2011 Workshops of International Conference on Advanced Information Networking and Applications

Tracing Moving Objects in Internet-based RFID Networks

Yanbo Wu, Quan Z. Sheng, Damith Ranasinghe
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
Email: {yanbo, qsheng, damith}@cs.adelaide.edu.au

***Abstract*—With recent advances in technologies such as radio-frequency identification (RFID) and new standards such as the electronic product code (EPC), large-scale traceability is emerging as a key differentiator in a wide range of enterprise applications. Such traceability applications often need to access data collected by individual enterprises in a distributed environment. Centralized approaches are not feasible for these applications due to their characteristics such as large volume of data and sovereignty of the participants. Inspired by IBM's Theseos approach, we have developed novel techniques enabling applications to share traceability data across independent enterprises in a pure Peer-to-Peer (P2P) fashion. In this paper, we present a generic approach for efficiently indexing and locating individual objects in large, distributed traceable networks. In particular, data are stored in local repositories of participants and indexed in the network based on structured P2P overlays.**

I. Introduction

Traceability refers to the capability of an application to track and trace the state (e.g., location) of an object, including discovering the information of its current and past state, as well as estimating the information of its future state. Traceability is essential to a wide range of important business applications such as manufacturing control, logistics of distribution, product recalls, and anti-counterfeiting [1], [7], [8].

Recent advances in sensor and RFID (radio-frequency identification) technologies make automatic tracking and tracing possible in large-scale applications (e.g., nation-wide supply chain management across companies). An emerging technique that targets large-scale traceability is the so called "Networked RFID" [7]. The basic idea behind the Networked RFID is to realize a "data-on-network" system, where RFID tags contain an unambiguous ID and other data pertaining to the objects are stored and accessed over the Internet. With "Networked RFID", traceability applications analyze automatically recorded identification events to discover the current location of an individual item. They can also retrieve historical information, such as previous locations, transportation time between locations, and time spent in storage.

Central to realizing these traceability applications is the management of data, particularly on how to share the data that are typically collected by individual enterprises. An obvious solution is to publish all data collected within each organization to a central data warehouse. Unfortunately, this approach has several severe drawbacks. Firstly, object movement and related data are valuable business information that companies may be very reluctant to put their data in a shared central warehouse. Secondly, such an approach has very limited scalability and is not feasible for large-scale applications where the amount of data collected could be enormous [1], [8].

An alternative would be a *peer data management system* (PDMS) that allows data to be stored in local repositories at each organization and makes those repositories accessible in the P2P network [15].

A recent representative effort in this direction is IBM's Theseos that enables traceability applications to process complex queries across organizations in a completely distributed setting. Theseos relies on a novel traceability data model that eliminates any data dependencies between organizations, which serves as a global schema that allows the formulation of a query without knowledge on how the data is stored, where it is located, and how a tracking query is executed [1]. To improve the performance of traceability query processing, Theseos introduces two attributes in its data model, namely *sentTo* and *receivedFrom* (We call the attributes the *information of object path* (IOP) [9]), that each organization is required to maintain the information on the movement path of an object. With this information, it is possible to minimize the number of nodes to be visited without flooding queries to all nodes in the network. Unfortunately, to obtain this information, Theseos requires high synchronization with other enterprise data (e.g., billing or accounting information). This is impractical for many applications where such enterprise data may be unavailable.

In this paper, we propose a generic approach to solve this problem, which is built on top of the DHT (Distributed Hash Table) based overlay network. We index an object and its latest state at a deterministic node called *gateway* node. Every time the object moves from a *source* node to a *destination* node, the gateway node updates object status at the source and the destination nodes of its movement, thereby establishing the information on movement path of the object. To reduce the indexing overhead from large volume data in traceability applications, we further propose an enhanced adaptive group-based indexing mechanism. Extensive experiments have been conducted to show the viability and scalability of our proposed techniques.

