Efficient Access Algorithms for Dynamic Many-tag Passive RFID Storage Systems
Technical Report

Victor K.Y. Wu* and Nitin H. Vaidya
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Email: {vwu3, nhv}@illinois.edu

Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: rhc@illinois.edu

***Abstract*—In this work, we investigate efficient algorithms for accessing information stored in a large number of RFID tags. Interrogators, equipped with RFID scanners, regularly arrive at the tag system, read from and write messages to the tags, and then leave. We develop techniques for the interrogators to read the newest messages from the tags as quickly as possible, since newer information is often the most relevant and interesting. We borrow ideas from aloha and query tree, two traditional singulation schemes, to form our *many-tag* access algorithms. Our query tree-based algorithms access tags faster, but are not as robust as our aloha-based algorithms. We study two scenarios. In the static scenario, the tag population is fixed. Here, it is naive to initially singulate all the tags, and then query them afterward to read the newest messages. Our results indicate that better-performing algorithms instead progressively segment the tag population at each round to find the newest messages. In the dynamic scenario, tags are continually arriving and departing, causing information to quickly disappear. We combat this by encoding messages, dividing the coded bits into multiple chunks, and then spreading them across multiple tags. This requires us to access a large number of tags when reading. Therefore, our results show that better-performing algorithms initially singulate all the tags (as opposed to the static scenario), and then continually query them afterward individually for data chunks in order to recover the newest messages.**

I. Introduction

RFID (radio frequency identification) technology is moving beyond its namesake of object identification. In particular, system designers are leveraging the user memory of passive RFID tags as storage systems. In this work, we investigate efficient algorithms to access information stored in a large number of tags.

A. Motivating application domains

We consider three potential application areas to motivate many-tag information access. This is by no means an exhaustive list.

1) Supply chain management: Passive RFID technology is traditionally used in warehousing and supply chain management. As goods move from factory production to consumer consumption, tags affixed to pallets and shipment containers aid in the tracking process. RFID interrogators (scanners) at various checkpoints scan the tags for their unique IDs, allowing the tracking system to monitor the flow of goods. Tags can also carry tracking history information, further improving the system.

2) RFID tag fields: In many mapping and localization applications, tags are distributed over a large physical area. As people or mobile robots equipped with interrogators move through the tag field, they can access the tags, reading from and writing to them messages, such as location information for use in various algorithms.

3) Whiteboarding: A system of many tags can form a digital whiteboard. Users equipped with interrogators can collaborate with each other by sharing information via the tags' storages.

B. Many-tag information access

We introduce the term *many-tag* to emphasize that tag multiplicity plays a defining role in the systems we consider. In single-tag scenarios, or multi-tag systems with a small number of tags (less than 50), we can use traditional means of tag access. That is, an interrogator first singulates[1] the tags, learning their IDs. Then, it reads and writes information by querying the tags individually according to their IDs. In this work, we consider many-tag systems, where an interrogator's scan range is powerful enough to encompass upwards of 1000 tags. (Equivalently, the interrogator transmit power may be weaker, but the tags are positioned more densely in space.) In these scenarios, we need algorithms that quickly access the information of interest. For example, we are often interested in reading the newest information, because it is the most relevant. Symmetrically, if we are writing information to an already full storage system and are forced to overwrite something, we

*This work is supported in part by NSF grant CNS-0519817.

[1]Singulation is the process whereby an interrogator learns the unique IDs (and thus the presence) of a set of tags. In passive RFID, the interrogator scans the batteryless tags, powering their chips. The interrogator queries the tags with certain conditions. Tags respond by sending their unique IDs. If only one tag responds, the interrogator learns its ID. If multiple tags respond, there is a collision. The interrogator resolves these collisions over several query rounds to learn all the IDs.

often choose to replace the oldest information. In this work, we focus on these two ideas.

We also note that our algorithms provide a black box interface for users. That is, a user equipped with an interrogator can interact with a system of tags, oblivious to the physical layer communications. She does not know the number of tags present, and does not need to access tags on an individual basis. In this sense, our algorithms are a form of middleware for RFID tag storage.

C. Summary of contributions

In this work, we borrow two traditional RFID singulation algorithms, aloha and query tree, to form our many-tag access algorithms. Our query tree-based algorithms access tags faster, but are not as robust as our aloha-based algorithms. We first study the static case, where the tag population is fixed. In this scenario, our results indicate that the better-performing algorithms progressively segment the tag population at each round to find the newest information. In the dynamic case, tags are continually arriving and departing. We anticipate that this will cause information to quickly disappear. Therefore, we combat this by encoding information, dividing the coded bits into multiple chunks, and spreading them across multiple tags. This requires us to access a large number of tags when reading information. Therefore, our results show that the better-performing algorithms initially singulate all the tags, and then continually query them individually for data chunks in order to recover information.

D. Outline

In Section II, we review relevant background literature. In Section III, we introduce our system model, and detail and evaluate our algorithms, for the static case. In Section IV, we generalize our ideas to the dynamic case. Section V concludes and provides future work.

II. Background literature

To the best of our knowledge, researchers are not currently studying information access in many-tag systems. We therefore review background literature that does already use tag storage, but could benefit even more using our proposed algorithms.

A. Manufacturing and supply chain management

In [1], the authors use RFID for smart parts manufacturing. As automobile parts affixed with tags move through the production line, interrogators scan the tags to ensure the processes are correct. For example, a particular process may require certain safety components to be present. Also, as these smart parts move between different domains (such as plants, automobile dealers, and repair shops), interrogators can scan them for service histories. Storing information inline in the tags themselves is helpful, since it may be difficult for these different domains to access a centralized information database. In this way, we can affix more tags to more parts or subcomponents, or even affix multiple tags to a single part for redundancy. The number of tags quickly increases, and therefore many-tag information access becomes very important.

In [2], the authors consider asset management in the supply chain. Tags are affixed to shipping containers in order to track them. Similar to smart parts above, these containers move through many different domains, and storing history information inline in the tags is a good design choice. We can again affix many tags to these containers, and access their information using our algorithms.

