Distributed Eventing Architecture:
RFID and Sensors in a Supply Chain

Kevin Emery

Abstract. We propose data structures to describe and query streaming RFID and sensor data. Furthermore, we propose an architecture built atop these data structures to build arbitrary real-time applications. To understand the nature of these applications, we decompose systems into four layers: *Physical*, *Data*, *Filtering*, *Application*. We describe each layer in terms of our presented data structures, and we discuss archtecture optimizations in terms of *Bandwidth*, *Computational Capacity*, and *Subsystem Transparency*. We provide an implementation of Track and Trace and Cold-Chain model applications to demonstrate our architecture.

Introduction

RFID and sensor data have the potential to add tremendous value to supply chain logistics. Logs of RFID data can provide an account of where particular objects have been and when they have been there. Sensors measure environmental attributes such as temperature, light, humidity, pressure, vibration, and sound.

Potential applications range from tracking a set of objects for expedited recall or delivery, to ensuring the quality of environmental-sensitive goods such food or medicine, to detecting breaches in shipping containers. Two particular applications we focus on are *Track and Trace* and *Cold-Chain*. In Track and Trace, one wishes to track the current location of a particular object, or trace its traversal in the supply chain. In Cold-Chain, one wishes to ensure the quality of temperature-sensitive goods as they traverse the supply chain.

In order to build such applications atop RFID and telemetry sensor data, we need a standard way to describe these data. We need efficient ways to sample the data and add semantic meaning. Furthermore, we need a sensible way to build applications on distributed data and machines. These applications must not only be distributed, but must also allow for dynamic reconfiguration so that new data models and applications may be seamlessly deployed.

We decompose such applications into various hierarchical layers. We begin by a study of the *physical layer*, which comprises the physical hardware of sensors with specific focus on temperature sensors [1], and we discuss some effects that propagate into higher layers. We

next discuss the *data layer*, which defines the data structures for describing RFID and sensor data, in addition to ancillary relational data, and also specifies a language for querying these data. Some related work in the field of streaming data comes from the Stanford Stream Project [3] and the MIT Aurora Project [5].

We next introduce the *filtering layer*, in which raw data is processed in the query language described in the data layer to be presented to higher layers in a compact and meaningful form. Some related work in this field particularly pertinent to temperature measurement comes from the Shelf-Life Model [9].

Lastly, we explore the *application layer*, which describes a distributed architecture built atop the lower layers. Not only do we describe a methodology for building distributed systems, but we also explore high-level optimizations and dynamic reconfigurations to allow for seemless deployment of new data models and applications. We finally present our implementation of this architecture in Track and Trace and Cold-Chain model applications.

Chapter 1

Physical Layer

In this chapter we describe properties of sensor hardware, which digitizes various environmental attributes. Once digitized, a computer can perform higher-level analysis on these data and other relational information to provide pertinent information for various applications. However, the way in which we capture environmental data digitally entails practical issues related conversion, resolution, error, delay, and power consumption. We study these physical properties because their effects propagate to higher layers.

Attributes one may wish to capture in applications include temperature, light, humidity, pressure, sound, vibration, location, etc. We begin with an in-depth study of temperature measurement, and we then show that the issues that arise are general to the measurement of many environmental attributes.

1.1 Temperature Sensor

Temperature is typically measured in Fahrenheit, Celsius, or Kelvin. The temperature in some volume is directly proportional to the internal energy in that volume, which corresponds to the magnitudes of the motions of the atoms and molecules. The most common way to obtain a digital measurement of temperature is with a thermistor [1]. A thermistor is a circuit element whose resistance varies predictably with ambient temperature.

A specific example of a thermistor used in a temperature sensor circuit is shown in Figure 1.1 [1]. In this diagram, the thermistor divides the voltage from source and provides input
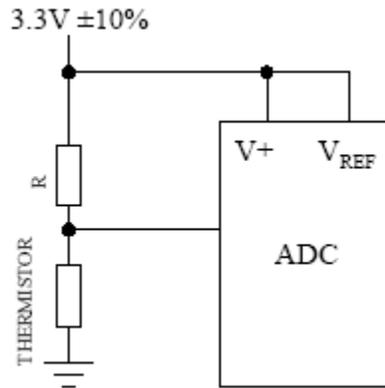

Figure 1.1 Circuit implementing a temperature sensor. The thermistor resistance varies with temperature and provides a divided voltage to the ADC representing a unit-less measure of temperature.

to an Analog-to-Digital Converter (ADC), which converts the analog voltage divided by the thermistor into a digital value that can then be stored in memory or sent off to be processed.