The remainder of the paper is organized as follows. Section II gives an overview of the related work. Section III

978-0-7695-4338-3/11 $26.00 © 2011 IEEE
DOI 10.1109/WAINA.2011.99

briefly introduces traceable networks, a moving object model, and traceability queries. Section IV describes a P2P approach for efficient processing of traceability queries in a DHT-based overlay network. Section V presents an enhanced indexing approach dealing with massive volumes of data in large-scale traceability applications. Finally, Section VI reports the results of the evaluation of the proposed techniques and Section VII provides some concluding remarks.

II. Related Work

The first step of modeling moving objects is to abstract the basic elements such as time, region and velocity. [10] introduces a data model called MOST (Moving Objects Spatio-Temporal) for databases with dynamic attributes, i.e. attributes that change continuously as a function of time. Later works mostly use the same or similar idea.

With the basic elements modeled, it is possible to answer basic queries such as location of an object at a certain time. In order to answer more complicated queries, such as an aggregate query "how many objects are in region $\mathcal{R}$ now?" and a future state query "where will the object $\mathcal{O}$ be after one hour?", various complex and domain-specific models have been developed. For example, in [12], adaptive multi-dimension histogram (AMH) is used in answering aggregate queries about past, present and future. Since this kind of work focuses on aggregate queries, it does not address the single-instance queries. Secondly, most of the works define region or similar concepts that cover a finite area in a continuous infinite domain in order to answer range queries.

Index is often used to quickly answer range queries for a specific attribute. However, dividing the space into fixed-size cells does not work well in dynamic environments because the boundary of the space must be decided in advance. Early works, including R-Tree, R*-tree, TR-tree, TB-tree, TPR-tree, TPR* R-tree and some other similar trees [6], dynamically decide which point or region to index and optimize the indexing process in different ways. Many recent works such as [3], [14] focus on other specific problems and provide corresponding solutions. These index methods all operate in centralized environments.

[13] extends Quadtree index into a P2P network to answer range queries efficiently while keeping the load on the nodes in the network well balanced. [5] provides a linear yet distributed structure that facilitates multiple search paths to be mixed together by sharing links. These distributed index methods are all based on spatial elements such as point, link or region thus they work well for range queries but not single-instance queries.

III. Preliminaries

In this section, we will briefly introduce traceable networks and a new moving object model.

A. Traceable Network

Figure 1 gives a high level description for *traceable networks*. There are three main entities in this network: *node*, *receptor*, and *object*. Each node governs a number of receptors

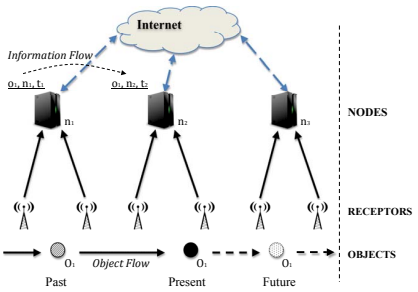


Fig. 1: Traceable Network

(e.g., RFID readers) that are used to capture the objects. Receptors are deployed at fixed locations (e.g., entry of a warehouse). The nodes are logical and abstracted to represent a partner in the network. Nodes store the *information flow segments* which occur inside their own territory. Nodes are connected (e.g., via the Internet) so that they can exchange information with other nodes in order to get the whole picture of an object's trajectory.

In the sequel, we will introduce a traceable network model and discuss some important traceability queries.

B. A Model for Moving Objects in Discrete Space

The traditional continuous models are unnecessarily heavy because of the definition and maintenance of various spatial elements. We therefore define a new model that can represent moving objects and their attributes in an economic way, namely *MOODS* (a Model for mOving Objects in Discrete Space).

$$\mathbb{L}(o,t) : \mathbb{O} \times \mathbb{T} \mapsto \mathbb{N} \tag{1}$$

Equation 1 shows the signature of the model. Given an object o in the object set $\mathbb{O}$ and a point of time t in the time domain $\mathbb{T}$, the locating function $\mathbb{L}$ derives the location where the object o was/is/will be at t.

In our model, the time domain $\mathbb{T}$ is continuous because receptors are working continuously to capture moving objects. However, as we mentioned above, the space domain is discrete. Instead of a continuous infinite domain, *MOODS* defines the space domain as a finite set of *nodes* $\mathbb{N} = \{n_1, n_2, ..., n_m\}$. This set is known to the system because they are the locations where the receptors are deployed. The set is dynamic since new nodes can join and existing nodes may leave the network. The object domain is the same as the existing models, i.e. $\mathbb{O} = \{o_1, o_2, ..., o_n\}$.