B. RFID tag fields

In [3], [4], [5], tags are distributed over a large physical area. Mobile robots equipped with interrogators move through the tag field, reading from and writing to the tags, for localization and mapping. In these works, the authors focus on scanning the tags for their IDs, and use tag storage to store tracking information. In [6], [7], we also consider a tag field, but with people carrying interrogators. We focus on tag storage for applications such as search and rescue. If the tags are deployed densely enough, or interrogators have sufficient scan range, they can potentially scan up to 1000 tags at any given moment. In these situations, many-tag information access algorithms become very useful.

C. Whiteboarding

In [8], the authors develop a system to collect information in a post-disaster scenario. RFID tags are deployed at disaster sites, after an earthquake for example. After authorities access the damage at a site, the results are stored in the tags. At a later stage, information can be easily aggregated since it is stored at the sites themselves. This is important, since a communications system may be unavailable. As well, we can further generalize using tags in these situations. For example, first responders can deploy tags at various key strategic locations, such as the ground zero location(s), medical tents, emergency shelters, and parking lots. People read and write to the tags, as well as carry them between these locations. This creates an ad-hoc communications network (which may also be delay-tolerant [9], [10]) for people to share messages, such as rescue status updates, food and medical supplies availability, and any other pertinent information. In essence, we can view the dynamic systems of tags as digital whiteboards, where we can apply our many-tag information access algorithms.

III. Static RFID system

A. System model

We first consider the static case where there is a system of n passive RFID tags fixed in a physical area. Each tag can store one message and an associated timestamp. Interrogators regularly arrive at the system, read from and write messages to the tags (by scanning them), and then leave. In particular, interrogator arrivals are modeled as a Poisson process with rate λ_I arrivals per second. We assume that the tag access time (reading and writing) during each interrogator's stay is negligible compared to the interrogators' inter-arrival times. Therefore, there is at most one interrogator at the system

any given time. (That is, we do not consider the interrogator collision problem [11], since it is beyond the scope of this work.) During each interrogator's stay, it writes X messages to X different tags, where X is a non-negative geometric random variable with mean $\frac{1-p}{p}$, where $p \in (0,1)$, and X is independent across stays. The interrogator first writes to any empty tags. Then, for any remaining messages, it overwrites existing messages in the tags, starting with the oldest one, and then the second oldest, ... When an interrogator writes a message to a tag, it also writes a timestamp of the current time to the tag. (Note that we say a tag is "new" or "old" if the timestamp it is currently storing is large or small, respectively.)

B. Algorithms

Reading and writing require finding the newest or oldest messages, respectively, which are symmetric processes in the following algorithms. Therefore, we only focus on reading. In particular, when an interrogator arrives at a system of tags already in steady state (all tags are full), these algorithms find the m different newest tags (learn their unique IDs) with the m newest messages, where $m \in \{1, \ldots, n\}$. Then, the interrogator queries each of these m tags individually to recover the messages. The algorithms are categorized into two classes, *aloha-based* and *query tree-based*. Aloha-based algorithms include {**aloha-normal, aloha-max, aloha-half**}. Query tree-based algorithms include {**query tree-normal, query tree-max**}.

1) Aloha-based: These class of alogrithms use aloha singulation [12]. In aloha, the interrogator singulates n tags in multiple query rounds. In each round, the interrogator first broadcasts $N \in \{16, 32, 64, 128, 256\}$ to all the tags, which is the number of tag response time slots. N depends on the interrogator's estimate of the tag population size. ([12] shows that $N > 256$ is not necessary.) The interrogator then listens for N time slots. Each tag randomly (uniformly) chooses one of those time slots to respond in with its ID. The interrogator learns the ID of a tag if no other tags respond in the same slot as it does. (There are no collisions in that slot.) At each round, the estimate of n, which we call $\hat{n}$, changes in general, and therefore N changes too. This repeats over multiple rounds until the interrogator is confident that is has singulated 99% of the tags, as detailed in [12].

In the first round, we initialize N to 16. In subsequent rounds, we double N if all slots in the previous round have collisions. Otherwise, we choose N according to $\hat{n}$, which we calculate with the following. First, we need a lookup table (stored in the interrogator) with E_0^n, E_1^n, and $E_{\geq 2}^n$, which is the expected number of empty slots, single occupancy slots, and collision slots, respectively, in the previous round, for varying values of n of N. These formulas are derived in [12].

$$E_0^n = N\left(1 - \frac{1}{N}\right)^n, E_1^n = n\left(1 - \frac{1}{N}\right)^{n-1}, \text{ and} \quad (1)$$

$$E_{\geq 2}^n = \frac{N^n - (N-1)^{n-1}(N+n-1)}{N^{n-1}}. \quad (2)$$

Now, let $s_0, s_1, s_{\geq 2}$ be the number of empty slots, single occupancy slots, and collision slots, respectively, measured from the previous round. Then,

$$\hat{n} := \arg\min_n \left|\left|\left(E_0^n \ E_1^n \ E_{\geq 2}^n\right) - \left(s_0 \ s_1 \ s_{\geq 2}\right)\right|\right|. \quad (3)$$

(Note that we use the E_0^n, E_1^n, and $E_{\geq 2}^n$ values associated with the previous round's N in the above minimization.) The interrogator then chooses N for the current round according to the following ranges of $\hat{n}$, as shown in [12].

$\hat{n} \in$	$[1,9]$	$[10,27]$	$[28,56]$	$[57,129]$	$[130,\infty)$
N	16	32	64	128	256

In **aloha-normal**, the interrogator first uses aloha to singulate all the tags. Tags respond with their respective timestamps, in addition to their IDs. The interrogator therefore learns all the tags' IDs, and their associated timestamps. It knows which tags are new. It then queries the m newest tags individually according to their IDs for the m newest messages.

