Architectural Considerations for Distributed RFID Tracking and Monitoring

Zhao Cao, Yanlei Diao, and Prashant Shenoy

Department of Computer Science
University of Massachusetts Amherst

Abstract

In this paper we discuss architectural challenges in designing a distributed, scalable system for RFID tracking and monitoring. We argue for the need to combine inference and query processing techniques into a single system and consider several architectural choices for building such a system. Key research challenges in designing our system include: (i) the design of inference techniques that span multiple sites, (ii) distributed maintenance of inference and query state, (iii) sharing of inference and query state for scalability, and (iv) the use of writeable RFID tags to transfer state information as objects move through the supply chain. We also present the status of our ongoing research and preliminary results from an early implementation.

1 Introduction

RFID is a promising electronic identification technology that enables a real-time information infrastructure to provide timely, high-value content to monitoring and tracking applications. An RFID-enabled information infrastructure is likely to revolutionize areas such as supply chain management, health-care and pharmaceuticals. Consider, for example, a distributed supply chain environment with multiple warehouses and millions of tagged objects that move through this supply chain. Each warehouse is equipped with RFID readers that scan objects and their associated cases and pallets upon arrival and departure and while they are processed in the warehouse. In order to track objects and monitor the supply chain for anomalies, several types of queries may be posed on the RFID streams generated at the warehouses.

- **Tracking queries**: Report any pallet that has deviated from its intended path. List the path taken by an item through the supply chain.
- **Containment queries**: Raise an alert if a flammable item is not packed in a fireproof case. Verify that food containing peanuts is never exposed to other food cases for more than an hour.
- **Hybrid queries**: Report if a drug has been exposed to a temperature of more than 80 degrees for 12 hours.

The first class of queries are location queries that require object locations or location history. The second class involves containment, i.e., relationships between objects, cases and pallets. The third kind involves processing of sensor streams (e.g., temperature readings) in conjunction with RFID streams to detect certain conditions. Typically raw RFID streams contain noisy data that lacks any location or containment information. Hence, such continuous queries require derivation of location and containment information from raw RFID data as well as processing of heterogeneous sensor streams along with RFID data streams.

In this paper, we discuss the architectural challenges in designing a scalable, distributed stream processing system for RFID tracking and monitoring. We propose to combine location and containment inference with scalable query processing into a single architecture, in contrast to prior approaches that dealt with these two problems separately. We present three architectural choices in instantiating such a system over large supply chains and present an analysis of their communication overheads. By doing so, we argue that a distributed architecture is best suited for our goals.

In this paper, we also describe key technical challenges in designing a distributed architecture, which include (i) the design of novel inference techniques that span multiple warehouses of the supply chain, (ii) distributed, consistent maintenance of inference and query state as objects move through the supply chain, and (iii) sharing of inference and query state for scalability. A novel aspect of our system is its ability to exploit writeable RFID tags, when available, and use the onboard tag storage to transfer query and inference state as the object moves from one location to another. We finally present the status of our ongoing research and preliminary results from an early implementation.

2 Related work

**RFID stream processing.** The HiFi system [7, 9] offers a declarative framework for RFID data cleaning and processing. It focuses on per-tag smoothing and multi-tag aggregation, but does not capture relationships between objects such as containment or estimate object locations via containment. Our system can produce a rich event stream with object location and containment information. It further offers distributed inference and event processing methods.

**RFID databases.** Siemens RFID middleware[15] uses application rules to archive RFID data streams into databases. Cascadia [16] supports RFID-based pervasive
computing with event specification, extraction and archival. Techniques are also available to integrate data cleansing with query processing [12], encode flow information [10] and recover high-level information from incomplete, noisy data by exploiting known constraints [18]. These techniques, however, are not designed for fast RFID stream processing and only for centralized processing.

Event query processing. Most event processing methods [1, 11, 17] use centralized processing which requires all RFID data to be transferred to a single site, incurring high communication cost. The system in [2] uses multi-step event acquisition and processing to minimize event transmission cost. In contrast, our system performs both inference and query processing, does so at a local site whenever possible, and then transmits computation state across sites for distributed inference and event pattern detection.

3 Overview of the SPIRE System

In this section, we provide an architectural overview of a distributed RFID data management system, which we call SPIRE. This system is designed to support RFID-based tracking and monitoring in large-scale supply chains with multiple warehouses and millions of objects.