The function $\mathbb{L}$ essentially locates an object. Another important function is to find the trace of an object. Similar to $\mathbb{L}$, we define the trace function $\mathbb{TR}$ as:

$$\mathbb{TR}(o, t_{start}, t_{end}) : \mathbb{O} \times \mathbb{T} \times \mathbb{T} \mapsto \mathbb{P} \tag{2}$$

$$\mathbb{P} : \{(n_{k_1}, n_{k_2}, ..., n_{k_p}, \ldots, n_{k_l}) | n_{k_p} \in \mathbb{N}, 0 \leq l \leq m\} \tag{3}$$

Given an object o and a time range (t_{start} to t_{end}), $\mathbb{TR}$ finds the trace of o during that timeframe. $\mathbb{P}$ denotes the domain of *path*. A path is a sorted list of nodes (can be empty). The sorting is done by the order of the nodes visited by object o (i.e., by time).

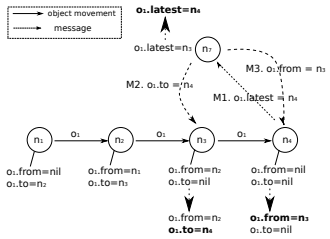


Fig. 2: Example of IOP Acquisition

IV. A P2P Approach for Traceable Networks

In this section, we will introduce a P2P approach that can process traceability queries in an effective and efficient way.

The main idea of our approach is to index an object and its latest location in the DHT network, while keeping the IOP information at the nodes where the object has been observed. Instead of a centralized index server, we store the index for an object at the node which is the result of a DHT lookup for the object's id. IOP contains information about the source (i.e., where the object comes from) and destination (i.e., where the object leaves to) nodes, thus it can be used to quickly answer trace queries. The place where the index is stored is determined solely by the id of the object thanks to DHT's determinism, so that from anywhere in the network, the object can be located by its id. The abstract data structure is essentially a double linked list distributed in the DHT network. The head of the list is the node where the latest location is indexed, which we call a *gateway node*. When the object moves to a new node, the gateway node is updated and the list is expanded. In our work, we adopt Chord [11] as the overlay for its adaptiveness as nodes join and leave. Figure 2 illustrates our idea with an example.

The node n_7 is the gateway node for the object o_1. It stores the latest location of o_1. In this case, before o_1 arrives at node n_4, it was captured at node n_3. After it arrives and is captured by a receptor governed by n_4, n_4 sends a message (M1) to n_7 by the P2P lookup interface, telling n_7 that o_1 has arrived at n_4. Upon receiving this message, n_7 updates its index and sends messages to n_3 and n_4. The message to n_3 (M2) indicates that o_1 arrives at n_4, so n_3 updates its IOP by setting $o_1.to$=n_4. The message to n_4 (M3) indicates that o_1 was from n_3, so n_4 updates its IOP by setting $o_1.from$=n_3. In this way, the IOP is established and the index is updated.

To answer queries about a specific object, the first step is to find out its latest location by querying the corresponding gateway node. We can then trace back the list and ask each node along the list for the desired information.

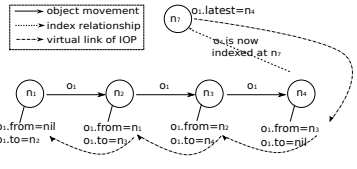


Fig. 3: Distributed Linked List

A problem of our IOP acquisition algorithm is the indexing cost. Whenever an object arrives at a node, the system has to issue 3 messages (1 message from the node and 2 messages from the object's corresponding gateway node) in order to establish the IOP links. This is clearly an expensive approach for large-scale applications where order of magnitude is often thousand, and objects often move in groups. In order to solve this problem while still have the IOP links established, we next introduce an enhanced algorithm.