For **aloha-max**, let $TS = \{ts_i\}_i$ be the set of timestamps the interrogator has collected from single-occupancy time slots in the current read session. Initialize TS with $TS := \{-\infty\}$. In each aloha round, first let $ts_{largest} := \max_i TS$. Then, the interrogator broadcasts N and $ts_{largest}$ to the tags. If a tag with timestamp ts, has $ts > ts_{largest}$, it responds (in one of the N time slots), with its ID and ts. For each single-occupancy time slot, the interrogator learns an ID and an associated ts, and updates $TS := TS \cup ts$. In this way, with each round, $ts_{largest}$ increases, and the interrogator progressively queries an effectively smaller proportion of the tag population. When tags no longer respond to the interrogator's broadcast, the interrogator knows that $ts_{largest}$ contains the largest timestamp among all the tags. It then queries that tag (using the ID associated with $ts_{largest}$) to read the newest message. To find the second newest message, (the third newest, ..., and the mth newest), the interrogator first mutes the newest tag it just read from (tell it not to respond anymore for this read session), and updates $TS := TS \setminus ts_{largest}$. The interrogator then repeats the above process. It becomes faster to find each subsequent newest message, since TS contains increasingly more timestamps.

Determining $\hat{n}$ is more difficult in aloha-max than aloha-normal, since the effective tag population size (tags that should respond) changes with each round. To keep track of the tag population, we first estimate the number of tags that are written to in a given period T with the mean.

$$\begin{aligned} &E\left[\text{number of messages written in } T \text{ seconds}\right] \\ &\quad = \sum_{i=0}^{\infty} E\left[\text{number of messages written} | i \text{ arrivals in } T\right] \\ &\qquad\qquad \times P\left(i \text{ arrivals in } T\right). \\ &\quad = \sum_{i=0}^{\infty} i \frac{1-p}{p} \frac{e^{-\lambda_I T}\left(\lambda_I T\right)^i}{i!} = \frac{1-p}{p}\lambda_I T. \end{aligned} \quad (4)$$

Then, at each round, the interrogator doubles N, if s_0 and s_1 are both zero in the previous round. Otherwise, it first

TABLE I
ALOHA-BASED COMMUNICATIONS

Aloha-normal	$I \xrightarrow{N} \{T_j\}_{j=1}^{n}$ For $j \in \{1, \ldots, n\}$, in $k_{T_j}^{th}$ time slot: $I \xleftarrow{ID_j, ts_j} T_j$,
Aloha-max	$I \xrightarrow{N, ts_{largest}} \{T_j\}_{j=1}^{n}$ For $j \in \{1, \ldots, n\}$, if $ts_j > ts_{largest}$, then in $k_{T_j}^{th}$ time slot: $I \xleftarrow{ID_j, ts_j} T_j$
Aloha-half	$I \xrightarrow{N, ts_{average}} \{T_j\}_{j=1}^{n}$ For $j \in \{1, \ldots, n\}$, if $ts_j > ts_{average}$, then in $k_{T_j}^{th}$ time slot: $I \xleftarrow{ID_j, ts_j} T_j$

estimates $\hat{n}$ for the previous round using (3). Then, it estimates the time spread (of messages' timestamps) of this previous round by taking the difference between the maximum and minimum timestamps collected in this previous round. We call this T_{spread}. From (4), it estimates $\frac{1-p}{p}\lambda_I T_{spread}$ as the number of tags written to in that particular time frame (which is in the past). In the current round, the tags written to in that time frame do not respond, since they are segmented out with the interrogator broadcasting $ts_{largest}$. The interrogator then updates the estimated number of tags that respond in the current round to be $\hat{n} := \lceil \hat{n} - \frac{1-p}{p}\lambda_I T_{spread} \rceil$. (If $\hat{n}$ turns out to be non-positive, set it to 1.) N is then determined from this new $\hat{n}$ using the same ranges described above for aloha singulation. Note that aloha-max requires an interrogator to know the statistics of previous interrogators. Namely it has to know λ_I and p.

In **aloha-half**, we assume that the interrogator does not know the statistics of previous interrogators. This makes it difficult to estimate the effective tag population size dynamically (and therefore adjust N accordingly). So instead of using $ts_{largest}$ as the "cut-off time", the interrogator uses the median. That is, a tag responds in the current round only if its timestamp is greater than the median of the timestamps collected by the interrogator in the previous round. Therefore, the tag population estimate is easily updated as $\hat{n} := \lceil \frac{\hat{n}}{2} \rceil$. (Again set $\hat{n}$ to be 1 if it is non-positive.) In essence, the interrogator is approximately halving the effective tag population in each round. When only one tag responds to the interrogator's broadcast, it knows that that tag is the largest timestamp tag. Everything else is the same as aloha-max.

We summarize the communications of the aloha-based algorithms in a single round in Table I. I is the interrogator, $ts_{largest}$ is the largest timestamp it has collected so far, and $ts_{average}$ is the average of the timestamps it collected in the previous round. T_j is the jth tag, with ID ID_j, and ts_j is its stored timestamp, where $j \in \{1, \ldots, n\}$. T_j responds in the $k_{T_j}^{th}$ time slot (if necessary), where $k_{T_j}^{th} \in \{1, \ldots, N\}$.

2) Query tree-based: These class of algorithms use query tree singulation [13]. In query tree, the interrogator singulates the n tags in multiple rounds. In each round, the interrogator broadcasts a bit string. A tag that has an ID that prefix matches the bit string responds with its entire ID. If only one tag responds, then that tag is successfully singulated. Then the interrogator chooses another bit string for the next round. Otherwise, multiple tags respond, and there is a collision. The interrogator then uses a longer bit string in the next round. Essentially, the interrogator walks through a binary tree starting at the root (using either depth-first search or breadth-first search) until it singulates all the tags. Note that not all nodes have to be visited. For example, in Fig. 1, when the bit string 01 is queried, only the tag with ID 0110 responds, and therefore it is singulated right away. Nodes $010, 011, 0100, 0101, 0110, 0111$ are not visited. (These bit strings are not queried.)