In a typical environment as depicted in Figure 1, each object is affixed with a tag that has a unique identity under the EPC standard [5]. Most tags are passive tags that have no individual power systems but small amounts of memory, e.g., 1-4 KB in the current generation of EPC tags [13] and up to 64 KB [8] in the next generation. Such tag memory can be used to store object information and facilitate query processing as we describe in the next section. Each RFID reader periodically sends a radio signal to tags in its read range; the tags use the radio energy to send back the tag id. The reader immediately returns the sensed data in the form of (tag_id, reader_id, time). The local servers of a warehouse collect raw RFID data streams from all readers, and then filter, aggregate, and process the data. The data streams from different warehouses are further aggregated to support global tracking and monitoring queries.

We next illustrate two tracking and monitoring queries mentioned earlier in the CQL language [3] extended with new constructs for event pattern matching [1, 17]. These queries assume that tuples in the input stream contain attributes (tag_id, time, location, container) and optional attributes that describe object properties such as the type of food and expiration date and can be obtained from the manufacturer’s database1. (Note the different schemas of raw RFID readings and tuples for query processing. Such differences motivate our discussion of inference below.)

Query 1 below is an example of containment queries. It sends an alert when peanut-free food has been contained in the same case as food containing peanuts for more than 1 hour. The inner (nested) query block performs a self-join over the input stream, where one join input retains only the tuples for peanut-free food, the other input retains only those for food containing peanuts, and the join is based on equality on container. Each join result represents an event that a container contains both food with and without peanuts. This event is published immediately into a new stream. The outer query block detects a sequence pattern over the new stream: each match of the pattern contains a sequence of events that refer to the same tag id of the peanut-free food and span a time period of more than 1 hour. For each pattern match, the query returns the tag id of the peanut free food and the length of the period. Such information can assist a retail store in deciding whether to dispose of the food.

Query 1:
```
From (Select Rstream(R1.tag_id, R1.loc)
From Food [Range 3 minutes] As R1,
Where R1.type = 'peanut free' and
R1.container = R2.container
) As S
[ Pattern SEQ(A+)
Where A[1].tag_id = A[1].tag_id and
A[A.len].time > A[1].time + 1 hr]  
```

Query 2 combines RFID readings with temperature sensor readings and alerts if an object has been exposed to a temperature of more than 80°C for 12 hours. It has a similar structure as Query 1, but returns all the sensor readings in the period when the temperature regulation was violated.

Query 2:
```
Select tag_id, A[].temp
From (Select Rstream(R.tag_id, R.loc, T.temp)
From Object [Now] as R,
Temperature [Rows 1 minute] as T
Where R.type = 'drug' and
T.temp > 80 °C and
R.loc ≃ T.loc
) As S
[ Pattern SEQ(A+)
Where A[1].tag_id = A[1].tag_id and
```

The SPIRE system we design, when presented in a centralized fashion, has two main functionalities: inference and query processing. The approaches to implementing them in a distributed environment are the focus of the next section.

---

1How to obtain the object properties to the site of query processing is an architectural issue, which we discuss in the next section.
Inference. While data stream management has been intensively studied recently, RFID data streams present new challenges to query processing:

Insufficient information. As the above examples show, query processing often requires information about object locations and inter-object relationships such as containment. However, raw RFID data contains only the observed tag id and its reader id due to the limitations of being an identification technology.

Incomplete, noisy data. The derivation of required information is compounded by the fact that RFID readings are inherently noisy, with read rates in actual deployments often in the 60%-70% range [9]. This is largely due to the sensitivity of radio frequencies to environmental factors such as occluding metal objects and interference [6]. Mobile RFID readers may read objects from arbitrary angles and distances, hence more susceptible to variable read rates.

In SPIRE, we design an inference module that derives object locations and containment relationships despite missed readings. We focus on containment in the following discussion (inference for object locations alone is detailed in our recent publication [14]). First, an RFID reader can read several containers and many items simultaneously, which makes it hard to infer exactly which item is contained in which container. In SPIRE, we explore the correlations of observations obtained at different locations at different times to infer containment. For instance, while the containment information at the loading dock of a warehouse is ambiguous due to the readings of two containers simultaneously, true containment may be revealed at the receiving belt that reads one container at a time. Such containment stays the same as containers are placed on shelves but may change later in the repackaging area. The inference algorithm needs to adapt to such changes in a timely fashion.

However, missing readings significantly complicate the inference problem. Consider a scenario that an item was last seen in location A with its container, but now its container is observed in B (but not the item). There are a few possibilities for the item: it was left behind in location A but the reading in A was missed; it moved to location B with its container and its reading in B was missed; it disappeared unexpectedly (e.g., stolen). The inference algorithm needs to account for all these possibilities when deriving containment and location information.