V. Enhanced Indexing and Query Processing

In a large traceable network, the data volume might be very high. Indexing each individual object will cause enormous number of messages to flood the network. An obvious solution to solve this problem is to classify the objects arriving at a node within a small period of time into different groups. In particular, objects' raw ids can be hashed using the secure hash algorithm SHA-1 and grouped by the prefixes of their hashed ids. For example, if 1024 objects arrived at a node n and we choose a prefix length of 4, there are at most 2^4=16 prefixes. As a consequence, instead of indexing all these 1024 objects, we simply classify them into at most 16 groups by prefixes, and index the groups. Based on this idea, we introduce our enhanced IOP acquisition algorithm in the following sections.

1) Group Generation: A *group* function is invoked periodically at time intervals of $\mathcal{T}_{interval}$[1]. Two objects belong to the same group when their ids have $\mathcal{L}_p$ prefix bits in common.$\mathcal{L}_p$, is the key to determine the total number of groups in the system and should be chosen wisely. If it is too big, the number of groups is high and in the worst case, it is close to the number of objects. As a result, the number of messages will not be significantly decreased. On the other hand, if it is too small, only a small portion of the nodes in the network will be responsible for indexing, thus the work load is not well balanced. Essentially, it is important to find an *optimal* value of $\mathcal{L}_p$ that can guarantee that *almost* every node in the network has the opportunity to index at least one prefix (i.e., group). We denote the probability that a node has at least one group to index as δ. Because prefixes are distributed uniformly at random over the nodes by the hash function, according to the probabilistic theory, we have:

$$\delta = 1 - \left(\frac{\mathcal{N}_n - 1}{\mathcal{N}_n}\right)^m \tag{4}$$

Where m is the number of groups to distribute, and it is a function of the length of prefix: $m = 2^{\mathcal{L}_p}$. The optimal value of $\mathcal{L}_p$ should yield the value of δ as close as possible to 1. In fact, when $m \leftarrow \mathcal{N}_n \log_2 \mathcal{N}_n$, we have:

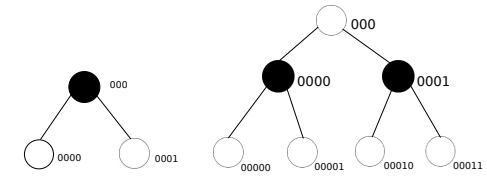
$$\lim_{\mathcal{N}_n \to \infty} \delta = \lim_{\mathcal{N}_n \to \infty} \left(1 - \left(\frac{\mathcal{N}_n - 1}{\mathcal{N}_n}\right)^{\mathcal{N}_n \log_2 \mathcal{N}_n}\right) = 1 \tag{5}$$

Thus we can determine the $\mathcal{L}_p$ value:

$$\mathcal{L}_p \geqslant \log_2 (\mathcal{N}_n \log_2 \mathcal{N}_n) = (\log_2 \mathcal{N}_n + \log_2 \log_2 \mathcal{N}_n) \tag{6}$$

According to Equation 6, we choose $\lceil \log_2 \mathcal{N}_n + \log_2 \log_2 \mathcal{N}_n \rceil$ for $\mathcal{L}_p$. As our experiments show (see Section VI), the value we have chosen is good enough to keep the performance while ensuring good load balancing.

[1]The value of $\mathcal{T}_{interval}$ can be determined by the system's timeliness restriction and is configurable



(a) A triangle for indexing objects with prefix '000'.The current prefix length is 3.

(b) A tree extended for the triangle in (a). The two nodes indexing '0000' and '0001' become parents for another four nodes indexing the four indices indicated in the last layer.

Fig. 4: Example of Data Triangle

As new nodes join and existing nodes leave, $\mathcal{N}_n$ is *dynamic*. As a result, there is no precise way to calculate this value. However, there are some algorithms available to estimate the value of $\mathcal{N}_n$. We will not explain these algorithms here. Interested readers are referred to [4] for details.

2) Algorithms for Indexing and Index Persistence: After the groups are built, the node that captured objects will send indexing messages to the gateway node for each group. The gateway node is determined by the hash value of the group id.

The IOP update process is almost the same as that in Section IV except instead of sending one message for each object, we send one message for a group of objects which are from the same node.

