J Inf Process Syst, Vol.11, No.1, pp.95~103, March 2015
http://dx.doi.org/10.3745/JIPS.03.0010

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS

A Memory Efficient Anti-Collision Protocol to Identify Memoryless RFID Tags

Haejae Jung*

Abstract
This paper presents a memory efficient tree based anti-collision protocol to identify memoryless RFID (Radio Frequency Identification) tags that may be attached to products. The proposed deterministic scheme utilizes two bit arrays instead of stack or queue and requires only $\Theta(n)$ space, which is better than the earlier schemes that use at least $O(n^2)$ space, where n is the length of a tag ID in a bit. Also, the size n of each bit array is independent of the number of tags to identify. Our simulation results show that our bit array scheme consumes much less memory space than the earlier schemes utilizing queue or stack.

Keywords
RFID tag identification, anti-collision protocol, data structure, query tree

1. Introduction

The identification of radio frequency identification (RFID) tags is an important technology for the automatic identification of an object by reading its tag ID (identification number) that is attached to it. This technology may have many applications, such as barcode replacement, supply chain, etc. But to replace a barcode, the tag structure should be very simple so that the tag price can be minimized. The authors of [1] proposed a memoryless tag in which a tag responds only based on the current query from the reader as opposed to being based on past queries. In other words, a memoryless tag does not maintain any state from past interactions with the reader. Consequently, the only functionality of a memoryless tag is to receive a query from the reader, match the query with the prefix of its own ID, and reply with its ID to the reader if they match. We also assume that every memoryless tag is passive. By passive, we mean that the memoryless tag does not have its own power. Therefore, it uses the power supplied by the reader to respond the reader's query.

For the memoryless tag, query tree protocols have been published to identify all the tags within the reader's signal range [1-3]. A query tree is a binary tree in which the left/right child is denoted by 0/1, respectively. A query is a bit string that is formed by concatenating binary numbers on the branches of the root through a node in sequence.

We may classify earlier protocols based on a query tree into two schemes—queue and stack schemes.

Manuscript received October 10, 2013; first revision February 11, 2014; accepted March 20, 2014; onlinefirst October 07, 2014.
Corresponding Author: Haejae Jung (hjjung@anu.ac.kr)
* Dept. of Information and Communication Engineering, Andong National University, Andong 760-749, Korea (hjjung@anu.ac.kr)

The queue/stack scheme utilizes a queue/stack to traverse a query tree, respectively.

In the queue scheme, the reader traverses the query tree in level order using a queue [1]. The worst-case space complexity of this scheme is $O(n2^n)$ since the maximum size of each query string in the queue is n bits and the maximum number of queries in the queue is 2^n equal to the number of leaf nodes where n is the size of a tag ID in bit.

In the stack scheme, pre-order traversal is performed using a stack instead of a queue [3-5]. By simply using a stack, the worst-case space complexity is reduced to $O(n^2)$ since the maximum number of queries in the stack is $O(n)$ equal to the height of the query tree and the size of each query is $O(n)$.

This paper proposes a memory efficient anti-collision protocol, which is called a bit array scheme, with only O(n) space complexity to identify memoryless tags and it utilizes two bit arrays whose size is n bits each. The next section describes related works. Section 3 presents the proposed protocol in detail and its space and time complexities are analyzed. Section 4 shows our experimental results and the conclusion is presented in Section 5.

2. Related Works

Suppose we have three tags to identify (listed below) and each tag has its own ID in binary string. For simplicity, it is assumed that the size n of each tag ID is 8 bits long. In reality, the size is much longer (i.e., 96 or 128 bits) [6].

- tag1: 1010 0111
- tag2: 1010 1000
- tag3: 1100 1010

These three IDs can be represented in a query tree, as shown in Fig. 1. In the query tree, the prefix of a tag ID is represented by a sequence of the binary numbers on the links starting from the root to a solid rectangle. During tree traversal, collisions occur at internal nodes, which are denoted by ellipses, and a tag ID is recognized at a leaf node, which is denoted by a solid rectangle.