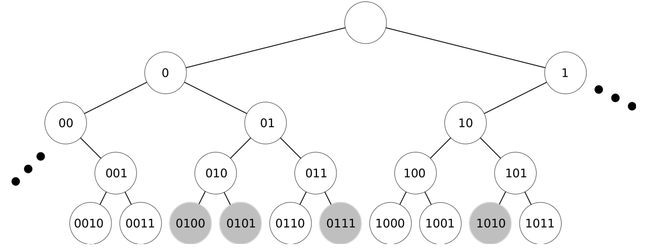


Fig. 1. Query tree singulation for a 4 bit tag ID space. Each node in the tree has an associated bit string indicating its position in the tree. The leaves indicate potential tags in the system. Shaded leaves mean that that tag is not in the system. Non-shaded leaves are tags in the system. Their bit strings represent their tag IDs.

In **query tree-normal**, the interrogator first uses query tree to singulate all the tags. Tags respond with their respective timestamps, in addition to their IDs. The interrogator therefore learns all the tags' IDs, and their associated timestamps. It then queries the m newest tags individually according to their IDs for the m newest messages.

In **query tree-max**, the interrogator progressively queries an effectively smaller proportion of the tag population with each round, similar to aloha-max. Let $TS = \{ts_i\}_i$ be the set of timestamps the interrogator has collected in the current read session. In each query tree round, first let $ts_{largest} := \max_i TS$ Then, the interrogator broadcasts a bit string and $ts_{largest}$. If a tag with timestamp ts, has $ts > ts_{largest}$, and an ID prefix match with the bit string, it responds with its ID and ts. (Note that timestamps are in general not in order according to IDs.) Each time the interrogator successively receives a tag's response (no collision), it updates $TS := TS \cup ts$. In this way, $ts_{largest}$ increases, and the interrogator progressively queries an effectively smaller proportion of the tag population with each round. When tags no longer respond to the interrogator's broadcast, or query tree is complete, $ts_{largest}$ contains the largest timestamp among all the tags. The interrogator then queries that tag (using the ID associated with $ts_{largest}$) to read the newest message. The interrogator repeats the above process, updating TS and muting tags, similar to aloha-max, to find the second newest message, the third newest, $\ldots$, m^{th} newest.

TABLE II
SIMULTANEOUS BITS TRANSMITTED IN EACH ROUND

Aloha-based	
Aloha-normal	Interrogator queries: 3 bits Tags respond: $N\,(96+17)$ bits
Aloha-max	Interrogator queries: $3+17$ bits Tags respond: $N\,(96+17)$ bits
Aloha-half	Interrogator queries: $3+17$ bits Tags respond: $N\,(96+17)$ bits
Query tree-based	
Query tree-normal	Interrogator queries: length(bit string) bits Tags respond: $96+17$ bits
Query tree-max	Interrogator queries: length(bit string) + 17 bits Tags respond: $96+17$ bits

C. Evaluation

1) Simulation performance metric: We simulate our algorithms. We are interested in the **message access time**. In particular, since reading and writing are symmetric (finding the newest and oldest tags are effectively the same), we focus on reading. To compare the different algorithms, we abstract out the wireless transfer bit rates between the interrogator and tags. We only count the total number of simultaneous bits that are transmitted through the air interface to find the IDs of the m newest tags. (We say "simultaneous", since multiple tags may respond at the same time. Additionally, tags may not respond, but time may still elapse, for the case of empty time slots in the aloha-based algorithms.)

Note that we are only measuring the time the interrogator uses to find the IDs of the m newest tags carrying the m newest messages. Afterward, the interrogator reads actual message data by querying these tags individually. Since this is the same for all the algorithms, we do not include this message data transfer time in our metric.

In our simulations, we consider the UHF Class 1 Gen 2 passive RFID tag [14], which uses a 96-bit unique ID. Timestamps are chosen to be 17 bits long, giving us a precision of seconds in a 24 hour period. N requires 3 bits, since it can take on 5 different values. We summarize the time (simultaneous bits transmitted) required for each query round of the algorithms in Table II.

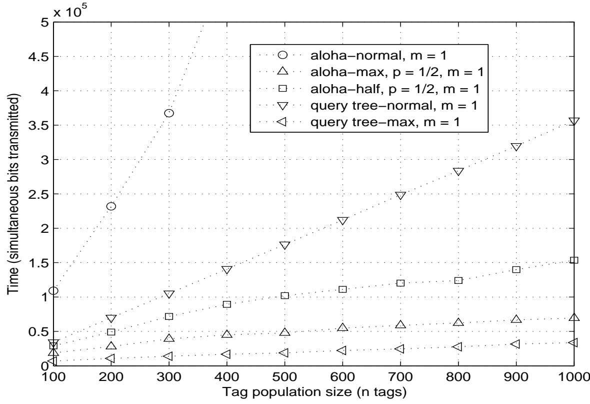
2) Simulation results and discussion: We simulate the average message access time when the system is in steady state, for $\lambda_I = 1$ arrival per second. Results are shown in Fig. 2. Figs. 2(a), 2(b), and 2(c) plot access time against the number of tags, n. Fig. 2(d) plots against m. Fig. 2(a) shows that aloha-normal and query tree-normal are naive schemes. They require singulating all the tags initially, using a lot of query rounds, thus resulting in a long access time. Aloha-max and aloha-half perform well, with the former better (as shown in Fig. 2(b)), as expected, since it is more aggressive in segmenting the tag population. Of course, the tradeoff is that aloha-max requires knowing the interrogators' statistics. Query tree-max is even better, since it segments the population, while performing query tree. The interrogator is effectively pruning the query tree with each round, allowing it to quickly find the newest tag with the largest timestamp.

For aloha-max and aloha-half, we see that varying p changes the performance very little, as shown in Fig. 2(d). In particular, it may slightly change how the tag population n is estimated in each round. The query tree-based algorithms do not use p in their algorithms, and therefore their performances are independent of p. That is, these algorithms just look at the relative ordering of the timestamps.