In SPIRE, we employ a time-varying graph model, as depicted in Figure 2(a), that uses nodes, edges, and statistics to encode for each object (i) its most recent observation, (ii) all of its possible containers, (iii) its co-location history with each of the containers, and (iv) its confirmed container in the past by a special reader like a belt reader (which may have become obsolete). All this information is collectively referred to as the inference state. SPIRE updates the graph as raw RFID readings arrive. Then an inference algorithm runs on each updated graph and estimates the most likely container and location of each object. Finally, when an object leaves an warehouse for the next, its inference state is transferred to the new place for subsequent inference. In other words, the inference state needs to be maintained across warehouses, i.e., on a global scale. We detail several architectural choices for doing so in the next section.

Query Processing. As the inference module streams out tuples with inferred location and containment information, the query processor, as shown in Figure 2(b), processes these tuples to answer continuous monitoring queries like the two examples above. As the figure shows, part of query processing can be performed at each warehouse, such as filtering of tuples based on object properties (Queries 1 and 2), a self-join over the input stream (Query 1), and a join between an object stream and a sensor stream (Query 2).

A more challenging issue is with the part of query processing that spans multiple warehouses, such as the detection of a complex pattern over a large period of time (see the pattern clause in Queries 1 and 2). Similar to inference, when an object leaves one warehouse for the next, the computation state for query processing, which we call the query state, also needs to be transferred to the new location. For a pattern query, a finite automaton is often used to govern the detection process. Then the query state includes (i) the current automaton state, (ii) the minimum set of values extracted from the input data that future evaluation requires, e.g., $A[1].\text{tag_id}$ and $A[1].\text{time}$ in Queries 1 and 2 (details for identifying the set of values are available in [1]), and (iii) the set of values that the query intends to return, which can be simple values as in Query 1, or a sequence of readings of finite yet unbounded size as in Query 2. If the pattern query is defined on a per-object basis, as in both of our example queries, the system needs to maintain a copy of query state for each object, as depicted by the copies labeled $O_1, \ldots, O_n$ in Figure 2(b). Finally, if a monitoring system supports multiple queries, the size of query state is further multiplied.
by the number of concurrent queries.

4 Architectural Choices: Benefits and Drawbacks

There are three possible choices for instantiating the system architecture presented in the previous section.

Centralized warehouse. This simplest approach is to employ a centralized architecture—similar to a centralized warehouse—where all RFID data is sent to a central location for stream processing, and possibly archival. The advantage of such a centralized approach is that the system has a global view of the entire supply chain, which simplifies stream processing. Inference is also simpler since all of the object state is maintained at a single location. In this case, the local servers depicted in Figure 1 only need to perform simple processing tasks such as cleaning and/or compression. The primary disadvantage of the approach is the high communication cost of transmitting RFID streams to the central location; since RFID data can be voluminous, the network bandwidth costs can be substantial.

Analysis: Consider a supply chain with 1000 warehouses, where each warehouse stores 10,000 cases, and each case contains 10 items. Assume that, on average, there are 5000 RFID readings per object every day. Further, assume that the temperature sensors report temperature readings every 5 minutes and that there are 100 temperature sensors in each warehouse. Let each RFID reading tuple be 20 bytes, and temperature reading tuple be 9 bytes. A simple calculation shows that, in this scenario, approximately 11 TB of data will be sent to the central location every day. Even if a compression scheme may offer a factor of 10 reduction in data volume, this will still yield over 1 TB of data each day. (For Details of the above analysis and other analyses in the rest of the section, please refer to appendix section 6.)

Distributed processing with state migration. An alternative to the centralized approach is to employ distributed stream processing. In this approach, each warehouse employs local stream processing—continuous queries on objects that reside in the warehouse are processed locally. As objects move through the supply chain from one warehouse to another, queries on those objects also “move” from one warehouse site to another in order to ensure local processing. The advantage of such an approach is that communication costs are significantly lowered since local processing implies that RFID streams are processed on-site.

However, the approach is not without drawbacks. First, the approach requires state migration to transfer the inference state and query state associated with an object whenever it moves from one warehouse to another. Such state overhead is high whenever historical information is involved (for either inference or query processing). Second, inference techniques become more complex. The inference state must be distributed across multiple sites while presenting the same logically unified view of the global state of the system as the centralized approach. The design of such distributed inference techniques is still an open research question and a focus of our work. Finally, since the distributed approach has no global view of the supply chain, queries that involve objects residing in different warehouse locations are more complex. For instance, queries might be tracking different parts required to assemble a product to ensure that they arrive at a factory within a short time of one another; since each part can take a different path through the supply chain, local query processing is either infeasible or may require substantial state exchange between different warehouse sites and the central location.