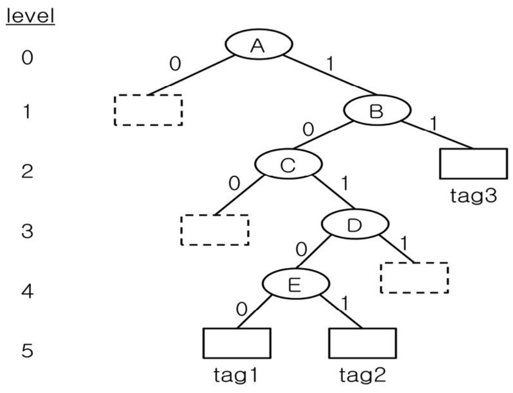

Fig. 1. A query tree consisting of three tags.

We may classify earlier protocols based on a query tree into two schemes—queue and stack schemes. The queue/stack scheme utilizes a queue/stack to traverse a query tree, respectively.

In the queue scheme, the reader traverses the query tree in level order using a queue [1]. To do that, the reader first initializes a queue with two query strings, '0' and '1,' in sequence. Then, the reader repeats the following operations until the queue becomes empty:

- Remove and broadcast the first string *s* from the queue.
- Recognize a tag if only one tag replied (success cycle).
- Do nothing if there is no response from the tags (idle cycle).
- Add two bit strings *s*||0 and *s*||1 to the end of the queue if more than one tag replied (collision cycle). Note that "||" is the concatenation operator of two bit strings here.

The worst-case space complexity of this scheme is $O(n2^n)$ since the maximum size of each query string in the queue is n bits and the maximum number of queries in the queue is 2^n equal to the number of leaf nodes where n is the size of a tag ID in bit.

In the stack scheme, pre-order traversal is performed using a stack instead of a queue [3-5]. In this scheme, the reader pushes s||1 into a stack and broadcasts a new query s||0 when a collision for query *s* occurs. For an idle or success cycle, the reader pops a new query from the stack and broadcasts it. The operation of this stack scheme is done when the stack becomes empty. By using a stack, the worst-case space complexity is reduced to $O(n^2)$ since the maximum number of queries in the stack is $O(n)$, which is equal to the height of the query tree, and the size of each query is $O(n)$.

To minimize collision and idle cycles, the query tree of Fig. 1 can be converted into the compressed query tree of Fig. 2, which is also called a collision tree or compressed binary trie [5-7]. This conversion can simply be done by removing all the internal nodes with only one child, not counting empty leaf nodes represented by dashed rectangles (i.e., nodes C and D of Fig. 1 can be removed). Notice that bit values on the removed links are concatenated into a single bit string, and its first bit denotes a collision position with other tag IDs. In Fig. 2, for example, bit 1 on the link of nodes B and tag3 collides with the first bit "0" of "010" on the link of nodes B and E. For the compressed query tree of this paper, the root is not compressed so that the reader can probe tags starting with 0 and 1.

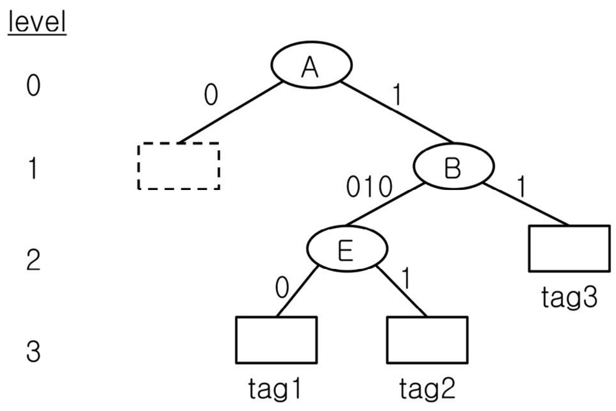

Fig. 2. Compressed query tree of Fig. 1.