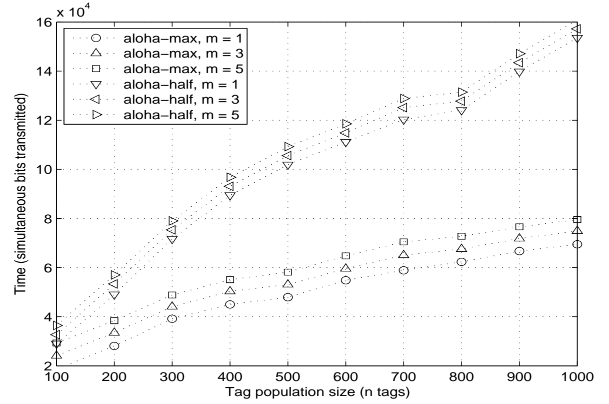
Fig. 2(d) shows how the access time varies as m is increased. We see that the marginal time to read each additional message is very small. That is, finding the newest tag takes the most time. Finding the second newest, third newest, $\ldots$, m^{th} newest requires little time, since the interrogator has already collected many timestamps, and thus has a "head start" in finding subsequent newest tags.

We see that the query tree-based algorithms perform better than the aloha-based ones. Both query tree and aloha do not require knowing the tag population size. However, aloha does continually estimate the number of tags with each round. Therefore, the aloha-based algorithms must pay this cost of $N\,(96+17)$ bits in the access time in each round, which is especially wasteful in the initial rounds when the interrogator is still learning the tag population size. However, aloha-based algorithms are in general more robust. Even if the environment changes (such as obstructing objects changing the wireless propagation characteristics) quickly within one interrogator access session, aloha handles this gracefully, since in each round, the interrogator scans all the tags it can and singulates them, whether or not those tags were scannable in previous rounds. In contrast, suppose in the query tree-based algorithms, the interrogator misses scanning a tag initially because of an obstructing object. The algorithms may quickly prune out the segment of the tree with that tag ID. Later when the obstruction is gone and that tag is within the scan range, the interrogator cannot singulate it, even if the tag carries the newest message.

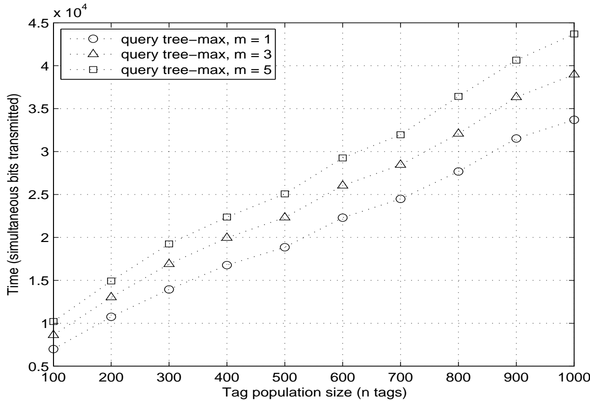
In our algorithms, we use timestamps. In practice, there are difficulties with this. First, we need to transmit the timestamps and store them in tags, which requires storage overhead. Second, interrogators need access to a synchronized clock, which is not necessarily trivial, depending on the granularity of the timestamps. Instead of using timestamps, one may consider logical sequence numbers. But aloha-max and aloha-half cannot use sequence numbers since they rely on the relative times of interrogator arrivals. The other algorithms can use sequence numbers. With sequence numbers, the problem of finding the newest message can become trivial. That is, if the interrogator knows what the newest sequence number has been assigned so far, it can use that right away to find the newest message. However, this is not possible if interrogators do not communicate with each other or they come from different domains. Additionally, if tags dynamically arrive and leave (which we address in the next section), it becomes difficult to track sequence numbers. Therefore, we argue for using timestamps, despite its difficulties.



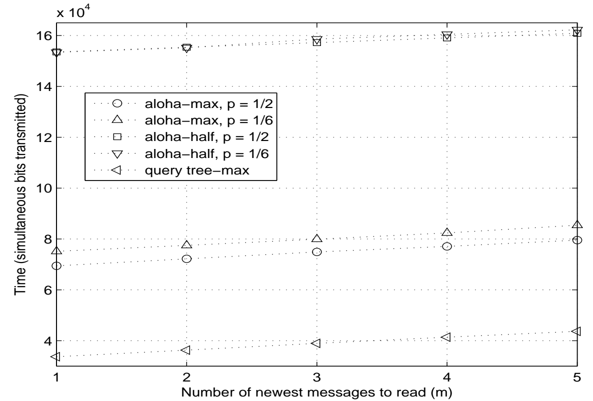
(a) Comparison of all algorithms for reading newest message



(b) Aloha-max and aloha-half, $p = \frac{1}{2}$



(c) Query tree-max



(d) $n = 1000$ tags

Fig. 2. Average message access time, $\lambda_I = 1$ arrival per second

IV. Dynamic RFID system

A. System model

1) Tag and reader dynamics: In the dynamic case, tags are continually arriving and departing, and therefore the tag population size changes dynamically. We model the situation as an $M/M/\infty$ queueing system. That is, tags arrive according to a Poisson process at a rate of λ_T arrivals per second. Each tag stays for an exponential time with mean $\frac{1}{\mu_T}$ seconds, and then departs, independent of all other tags. Therefore, at steady state, $E\left[\text{number of tags in system}\right] = \frac{\lambda_T}{\mu_T}$. (That is, for the dynamic RFID system, we define steady state as when the queueing system of tags is at steady state. Thus, there are likely to be non-full tags in steady state, as opposed to that of the case of the static RFID system.) Without loss of generality, we take $\lambda_T = 1$. Then, we vary $\mu_T \in (0, 1)$. As $\frac{1}{\mu_T}$ increases, the expected tag population size increases. Also note that if μ_T is large, tags leave sooner, and therefore the lifetimes of messages in the system are reduced. That is, a message is effectively destroyed if enough of the multiple tags carrying it (using message encoding, explained below) are no longer in the system.

Interrogators arrive according to a Poisson process at a rate of λ_I arrivals per second. After an interrogator arrives, it reads and writes very quickly, and then leaves. In particular, we assume this tag access occurs on a very small time scale compared to the tag dynamics. Practically, it means we can assume the tag population is fixed when an interrogator is accessing tags. We are, nonetheless, interested at how the tag access time varies at this microscopic level. During each interrogator's stay, it writes X messages, where X is a non-negative geometric random variable with mean $\frac{1-p}{p}$, where $p \in (0, 1)$, and X is independent across stays. When an interrogator writes a message to a tag, it includes a timestamp of the current time. If λ_I is large, or if p is large, or both, more messages are written to tags. Ultimately, this reduces the lifetimes of messages stored in the system, since they are quickly replaced.