However, we cannot simply store and lookup the index at the gateway node. There are two issues we need to address because of the introduction of the grouping scheme.

Firstly, changes of $\mathcal{L}_p$ cause grouping inconsistencies. For example, at node n_1, $\mathcal{L}_p$ was 2 and we grouped objects 0000 and 0001 into the same group 00. The index was stored at gateway node *hash("00")*. Before these objects arrived at node n_2, new nodes joined that caused $\mathcal{L}_p$ to become 3. After objects 0000 and 0001 arrived, they are grouped into group 000 and the corresponding gateway node becomes *hash("000")*. At this moment, the node *hash("000")* does not have previous information about these two objects and it may assume that node n_2 is the first node for the trace of these objects, which in fact should be n_1. A similar situation also happens when the size of the network decreases. This causes the IOP information to be updated incorrectly.

Secondly, once $\mathcal{L}_p$ is fixed, gateway nodes are also determined. If $\mathcal{L}_p$ does not change and nodes do not leave the network, the gateway nodes remain the same. Consequently new nodes have no chance to index any group. The load is then not well balanced.

To solve the above two problems, we introduce a distributed data structure called *Data Triangle*. A data triangle consists of three nodes in the network. Figure 4 (a) shows an example of the triangle. The gateway node is responsible for indexing objects whose ids are prefixed by "000" (current prefix length is 3). The two child nodes are responsible for indexing "0000" and "0001". But the child nodes do not index groups directly, instead they are *secondary storage* for their parent. The parent delegates part of the indexing data according to the *next bit* in an object id after the prefix to the child nodes respectively, thereby keeping the workload between the three balanced.

Suppose to increase the current $\mathcal{L}_p$ by 1, we need $\Delta\mathcal{N}_n$ new nodes to join. We have:

Algorithm : Index

Input: prefix p with length $\mathcal{L}_p$
and the set of $objects$ belong to the group $hash(p)$
Output: None (the algorithm only updates indexing information)

update local index with $local.objects \cap objects$
$objects' \leftarrow local.objects - objects$
// get the set of objects which are not stored locally
if $objects'$ is not empty
$refresh\_from\_ascent(objects', p)$
$refresh\_from\_descent(objects', p)$
$update\_index(objects', p)$
end if

refresh_from_ascent

$i \leftarrow 1$
do
$p' \leftarrow p.sub(1, \mathcal{L}_p - i)$
$objects' \leftarrow filter(objects, p')$
if $objects' = \Phi$ **break**
$objects_{parent} \leftarrow fetch(objects', p')$
update local index with $objects_{parent}$
$i \leftarrow i + 1$
while p' exists and $i \leq \mathcal{L}_p - \mathcal{L}_{min}$

refresh_from_descent

$objects' \leftarrow filter(objects, p$ + '0')
if $objects'$ is not empty
$objects_{child0} \leftarrow fetch(objects', p$ + '0')
update local index with $objects_{child0}$
$refresh\_from\_descent(objects, p$ + '0')
end if
$objects' \leftarrow filter(objects, p$ + '1')
if $objects'$ is not empty
$objects_{child1} \leftarrow fetch(objects', p$ + '1')
update local index with $objects_{child1}$
$refresh\_from\_descent(objects, p$ + '1')
end if

update_index

update local index with $objects$
if delegation is required
select $\alpha * objects.count$ earliest local indexing records as $objects'$
delegate $objects'$ to the two children according to the $\mathcal{L}_p$ bit of their ids

Fig. 5: Algorithm to Index a Group of Objects

$$\lceil (\mathcal{N}_n + \Delta\mathcal{N}_n) log_2(\mathcal{N}_n + \Delta\mathcal{N}_n) \rceil - \lceil \mathcal{N}_n log_2 \mathcal{N}_n \rceil = 1 \quad (7)$$

It is easy to prove that $\Delta\mathcal{N} < \mathcal{N}_n$. In the worst case, there are less than $\mathcal{N}_n$ nodes idle. For a network whose global prefix length is $\mathcal{L}_p$, we use extra $2^{\mathcal{L}_p+1}$ logical nodes (the child nodes) for indexing. The $2^{\mathcal{L}_p+1} + 2^{\mathcal{L}_p} = 3\mathcal{N}_n \log_2 \mathcal{N}_n$ logical nodes are distributed among at most $\Delta\mathcal{N}_n + \mathcal{N}_n < 2\mathcal{N}_n$ physical nodes. By similar inference to Equation 6, there is a high probability that all physical nodes have at least one group to index.