In Fig. 2, The IDs of all three tags begin with 1. So, they all are in the right subtree of the root and the left subtree is empty. Tag3 is attached to node B as a right child, since the second bit of tag3 is 1. The rest of the tags are placed in the left subtree of node B as the value of their second bit is 0. The compressed link between node B and node E denotes the common substring "010" of tag1 and tag2. In the compressed query tree, all the tag IDs are placed at the leaves and every internal node, except the root, has both children. This represents a collision among tag IDs. The authors of [5], [6], and [8] proposed efficient protocols using this compressed query tree and showed the efficiency of these protocols via experiments. Their works assume the Manchester code, which can detect the position of the collision from the responses of the tags [9].

3. Proposed Identification Protocol

This section presents an anti-collision protocol with O(n) space complexity to recognize all the tags in the reader's signal range where n is the size of a tag ID in bit. The proposed algorithm uses the compressed query tree and the Manchester code.

First, we assume that each memoryless tag has the following simple functionality, as shown in Algorithm 1. Each tag has its own ID in a bit array myTid[0..n-1]. Notice that array a[s..t] means the bit sequence indexed by s through t, inclusively. A tag receives a prefix pre[0..c] and compares it to the prefix myTid[0..c] of its own ID where $c < n$. If they match, the tag replies with the rest myTid[c+1..n-1] of its own tag ID.

Algorithm 1. Tag side algorithm

```
startTag( ) {
        receive( pre[0..c] );  // receive a prefix c+1 bits
         if( isEqual( pre[0..c], myTid[0..c] )
                  reply(myTid[c+1..n-1] );
}
```

In our bit array scheme, the reader maintains two bit arrays poc[0..n-1] and pre[0..n-1], instead of a queue or stack. The indexes of poc[0..n-1] with bit value '1' denote the positions of collisions. Array pre[0..n-1] contains a query string that may be the prefix of tag IDs. The bit sequence of pre[0..n-1] corresponds to that of myTid[0..n-1]. That is, pre[i] corresponds to myTid[i] where $0 \le i < n$.

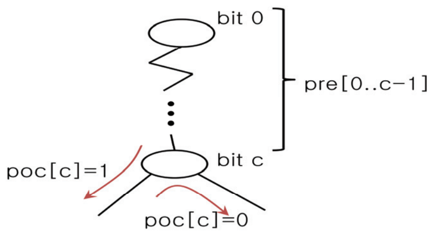

Fig. 3. Tree traversal at collision position c.

The pre-order traversal of a tree can be performed using the two bit arrays. Suppose the reader has broadcasted a query string and found the first collision at bit *c* from the responses of tags, as shown in Fig. 3. The reader sets poc[*c*] = 1 and broadcasts pre[0..*c*-1] || 0 to visit the left subtree of node *c*. After visiting the left subtree, the reader finds *c* that poc[*c*] is equal to 1, resets poc[*c*] and broadcasts pre[0..*c*-1] || 1 to visit the right subtree of node *c*.

Algorithm 2. Reader side algorithm using the bit array scheme

```
01: IDLE = -1;  // No tag replied -- idle cycle
02: SUCCESS = n;// Only one tag replied -- success cycle
03:
04: startReader( ) {
05:   while( true) identifyAllTags( );
06: }
07:
08: identifyAllTags( ) {
09:       poc[0..n-1] = pre[0..n-1] = ‘0’;  // reset every bit.
10:
11:       c = 0;   // assume collision at the root.
12:       poc[c] = '1';  pre[c] = '0';
13:
14:       while( true )
15:       {
16:               broadcast( pre[0..c] );
17:
18:               // d = 1st collision index (c < d < n),
19:               d = receive( pre[c+1..n-1] );
20:
21:               if( 0 < d < n ) { // collision occurred at index d
22:                       c = d;  poc[c] = '1';  pre[c] = ‘0’;
23:                       continue;
24:               }
25:
26:               // success or idle cycle.
27:               if( d == SUCCESS ) read tag ID pre[0..n-1];
28:
29:               // all tags are identified when every bit is zero,
30:               if(poc[0..n-1] == ‘0’ ) return;
31:
32:               c = index of the right most ‘1’ in poc[0..n-1];
33:               // go to the right child of c.
34:               poc[c] = '0';  pre[c] = '1';
35:       } // end of while
36: }
```