Analysis: Assuming the same scenario as before. Based on the time-varying graph model in Section 3, if 10 cases are read by the exit reader of the warehouse each time, the number of possible containers for each item is approximated by 10 in this analysis (it varies in practice with the colocation history and actual read rates of readers). Then the inference state of each object can be (roughly) estimated to be 184 bytes. Hence, the size of the inference state migrated every day across all warehouses is round 18.4 GB. As for query processing, object properties such as type of food are required for processing and need to be fetched from the manufacturer’s database to each warehouse. Assuming 100 bytes for each object, the total cost of fetching object properties is 11 GB every day. We estimate the size of query state using techniques in [1], e.g., 17 and 37 bytes (including a 12 byte tag id) per object for Query 1 and Query 2, respectively. If there are 10 queries of type Query 1 and 10 queries of type Query 2, the query state migrated every day is 54GB. Then the total communication cost is 83.4 GB every day, which is only 1/132 of the centralized approach.

Distributed processing with local tag storage. Our final architecture is an optimization of the distributed architecture where writeable RFID tags with onboard memory are exploited for further reducing the communication overheads. In this case, the state of all queries associated with an object as well as inference state is written onto the tag memory prior to departure from a warehouse. Upon arrival at a new warehouse, the tag memory is read and is used to “seed” the queries and the inference graph at the new location. It is important to note that the tag memory is used as a cache and that query/inference state continues to be stored at the prior location as before — in the event the tag memory can not be read for any reason, the approach reverts to state migration, where this state is fetched from the previous warehouse as before. Thus, writeable tags can be exploited to optimize overheads and the system correctness is not impacted even when tag memory is not available (or becomes corrupt).

Analysis: Since each tag already carries its id, the 12-byte tag id can be saved from any of the state regarding the object when stored in the tag’s memory. For each object, the size of object properties then becomes 88 bytes, the inference state becomes 172 bytes, and with the same number of queries, the size of query state becomes 300 bytes. All the object properties and inference/query state that need to
be stored on each tag total 560 bytes.

The above analysis shows that query state dominates the storage cost. Larger numbers of queries may challenge the scalability of this approach. Advanced techniques that share query state to save storage are outlined in the next section.

5 Status and Ongoing Work

As of June 2009, we have implemented the inference module and the query processor in a centralized fashion as well as a simulator for enterprise supply chains.

Our inference module includes a time-varying graph model that captures recent observations of objects and possible object containment relationships, and an online probabilistic algorithm that estimates the most-likely container and location of each object as new data arrives. Using RFID streams emulating a large warehouse, our results as shown in Figure 3 indicate that location inference can achieve 90% accuracy for the range of low read rates such as 50%. Our containment inference can achieve 90% accuracy when the read rates reach 85%. It has reduced accuracy for lower read rates due to both the loss of containment confirmation from special readers like belt readers and lack of consistent observations in the recent history. We are currently investigating advanced machine learning techniques to improve inference accuracy for the range of low read rates. Our initial results also show the inference techniques to be efficient: they can scale to 100,000 objects while running at stream speed.

Our ongoing work further explores a host of research issues for distributed inference and event query processing.

Splitting inference state. When a set of objects leave a warehouse, we need to split the graph constructed for inference, and transfer a subgraph relevant to these objects to the next warehouse (state migration) or store a subgraph relevant to each object in its tag memory (using local storage). These objects may be connected to other objects in the graph due to ambiguous containment, and only some of them may be sensed by the exit reader. Hence, it is a nontrivial issue to identify a subgraph that is large enough for accurate inference later but does not include unnecessary data. To further save bandwidth or storage, we also consider methods to truncate the historical information included in the graph without affecting accuracy.

Sharing query and inference state. To support multiple queries over millions of objects, we need to maintain a copy of query state per query per object. Consider the approach that stores query state in local tag memory. The tag memory then needs to hold the state of all the queries relevant to this object. This requires intelligent packing schemes given limited tag memory. One of our schemes exploits multi-query optimization. For all queries relevant to an object, we construct a shared query plan (e.g., a combined automaton) and further merge their query states once they become equivalent. As such, we can store merged query state using less tag memory. Another scheme exploits stable containment: all the objects that have been contained in the same container are likely to have the same state for each active query as well as the same inference state. Therefore, all the objects can share a single copy of the state in their aggregate memory, hence reducing the amortized memory usage in each tag. These schemes can also be used to reduce communication cost during state migration.