2) Message encoding and tag storage queue: The lifetimes of messages are reduced if tags quickly leave and/or interrogators come often and overwrite a lot of messages. Therefore, we use Reed Solomon coding [15] to alleviate this problem. To store a k-byte message, an interrogator first encodes it into a q-byte codeword using an $RS\left(q, k\right)$ code. The codeword is then divided into q one-byte chunks, and written to different tags. To recover the message later, an interrogator must recover at least any k out of the q chunks (reading from multiple

tags), and also know their respective positions in the codeword. Therefore, we associate a sequence number with each of the q chunks. The sequence number thus requires $\lceil log_2 q \rceil$ bits. When an interrogator writes a message chunk to a tag, it includes the sequence number and a timestamp. The timestamp also serves as a message ID, identifying which chunks belong to which message, since all the chunks of the same message share the same timestamp.

A tag's storage is maintained as a first-in first-out (FIFO) queue with l storage slots. That is, when a tag arrives at the system, it is empty. As interrogators write message chunks to it (with associated sequence numbers and timestamps), by inserting chunks at the back of the queue, existing chunks are pushed through the queue. When the queue is full, the next incoming message chunk forces out the oldest existing chunk in the tag. In other words, the new chunks are at the back, and the old chunks are at the front. To access (read) chunks from the queue, an interrogator specifies the ith newest chunk in the queue, where $i \in \{1, \ldots, \text{queue size}\}$.

When an interrogator wants to write q message chunks of a message (with associated sequence numbers and timestamp), it first singulates all the tags. It then writes to any empty storage slots in the tags' queues first. If there are q_{remain} remaining chunks and n tags in the system, it writes to the $\min(n, q_{remain})$ tags (inserting chunks at the back of their respective queues) with the oldest timestamps. This is repeated until there are no more remaining chunks to be written. In essence, an interrogator spreads out the chunks among the tags as much as possible, while at the same time replacing the oldest information in the system. The interrogator can easily find the tags with the oldest timestamps by examining just the timestamps of each of the chunks at the front of each queue.

B. Algorithms

In this work, we focus on reading messages. As before, the following algorithms find the m newest messages stored in the tag system. Note that an interrogator only has to recover k chunks of a message to reconstruct and thus read it. If there are less than k chunks remaining in the system, that message is effectively destroyed, and can no longer be accessed.

1) Aloha-based: **Aloha-normal** is similar to its counterpart in Section III-B. In stage 1, the interrogator uses aloha to first singulate all the tags, learning their IDs. Then in stage 2, it queries tags individually, reading message chunks from them. That is, it reads the newest chunk (and sequence number) from every tag, and then the second newest from every tag, $\ldots$. After each read, the interrogator recovers as many messages as possible. That is, if at least k chunks of a message are recovered, the message itself is recovered. The interrogator stops reading chunks when it has recovered m messages. (These being the m newest messages). Messages in the system that have fewer than k surviving chunks are considered destroyed.

Note that it is difficult to use the aloha-max and aloha-half algorithms from before, because even if we know the interrogator statistics, we do not know if the interrogator necessarily can spread message chunks evenly across the tags, especially if there are very few tags. Tags arriving and departing also add to the uncertainty, making these algorithms infeasible.

2) Query tree-based: **Query tree-normal** is the same as aloha-normal, except that the interrogator uses query tree in the initial singulation process of stage 1. Everything else in stage 2 is the same, with the interrogator querying tags individually for their message chunks.

Query tree-max is similar to its counterpart in Section III-B. Let $TS = \{ts_i\}_i$ be the set of timestamps the interrogator has collected in the current read session. In stage 1, the interrogator finds the largest timestamp among all message chunks in all tags. In each query tree round, first let $ts_{largest} := \max_i TS$ Then, the interrogator broadcasts a bit string and $ts_{largest}$. If a tag with its largest (among all its chunks) unflagged timestamp ts, has $ts > ts_{largest}$, and an ID prefix match with the bit string, it responds with its ID and ts. Each time the interrogator successively receives a tag's response (no collision), it updates $TS := TS \cup ts$. In this way, $ts_{largest}$ increases, and the interrogator progressively queries an effectively smaller proportion of the tag population with each round. Stage 1 ends when tags no longer respond to the interrogator's broadcast, or query tree is complete. $ts_{largest}$ contains the largest unflagged timestamp among all message chunks in all tags. In stage 2, we focus on $ts_{interest} := ts_{largest}$. First, the interrogator broadcasts a notification, telling tags that have $ts_{interest}$ to flag it (so that it will be ignored in future iterations of stage 1). Then, the interrogator singulates these tags that have $ts_{interest}$, using query tree on the IDs. If a tag has $ts_{interest}$ (and an ID that prefix matches the broadcast string), it responds with the associated message chunk and sequence number (in addition to its ID). Stage 2 ends when the interrogator has collected k chunks, or has completed the singulation (in which case it may have collected less than k chunks, and therefore it knows that the message is no longer alive in the system). To find the next newest message, first update $TS := TS \setminus ts_{largest}$. Then, the interrogator goes through stage 1 and 2 again. This process repeats until the interrogator has recovered m newest messages.

C. Evaluation

1) Performance metrics: We simulate our algorithms. We are interested in the **measured message lifetime per byte** of a message in steady state. That is, a message is "born" when an interrogator writes its q constituent chunks to tags. It is destroyed when fewer than k chunks remain in the system, which may occur if chunks are overwritten. This is the "death" time. However, the message may also be destroyed if tags leave. In that case, we take the "death" time to be when the next interrogator arrives at the system. Thus, it is a measured lifetime, because it is with respect to an interrogator discovering that the message is no longer alive. We normalize the lifetime by dividing by k message bytes, for a fair comparison of different coding schemes.

We are also interested in the **message access time per byte**, which is similar to that in Section III-C1. However, in this case we do include the actual message data transfer time in the metric, since there are differences in message sizes.