For the reader side algorithm, as can be seen in Algorithm 2, the reader resets every bit of poc[0..*n*-1] and pre[0..*n*-1] and initializes poc[0]/pre[0]= 1 /0 at line 12 so that the reader can check if there is any tag ID starting with 1 after probing tag IDs that start with 0. In the while loop, the reader broadcasts a query pre[0..c] and receives the rest in pre[*c*+1..*n*-1] through the `receive( )` function. The `receive( )` function checks if there is a collision between responses from tags while it receives reply messages from tags. If there is a collision, it returns the first collision bit position *d*. Otherwise, it returns IDLE, which denotes no response from any tag, or SUCCESS, which denotes only one response from a tag. In the case of a collision, the reader follows the left branch of node *d* marking poc[*d*] = 1 at lines 22-23. If the response is SUCCESS, the reader reads the tag ID pre[0..*n*-1] at line 27. If every bit of poc[0..*n*-1] is 0 at line 30, the function `identifyAllTags()` returns since the reader has identified all the tags in its signal range once. Otherwise, the reader finds the rightmost 1 in poc[0..*n*-1], sets its index to *c*, and visits the right branch of node *c* at lines 32-34. After these updates of c, poc[ ], and pre[ ], the reader broadcasts the updated query pre[0..c] at line 16 to proceed with further identification.

Table 1 shows the steps to identify all the tags in Fig. 2. Each step represents a cycle in which the reader broadcasts a query and processes tag responses with their IDs. Tag IDs received from tags are represented within the set of parentheses. The symbols 'x' and '-' denote the first collision bit detected and the bit ignored by the reader, due to the collision at a prior bit position, respectively.

Table 1. Tag identification steps of bit array scheme

Step	poc[0..7] pre[0..7]								Description
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Array index**
1	1	0	0	0	0	0	0	0	Initialize
	0	(						)	idle
2	0	0	0	0	0	0	0	0	Collision at 1
	1	(x	-	-	-	-	-	-)	
3	0	1	0	0	0	0	0	0	Collision at 4
	1	0	(1	0	x	-	-	-)	
4	0	1	0	0	1	0	0	0	Tag1 read
	1	0	1	0	0	(1	1	1)	
5	0	1	0	0	0	0	0	0	Tag2 read
	1	0	1	0	1	(0	0	0)	
6	0	0	0	0	0	0	0	0	Tag3 read
	1	1	(0	0	1	0	1	0)	Done (Return)

In Step 1 of Table 1, the reader broadcasts a query string of '0' (pre[0]) and receives nothing from the tags. In Step 2, the reader finds the rightmost 1 at position(index) 0, sets poc[0]/pre[0] = 0/1, respectively, and broadcasts string 1 in pre[0] and receives pre[1..7] in which the first collision marked 'x' occurs at position 1 among the three tags. In Step 3, the reader sets poc[1]/pre[1] = 1/0 and broadcasts string 10 and receives pre[2..7], in which the first collision occurs at position 4. In Step 4, the reader sets poc[4]/pre[4] = 1/0 and broadcasts the string '10100' and receives pre[5..7]. In this step, the reader recognizes tag1 since only tag1 replied. In Step 5, the reader sets poc[4]/pre[4] = 0/1 since it finds the rightmost 1 in poc[0..n-1] at position 4, and broadcasts '10101' to identify tag2. In Step 6, the reader sets poc[1]/pre[1] = 0/1, since it finds the rightmost 1 in poc[0..*n*-1] at position 1, and broadcasts '11' to read tag3. At this point, every bit of poc0..*n*-1] is 0. This means that all the tags have been identified once. So, the function identifyAllTags( ) is done.