With the changes to the network (nodes join and leave), the child nodes may have their children too. Eventually, the indexing will be distributed in a tree structure (see Figure 4 (b)). However, the tree is not necessary. From the analysis above, we can see that the data triangle is good enough to maintain a well balanced workload. Maintaining a tree introduces longer delay for looking up.

To solve this problem, we start a *splitting-merging* process when the prefix length changes. If the prefix length increases, the data stored in the old parent are be delegated into the two new parent nodes which are its child nodes. Similarly, if

the prefix length decreases, the parent node's two child nodes migrate the data they are indexing to the parent node.

Figure 5 shows our indexing algorithm. It should be noted that the splitting-merging process is trivial and we have not included the details of the algorithm.

index. The algorithm *index* refreshes the local indexing records by searching up and down in the tree (lines 5 and 6) for the objects in the difference set. After the index of all the objects are downloaded to the local storage, or the tree has been fully traversed, the index is updated with the new information (line 7).

refresh_from_ascent and refresh_from_descent. The two procedures are used to retrieve the index by traversing the tree up and down, respectively. Before sending the fetching request to the respective nodes, the object list is filtered by the prefix (the filter function in line 4 of $refresh\_from\_ascent$ and lines 3, 9 of $refresh\_from\_descent$) for pruning.

update_index. The system first updates local indexes with the $objects$ to be indexed (line 1). Then it checks whether it is necessary to delegate some records to the two child nodes (line 2). There can be different strategies to determine this. If a delegation is required, we select the earliest $\alpha * objects.count$ ($0 < \alpha \leq 1$) objects indexed at this gateway and delegate them to the two child nodes. The delegation is similar to the *FIFO* cache replacement policy. This is based on the observation that the latest records are more likely to be read and updated in the near future. Here α is a global configuration to control the number of objects to be delegated.

3) Lookup Algorithm: Because the index is stored in a tree structure, the $lookup$ function call should be performed in the tree instead of the local database. We first try to find it in the gateway node for the prefix of current length. If it cannot be found, we try a bidirectional linear search starting from the gateway node. The algorithm is formally defined in Figure 6.

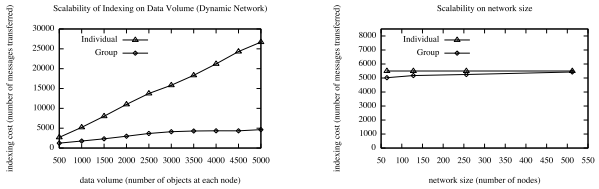
Algorithm: Lookup	
Input: the id of an object obj	
Output: the indexing information of object obj	
1:	**if** obj is stored locally
2:	return its indexing information
3:	$i \leftarrow 1$
4:	$p = obj.id.sub(1, \mathcal{L}_p - i)$
5:	**while true**
6:	$ret =$ lookup obj at $node(hash(p))$
7:	**if** ret is not nil return ret
8:	**else**
9:	$i++, p = obj.id.sub(1, \mathcal{L}_p + i)$

Fig. 6: Algorithm to Lookup an Object

VI. Experiments

Firstly, we evaluated the performance of the indexing algorithms and compared the scalability of the individual indexing approach and the enhanced group indexing approach. Secondly, we studied the performance of query processing using these techniques.Finally, we examined the effect of the prefix length ($\mathcal{L}_p$) on load balancing and indexing performance.

We used the open source P2P simulator OverSim [2] in the experiments. In our tests, the maximum number of nodes is 512 and the maximum number of objects at each node is 5000, corresponding to the limit of our experimental environment.