As before, tags use a 96-bit unique ID. Timestamps are 17 bits long and N requires 3 bits. We take $q = 32$ chunks, and vary $k \in \{16, 20, 24, 28\}$. Therefore, chunk sequence numbers require $\log_2 q = 5$ bits. The actual message data for each chunk is 1 byte = 8 bits. We use timestamps as unique message IDs. Therefore, each chunk, along with the sequence number and message ID, requires $8 + 5 + 17 = 30$ bits. A typical tag (such as the Alien Higgs-3 family [16]) has 512 bits of user storage. So we take each tag to have space for $\lfloor 512/30 \rfloor = 17$ storage slots.

2) Simulation results and discussion: We simulate the average measured lifetime of a message per byte in steady state. We plot this against the expected tag population size $E[n] = \frac{\lambda_T}{\mu_T}$. We take $\lambda_T = 1$, and vary $\mu_T \in \{\frac{1}{100}, \frac{1}{200}, \ldots, \frac{1}{1000}\}$. Results are shown in Fig. 3. As expected, as interrogators come more often (λ_I large), message lifetimes are reduced, since chunks are overwritten more quickly. We see that increasing k reduces the per byte lifetimes. That is, the coding buffer per byte of $\frac{q-k}{k}$ bytes is reduced, and messages are destroyed sooner.

We simulate the average message access time per byte in steady state, for $\lambda_I = 0.2$ and $p = \frac{1}{2}$. Results are shown in Fig. 4. Increasing k improves performance. However, this is only when the message is still alive. The points in Fig. 4 only average the simulation iterations where there are still at least k message chunks in the tags. (As already discussed above, the average measured message lifetime per byte is small when k and λ_I are large.) In the most extreme case, when $k = 28, E[n] = 100$, and $m = 5$, the message is not alive in 74% of the simulation iterations. For $k = 16$, only the $E[n] = 100$ cases have a non-zero percentage (and just less than 8%) of not being alive. In other words, Figs. 3 and 4 together show a tradeoff, summarized below. That is, we cannot have both long per byte lifetimes and short per byte access times with the same system parameters.

k	**Coding buffer**	**Performance metrics (per byte)**	
		Message lifetime	Message access time
small	large	long	long
large	small	short	short

We see in Fig. 4 that query tree-max performs the worst. It requires two stages to operate, and is thus slow. Aloha-normal is better, and query tree-normal is the best. These two "normal" schemes are bad in the static RFID system because they singulate all the tags initially. However, in the dynamic system, since message chunks are spread over many tags, it is actually advantageous to first find the IDs of the all the tags, and then do message recovery. This is the key difference between the static and dynamic cases.

V. Conclusion and Future Work

In this work, we investigate many-tag access algorithms for RFID storage systems. We borrow ideas from traditional singulation algorithms, but modify them to make tag access efficient in systems with many tags. We study both static and dynamic situations. In the static case, query tree-max provides very good performance. The aloha-based algorithms perform not as well, but are more robust. In the dynamic situation, tags are continually arriving and departing. We use message encoding and spread chunks across tags to combat tags departing and being overwritten. Therefore, when reading (and recovering) messages, we need to find the IDs of a large number of the tags. The best algorithms in this case require first singulating all the IDs of the tags (which was a naive strategy in the static case).

In this work, we focused on reading the newest messages, and overwriting the oldest ones. In future work, we consider random access of information, keyed to a filename, for example. This requires more sophisticated algorithms and requires more overhead in terms of access time and tag storage. In particular, we need robust distributed data structures, to keep track of information in the tags.

References

[1] Z. Li, R. Gadh, and B. S. Prabhu, "Applications of RFID Technology and Smart Parts in Manufacturing, in *ASME Proc. Design Engineering Technical Conferences (DETC)*, Salt Lake City, UT, Sep.-Oct. 2004.

[2] T. Inaba, "Value of Sparse RFID Traceability Information in Asset Tracking during Migration Period, in *Proc. IEEE International Conference on RFID (RFID)*, Las Vegas, NV, Apr. 2008, pp. 183-190.

[3] D. Hahnel, W. Burgard, D. Fox, K. Fishkin, and M. Philipose, "Mapping and Localization with RFID Technology, in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, New Orleans, LA, Apr. 2004, vol. 1, pp. 1015-1020.

[4] A. Kleiner, J. Prediger, and B. Nebel, "RFID Technology-based Exploration and SLAM for Search and Rescue, in *Proc. IEEE International Conference on Intelligent Robots and Systems (IROS)*, Beijing, China, Oct. 2006, pp. 4054-4059.

[5] J. Bohn and F. Mattern, "Super-distributed RFID Tag Infrastructures, *Lecture Notes in Computer Science*. Berlin, Germany. Springer, 2004. vol. 3295.

[6] V.K.Y. Wu and N.H. Vaidya, "RFID Trees: A Distributed RFID Tag Storage Infrastructure for Forest Search and Rescue," in *Proc. IEEE Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON)*, Boston, MA, Jun. 2010, pp. 253-260.

[7] V.K.Y. Wu and N.H. Vaidya, "Exploiting Space-Time Correlations in an RFID Tag Field for Localization and Tracking," in *Proc. IEEE Global Communications Conference (GLOBECOM)*, Miami, FL, Dec. 2010. [To appear]

[8] A. Shibayama, Y. Hisada, M. Mirakami, M. Endo, S. Zama, O. Takizawa, M. Hosokawa, and T. Ichii, "A Study on the Disaster Information Collection Support System, Incorporating Information and Communication Technology," in *Proc. World Conference on Earthquake Engineering*, Beijing China, Oct. 2008.