Theorem 1. The proposed protocol uses only Θ(*n*) memory space where *n* is the size of a tag ID in bit.

Proof. The proposed protocol uses two bit arrays poc[0..*n*-1] and pre[0..*n*-1] and integer variables *c* and *d*. So, the space complexity is Θ(n) since the size of the two n-bit arrays dominate that of integer variables. ■

Theorem 2. The time complexity of the proposed protocol is Θ(m) cycles where m is the number of tags to identify.

Proof. In the compressed query tree, every internal node, except for the root, has exactly two children. So, the total number t of nodes is $i + m$, where i is the number of internal nodes and m is the number of leaf nodes.

$$t = i + m \tag{1}$$

Also, the total number t of nodes can be expressed in $2i + 1$, where $2i$ is the number of branches.

$$t = 2i + 1 \tag{2}$$

From (1) and (2),

$$t = 2m - 1 \tag{3}$$

On the other hand, in the function identifyAllTags( ) of the proposed protocol, as shown in Algorithm 2, each iteration of the while statement forms a cycle, which is performed at each node. From (3), the total number of cycles for m tags identification is t that is equal to $2m\text{-}1$.

Therefore, the total time complexity to identify m tags within the reader's signal range is O($2m\text{-}1$) = O(m). ■

4. Experimental Results

To obtain an experimental evaluation of the memory space efficiency of our bit array scheme in relation to queue and stack schemes, we implemented simulation programs of compressed query tree protocols using queue, stack, and bit arrays. Queue and stack were implemented in linked structures using the node structure of Fig. 4. Each node consists of `queryPtr` pointing to a query string and `nextNodePtr` pointing to the next node. The query string is a variable size q of a bit string. Assuming that the size of a pointer field is 32 bits, the size of a node to store a query string is $2*32+q = 64+q$ bits. The memory space of all the three fields was counted in our experimental results for the queue and stack schemes. For our bit array scheme, the wasted memory size for the two bit arrays poc[0..n-1] and pre[0..n-1] is $2n$ where n is the size of a tag ID. In our simulation, the size of a tag ID was 96 bits.

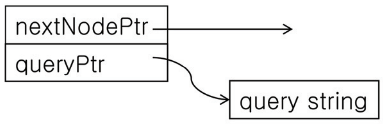

Fig. 4. Node structure of stack and queue.

We experimented with right-aligned sequential tag IDs, left-aligned sequential tag IDs, and random tag IDs, in which each tag ID was randomly generated. The right/left-aligned sequential tag IDs means that each generated tag ID is placed to the right/left end of a tag ID bit array, respectively. For the experiments with the random tag IDs, we ran ten different sets of tag IDs, measured the maximum space used for each scheme, and calculated the average of the ten maximum values.

Tables 2–4 show the memory space consumed for each scheme using sequential and random tag IDs, respectively. The number in parentheses is the memory space efficiency that is defined to be the ratio of the memory space of a scheme divided by that of the bit array scheme. As can be seen in Tables 2–4, the queue scheme is the worst and the bit array scheme is the best. As the number of tags increases, the queue and stack schemes become worse than the bit array scheme.

Table 2. The wasted memory space using left-aligned sequential tag IDs in bit

Number of tags	Queue	Stack	Bit array
32	2208 (11.5)	335 (1.7)	192
64	4480 (23.3)	405 (2.1)	192
128	9088 (47.3)	476 (2.5)	192
256	18432 (96.0)	548 (2.9)	192
512	37376 (194.7)	621 (3.2)	192
1024	75776 (394.7)	695 (3.6)	192
2048	153600 (800.0)	770 (4.0)	192
4096	311296 (1621.3)	846 (4.4)	192
8192	630784 (3285.3)	923 (4.8)	192
16384	1277952 (6656.0)	1001 (5.2)	192

The number in the parentheses is the space efficiency relative to the bit array scheme.