(a) on Data Volume (b) on Network Size

Fig. 7: Scalability of Indexing

A. Indexing

The indexing cost, measured by the total volume of messages transferred over the network, is considered to study the feasibility of the proposed group indexing algorithm and related data structures.

We first studied the indexing cost to verify that the architecture is scalable.In this test, we continuously added new nodes into the network until the size of the network reached 512. When a new node joined the network, a specific number of objects were generated on the node and the indexing process started. After a random period of time, the new node moved 10% of the objects to a randomly chosen destination. We collected the total volume of messages transferred during the whole process. We ran the test 10 times and for the i_{th} ($1 \leq i \leq 10$) time, the number of objects generated at each node is $500 * i$. As we can see from Figure 7a, the indexing cost of group indexing algorithm increases much slower than the individual indexing algorithm.

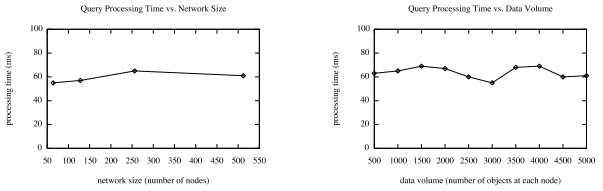
We also studied the performance of the indexing algorithms under different network sizes. In the experiment, the number of objects generated at each node is fixed to 5000 and the network size varies from 64, 128, 256, to 512. The result in Figure 7b shows that with the increase in network size, the indexing cost for the group indexing algorithm increases while the individual indexing algorithm shows a horizontal line (due to the fixed data volume). The reason is that when the network size increases, $\mathcal{L}_p$ increases as well, which leads to an increase in the number of groups ($2^{\mathcal{L}_p}$) and the messages to be transferred. Generally, the performance degrades when the ratio of data volume to network size becomes small. As we can see, when the size of network increases, the indexing cost for the group indexing algorithm becomes closer to that for the individual indexing algorithm. However in reality, particularly large-scale applications, there are much more objects than nodes. These applications can take advantage of the benefits brought by our proposed group indexing approach.

B. Query Processing

In this experiment, we studied the performance of our overall approach for query processing under different network sizes and different data volumes. We used a typical query for tracing individual objects in the study.

Figure 8a shows the query processing time for tracing an object in networks of different sizes. From the figure we can see that the query processing time is almost constant. Figure 8b

shows the query processing time for different data volumes. Similar result was obtained (i.e., the time used is almost constant). The high scalability shown in the experimental results is due to the IOP information in our design.



(a) vs. Network Size (b) vs. Data Volume

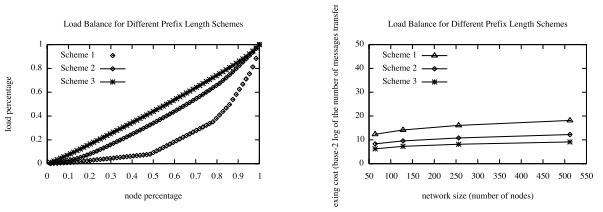
Fig. 8: Query Processing

C. The Effect of $\mathcal{L}_p$

In this experiment, we studied the different schemes of $\mathcal{L}_p$ and tested their corresponding load balancing capabilities. Figure 9a shows the result for the three different schemes (i.e., Scheme 1: $\mathcal{L}_p$=$\log_2 \mathcal{N}_n$, Scheme 2: $\mathcal{L}_p$=$\log_2 \mathcal{N}_n$ + $\log_2 \log_2 \mathcal{N}_n$, and Scheme 3: $\mathcal{L}_p$=$2 \log_2 \mathcal{N}_n$). Scheme 2 is the one that we have chosen for our system (which is also the scheme used in all other experiments).

We illustrate the load balance by showing the load percentage (i.e., the number of objects handled by a given set of nodes divided by the total number of objects) for a given node percentage (i.e., the number of nodes in the set divided by the total number of nodes). A well balanced scheme should yield a linear relationship between the load percentage and the node percentage (where $y = x$), meaning that each node receives the same number of objects to index. The farther the curve is away from the diagonal, the worse it is.