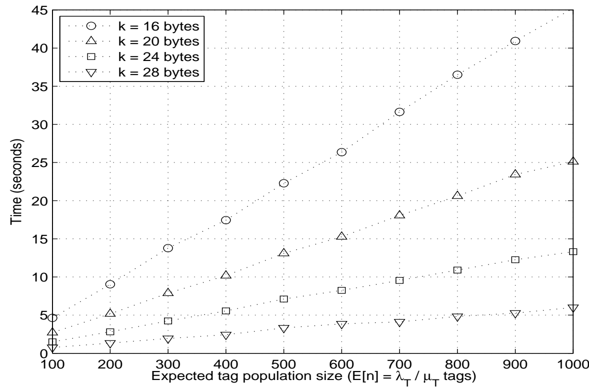
[9] K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets," in *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Karlsruhe, Germany, 2003.

[10] Z. Yang and H. Wu, "Featherlight Information Network with Delay-Endurable RFID Support (FINDERS)," in *Proc. IEEE Conference on Sensor, Mesh, and Ad Hoc Communications and Networks*, Rome, Italy, Jun. 2009.

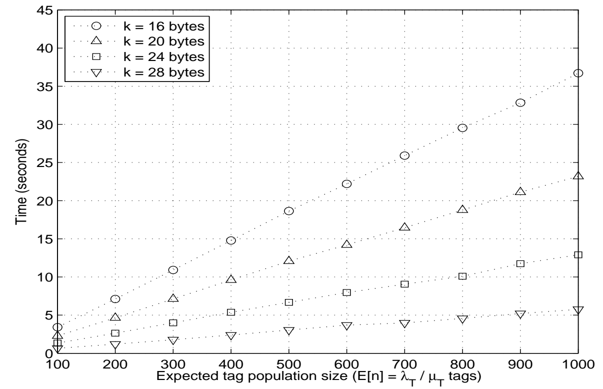
[11] D.W. Engels, "The Reader Collision Problem," *Whitepaper*, Sep. 2006.

[12] H. Vogt, "Multiple Object Identification with Passive RFID Tags," in *Proc. IEEE Conference on Systems, Man, and Cybernetics (SMC)*, Hammamet, Tunisia, Oct. 2002.

[13] C. Law, K. Lee, and K.-Y. Siu, "Efficient Memoryless Protocol for Tag Identification," in *Proc. ACM International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, Boston, MA, Aug. 2000, pp. 75-84.

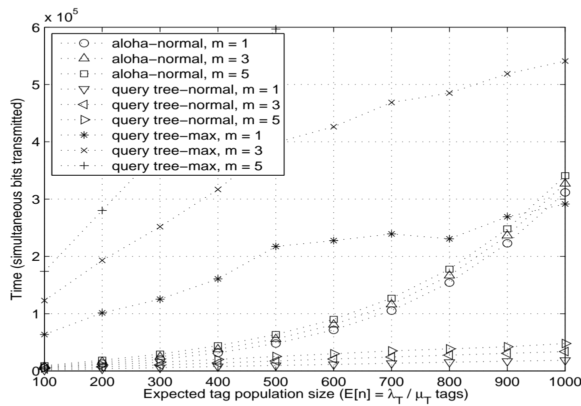


(a) $\lambda_I = 0.2$ arrivals per second

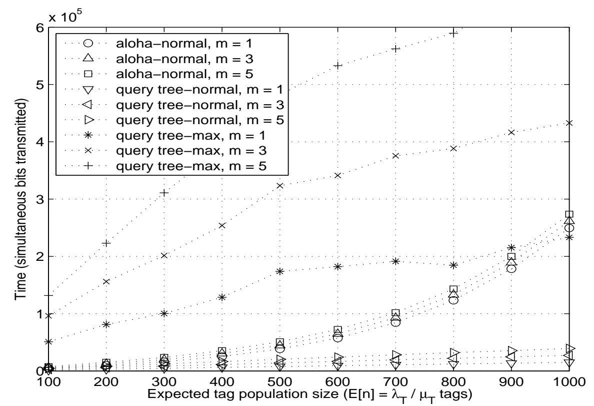


(b) $\lambda_I = 0.6$ arrivals per second

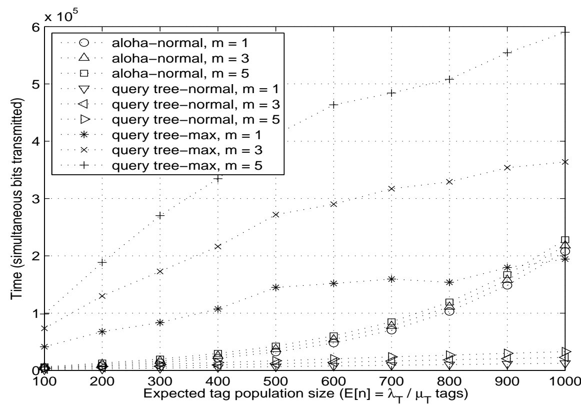
Fig. 3. Average measured message lifetime per byte, $p = \frac{1}{2}$



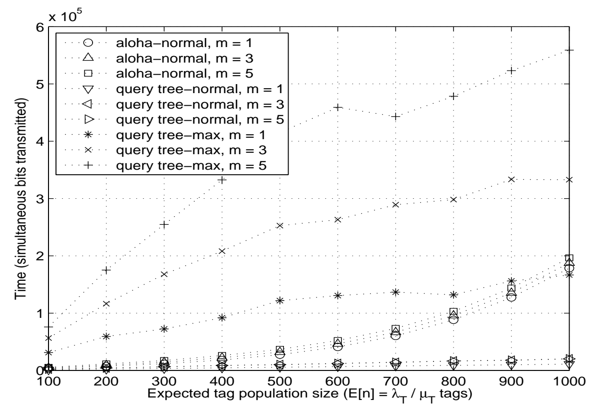
(a) $k = 16$ bytes



(b) $k = 20$ bytes



(c) $k = 24$ bytes



(d) $k = 28$ bytes

Fig. 4. Average message access time per byte, $\lambda_I = 0.2$ arrivals per second, $p = \frac{1}{2}$

[14] "EPCglobal UHF Class 1 Gen 2, *EPCglobal*. [Online]. Available: http://www.epcglobalinc.org/standards/uhfc1g2.
[15] S.B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, USA, 1994.
[16] "Alien RFID ICs," *Alien Technology*. [Online]. Available: http://www.alientechnology.com/tags/rfid_ic.php