Table 3. The wasted memory space using right-aligned sequential tag IDs in bit

Number of tags	Queue	Stack	Bit array
32	5120 (26.7)	855 (4.5)	192
64	10240 (53.3)	1010 (5.3)	192
128	20480 (106.7)	1164 (6.1)	192
256	40960 (213.3)	1317 (6.9)	192
512	81920 (426.7)	1469 (7.7)	192
1024	163840 (853.3)	1620 (8.4)	192
2048	327680 (1706.7)	1770 (9.2)	192
4096	655360 (3413.3)	1919 (10.0)	192
8192	1310720 (6826.7)	2067 (10.8)	192
16384	2621440 (13653.3)	2214 (11.5)	192

The number in the parentheses is the space efficiency relative to the bit array scheme.

Table 4. The wasted memory space using random tag IDs in bit

Number of tags	Queue	Stack	Bit array
32	4742 (24.7)	624 (3.3)	192
64	9676 (50.4)	739 (3.8)	192
128	19072 (99.3)	835 (4.3)	192
256	39372 (205.1)	962 (5.0)	192
512	76800 (400.0)	1036 (5.4)	192
1024	152780 (795.7)	1134 (5.9)	192
2048	305971 (1593.6)	1255 (6.5)	192
4096	641024 (3338.7)	1399 (7.3)	192
8192	1244364 (6481.1)	1479 (7.7)	192
16384	2382233 (12407.5)	1537 (8.0)	192

The number in the parentheses is the space efficiency relative to the bit array scheme.

5. Conclusion

This paper proposes a tag identification scheme using two bit arrays with *n* bits each, where *n* is the size of a tag ID, and demonstrates its memory efficiency in relation to queue and stack schemes. The size of the two bit arrays depends on only the length of tag ID and is independent of the number of tags to identify, while the size of the queue/stack of queue/stack schemes depends on both of them. Also, the proposed bit array scheme would be easier to implement in hardware for speedup than for queue or stack schemes.

References

[1] C. Law, K. Lee, and K. Y. Siu, "Efficient memoryless protocol for tag identification," in Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, 2000, pp. 75-84.
[2] J. Myung and W. Lee, "An adaptive memoryless tag anti-collision protocol for RFID networks," in Proceedings of the 23rd Conference of the IEEE Communications Society, 2005, p. 1-3.
[3] A. Juels, R. L. Rivest, and M. Szydlo, "The blocker tag: selective blocking of RFID tags for consumer privacy," in Proceedings of the 10th ACM conference on Computer and Communication Security, 2003, pp. 103-111.
[4] H. G. Seo, "Collision tree based anti-collision algorithm in RFID system," Journal of KISS: Information Networking, vol. 34, no. 5, pp. 316-327, Oct. 2007.
[5] X. Jia, Q. Feng, and C. Ma, "An efficient anti-collision protocol for RFID tag identification," IEEE Communications Letters, vol. 14, no. 11, pp. 1014-1016, 2010.
[6] EPC Tag Data Standard, http://www.gs1.org/gsmp/kc/epcglobal/tds.
[7] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of Data Structures in C++ (2nd ed.). Summit, NJ: Silicon Press, 2007.
[8] H. Jung, "A succinct anti-collision protocol for RFID tag identification," in Proceedings of ICKIICE, 2012, pp. 21-23.
[9] K. Finkenzeller, RFID Handbook: Fundamentals and Applications in Contactless Smart Cards, Radio Frequency Identification and Near-Field Communication (3rd ed.). Hoboken, NJ: John Wiley & Sons, 2010, pp. 199-211.

Haejae Jung http://orcid.org/0000-0002-6538-168X
He received a Ph.D. degree in Computer & Information Science & Engineering from University of Florida in 2000. He has been with the department of information and communication engineering at Andong National Univ. since 2005. His research interest is in the design of efficient computer algorithms.