As we can see from Figure 9a, when Scheme 1 was chosen as the length of prefix $\mathcal{L}_p$, the load is not well balanced. Scheme 3 performs best among the three. However, Scheme 3 makes $\mathcal{L}_p$ too long and the number of groups becomes too big, leading to less objects in each group. This significantly affects the indexing cost (see Figure 9b). Scheme 2 provides a good choice for $\mathcal{L}_p$, with which the work load is also well balanced.



(a) to Load Balancing (b) to Indexing Cost

Fig. 9: The Effect of Prefix Length

We also tested the performance of the three schemes with different network sizes at fixed data volume (5000 objects at each node). Figure 9b shows that among the three, Scheme 1 seems to be the most efficient one. Scheme 3 is the worst (its indexing cost is the square of the number of nodes). However, there is a tradeoff between indexing performance and load balancing. To improve the former, $\mathcal{L}_p$ should be smaller, which leads to poor load balancing. To improve the latter, $\mathcal{L}_p$ should be bigger to ensure that all nodes have groups to be responsible for, which leads to a higher indexing cost. Scheme 2 shows acceptable results on both indexing performance and load balancing. In reality, we can choose different schemes for different scenarios.

VII. Conclusion

Recent advances in technologies such as radio-frequency identification (RFID) make automatic tracking and tracing possible in a wide range of applications. Unfortunately, realizing traceability applications in large-scale, distributed environments presents significant challenges due to their unique characteristics such as large volume of data and sovereignty of the participants. In this paper, we have presented a generic approach that enables applications to share traceability data across independent enterprises in a P2P fashion. To reduce the indexing overhead from massive volumes of data in large-scale traceability applications, we further proposed an enhanced group-based indexing approach. Extensive experiments showed the viability and scalability of our approach.

Ongoing work includes further performance study of the proposed techniques. Another important direction for future work is to add capabilities for predicting future status of objects. This typically involves overcoming uncertainty issues introduced by traceability applications using statistical and probabilistic techniques.

References

[1] R. Agrawal, A. Cheung, K. Kailing, and S. Schönauer. Towards Traceability Across Sovereign, Distributed RFID Databases. In *Proc. of IDEAS'06*, India.

[2] I. Baumgart, B. Heep, and S. Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proc. of GI'07*, USA.

[3] Y. Gao, G. Chen, Q. Li, B. Zheng, and C. Li. Processing Mutual Nearest Neighbor Queries for Moving Object Trajectories. In *Proc. of MDM'08*, USA.

[4] M. Jelasity and A. Montresor. Epidemic-Style Proactive Aggregation in Large Overlay Networks. In *Proc. of ICDCS'04*, USA.

[5] W.-C. Lee and B. Zheng. DSI: A Fully Distributed Spatial Index for Location-Based Wireless Broadcast Services. In *Proc. of ICDCS'05*, USA.

[6] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Engineering Bulletin*, 26:40–49, 2003.

[7] G. Roussos, S. S. Duri, and C. W. Thompson. RFID Meets The Internet. *IEEE Internet Computing*, 13(1):11–13, 2009.

[8] Q. Z. Sheng, X. Li, and S. Zeadally. Enabling Next-Generation RFID Applications: Solutions and Challenges. *IEEE Computer*, 41(9):21–28, September 2008.

[9] Q. Z. Sheng, Y. Wu, and D. Ranasinghe. Enabling Scalable RFID Traceability Networks. In *Proc. of AINA'10*, Australia.

[10] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. of ICDE'97*, Birmingham, U.K., 1997.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of SIGCOMM'01*.

[12] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present, and the Future in Spatio-Temporal Databases. In *Proc. of ICDE'04*, Boston, USA, 2004.

[13] E. Tanin, A. Harwood, and H. Samet. Using a Distributed Quadtree Index in Peer-to-Peer Networks. *The VLDB Journal*, 16(2):165–178, 2007.

[14] K. Tzoumas, M. L. Yiu, and C. S. Jensen. Workload-aware Indexing of Continuously Moving Objects. *PVLDB*, 2(1):1186–1197, 2009.

[15] S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan. Distributed Online Aggregations. *PVLDB*, 2(1):443–454, 2009.