References

Appendix

Consider a supply chain with 1000 warehouses, where each warehouse stores 10,000 cases, and each case contains 10 items. There are 1000*(10,000+10,000*10) = 110,000,000,000 objects in this supply chain. Here, the object includes cases and items.

RFID readers in the shelves scan objects for every 10 seconds. An object is scanned 24*60*60/10=8640 times every day if it is always on shelves. The readers in other locations would have even higher read frequency, such as reader in receiving belt would read each second. However, object would not always in the scan range of the reader, for example, objects are moving from one warehouse to another by a truck. Assume that, on average, there are 5000 RFID readings per object every day.

The schema of raw RFID reading is as follows:
- Reader id, which is 4 bytes.
- Object id, which is 12 bytes.
- Time, which is 4 bytes.

The size of each raw RFID reading is 20 bytes.

Further, assume that the temperature sensors report temperature readings every 5 minutes and that there are 100 temperature sensors in each warehouse. The schema of temperature reading is as follows:
- Sensor id, which is 4 bytes.
- Temperature, which is 1 byte.
- Time, which is 4 bytes.

The size of each temperature reading is 9 bytes.

Centralized warehouse

In this approach, the communication cost consists of transferring raw RFID readings and temperature readings to the central sever. The number of raw RFID readings sends to the central sever is 110,000,000,000 * 5000=550G every day. The size of RFID tuples generated totals 20*550G=11 TB every day.

The number of temperature readings generated each sensor is 24*60/5=288 every day. The size of temperature readings generated every day is 1000*100*288*9=259,200,000 bytes.

In total, the communication cost is 11,000,259,200,000 bytes, which is approximately 11 TB. Comparing to the size of raw RFID readings, the communication cost of temperature readings can be ignored. Even if a compression scheme may offer a factor of 10 reduction in data volume, this will still yield over 1 TB of data each day.

Distributed processing with state migration

In this approach, the communication cost consists of the following three components:
- Fetching object properties from the central server.
- Query state migration.
- Inference state migration.

Fetching object properties Assume that each object moves to another warehouse every day. So, we should fetch the object properties from the central server to a new local server every day. Assuming 100 bytes of object properties for each object (may be compacted), the size of object properties fetched is 110,000,000,000*100=11G bytes every day.

Query state The query state of Q1 is as follows:
- Item id, which is 12 bytes.
- A[1].time, which is 4 bytes.
- Current automata state, which is 1 byte.

The size of query state for each Q1 is 17 bytes.

The query state for Q2 is as follows:
- Item id, which is 12 bytes.
- A[1].time, which is 4 bytes.
- Current automata state, which is 1 byte.
- Temperature history. We assume the average length of the history is 20, so temperature history uses 20 bytes.

In total, the size of query state for each Q2 use 37 bytes.

If there are 10 queries of type Q1 and 10 queries of type Q2. The size of query state for each item is 10*17+10*37=540 bytes. The query state migrated every day is the number of items multiplies the size of query state for each item, which is 100,000,000,000*540=54G bytes every day.

Inference state The inference state for each items includes the following information:
- Item id, which is 12 bytes.
- Last confirmed container id, which is 12 bytes.
- Recent co-location vectors with its possible containers:
  - Possible container id, which is 12 bytes.
  - Co-location vector, which is 4 bytes.

If 10 cases are read by exit reader of the warehouse each time, the number of possible container for each item is approximated by 10 in this analysis (it varies in practice with the co-location history and actual read rates of readers). The size of recent co-location vectors is estimated to be 160 bytes.

Then the inference state of each object can be (roughly) estimated to be 184 bytes. The size of inference state migrated every day is the number of items multiplies the size of inference state for each item, which is 100,000,000,000*184=18.4G bytes.

Then the total communication cost is 83.4 GB every day, which is only 1/132 of the centralized approach. From this analysis, the query state dominates the communication cost.

Distributed processing with local tag storage

Since each tag already carries its id, the 12-byte tag id can be saved from any of the state regarding the object when stored in the tag’s memory. For each object, the size of object properties then becomes 88 bytes, the inference state
becomes 172 bytes, and with the same number of queries, the size of query state becomes 300 bytes. All the object properties and inference/query state that need to be stored on each tag total 560 bytes.