Speed Measurement

The following are the evaluation system specifications we have tested:

- CPU & Memory: AMD 1GHz and 1GByte main memory
- OS: Windows XP for EdgeManager, Redhat Linux for Virtual Reader
- Programming language: Java for EdgeManager [6], C for Virtual Reader [9]

In preparation, we created a file which contains 10,000 pseudo tag data generated by the Virtual Reader with options (i.e., a combination of three tag generation

patterns) and they are randomly allowed duplication less than 16 times to satisfy the first major characteristic of our assumption. Afterwards, we measured filtering speed through the EdgeManager in each of the following four cases in Fig. 4:

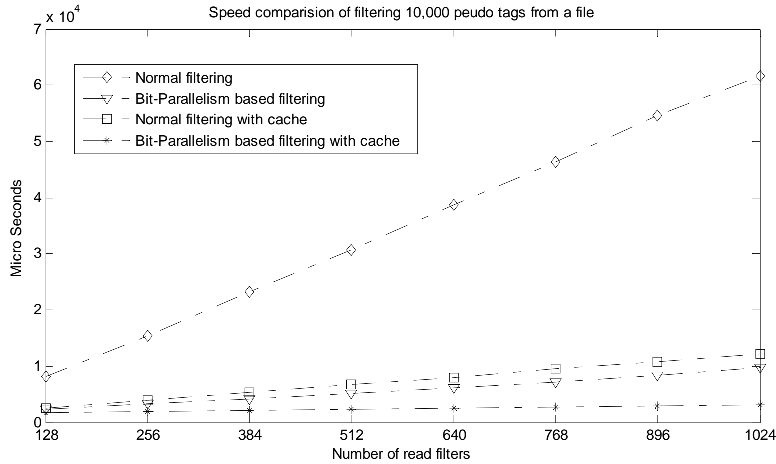

Fig. 4. Speed comparison of filtering 10,000 pseudo tags from a file by increasing 128 filters

As a measuring of filtering time to finish 10,000 pseudo tags from a file, either Bit-Parallelism based or cache based filtering takes much shorter time than normal filtering. However, using both of them make the best performance.

4.3 CPU Usage Measurement

As another way of evaluating our scheme, we measured a CPU usage. We used a CPU usage graph in the Microsoft Windows Operating System (OS) which can show indirectly how much filtering operation burden RFID middleware. We believe this approach is quite reasonable because measuring CPU usage presents quite reliable evaluation data as an OS embedded utility. In addition, Windows OS is very popular so we can approach it very easily.

We used our EdgeManager application again. The difference from the former filtering speed measurement is that the source data is not a file but two Virtual Readers. We let each Virtual Reader send 500 tags per second on average in the real time. As a result, Fig.5 shows the CPU usage of the normal filtering, the Bit-Parallelism based filtering and Bit-Parallelism based filtering with cache from the left. We can verify that Bit-Parallelism based filtering with cache used the least CPU resources.

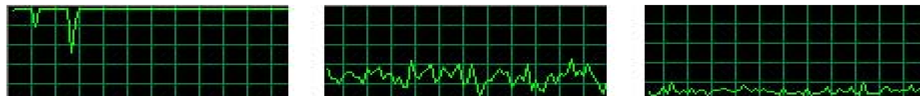

Fig. 5. Filtering comparison graphs in 500 (tags/sec) with 1024 filters

5. Conclusions

Our paper has presented a high-speed RFID data filtering engine designed to carry out filtering under massive data with massive filter conditions for the first time. To do this, we extracted the similarity between the RFID data filtering and packet classification. Then, we designed our filtering engine based on the existing packet classification algorithms, Bit-Parallelism and ABV (Aggregated Bit Vector). Further, we also found that there are RFID data specific characteristics - strong temporal relation and redundancy in the RFID data filtering operations. To make use of these characteristics to improve the efficiency of the filtering engine, we incorporated two kinds of caches, tag and filter caches. We verified our scheme through the synthetically generated filter database and implemented them to the prototype applications – Virtual Reader and EdgeManager. Compared to the normal RFID data filtering, our engine shows much better performance, and it gets better as the number of filters increases.

As an addition to our conclusion, we'd like to emphasize on the need for system customization. In this paper, there are several factors which need customization. They affect the performance of filtering algorithms and caches a great deal. For example, in the case of applying ABV to RFID data filtering, the filters must have sparsely set bits. Also, in the case of applying a cache to a particular RFID system, the applied RFID system must satisfy two major characteristic we assumed in the previous chapters. In addition, other factors like tag generation patterns and cache size also need to be customized to reach optimal performance.

References

1. "EPCglobal Reader Protocol 1.0, Last Call Working Draft Version of 17," March 2005.
2. "EPCglobal Tag Data Standards Version1.3 Standard Specification", September 2005.
3. Pankaj Gupta and Nick McKeown, "Algorithms for Packet Classification", *IEEE Network*, Vol. 15, No. 2, pages 24-32, March-April 2001.
4. T.V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," *Proceedings of ACM SIGCOMM*, pages 191-202, September 1998.
5. Florin Baboescu and George Varghese, "Scalable Packet Classification," *Proceedings of the 2001 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 199-210, 2001, University of California, San Diego.
6. Sun Microsystems, "Java API Specification., Java^{TM} 2 Platform, Standard Edition, v 1.4.2 API Specification," Sun Microsystems, Inc., 2005.
7. Alien Technology, "Reader Interface Guide ALR-9780 ALR-8780 ALR-9640 Doc. Control #8101938-000 Rev D," Alien Technology Corporation, Nov 2004.
8 EPCglobal, "The Application Level Events (ALE) Specification, Version 1.0," February 2005.
9. "The GNU C Library, Edition 0.10, Last Updated 2001-07-06, of The GNU C Library Reference Manual, for Version 2.3.x.," Free Software Foundation, Inc.,