1.1.1 Conversion

The digital value output by the ADC does not yet correspond to the actual temperature in proximity to the thermistor. Specifically, this value encodes:

$$V_{ref}*(R_{pullup}/(R_{pullup}+R_{thermistor})),$$

where V_{ref} is the source voltage provided to the circuit, R_{pullup} is the fixed resistance labeled R in Figure 1.1, and $R_{thermistor}$ is the effective thermistor resistance. To actually determine temperature from this circuit, we use the characteristic output voltage for this thermistor for particular values of V_{ref} and R_{pullup}, shown in Figure 1.2 [1]. Notice that the curve is roughly

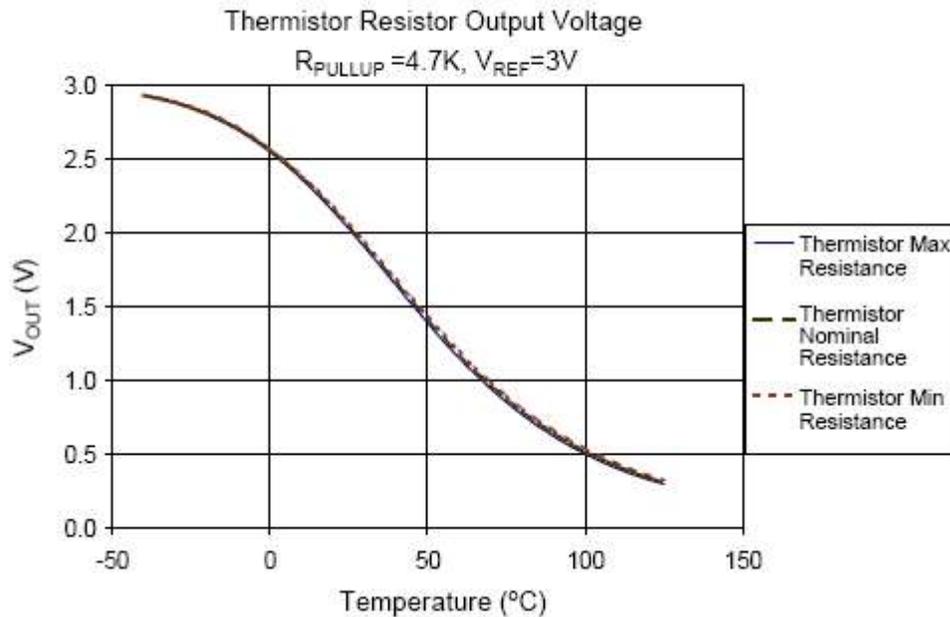linear for temperatures in the range 0 to 100 degrees Celsius.

Figure 1.2 The characteristic conversion curve for the thermistor in Figure 1.1. The operating region is between 0 and 100 °C, where the curve is approximately linear.

As such, if we intend the temperature sensor to operate in this region, we can estimate the output temperature as simply a linear function of the voltage measure between the two resistors shown in Figure 1.1 (V_{out}).

$$\text{Temperature (°C)} = a * V_{out} + b,$$

for appropriate coefficients a and b. This linear function on the output bits of the ADC can be computed by a digital processor. In general, if the transfer curve does not approximate an explicit function (linear, quadratic, exponential, etc.), or if the explicit function is difficult to compute, then a mapping can be implemented as a lookup table that simply maps the digital voltage value to a digital attribute value of particular units.

1.1.2 Resolution

To obtain a desired resolution on the digital temperature value, we must choose an appropriate number of ADC output bits. For example, in the range 0 to 100 degrees Celsius, the output voltage ranges over 2 volts as shown in Figure 1.2. For example, if we use 8 ADC output bits, then we have a resolution of roughly 10^{-2} on the voltage, since voltage ranges over 2 volts ($(2 \text{ Volts})/(2^8-1) \sim 10^{-2}$ Volts). This corresponds to a resolution on the order of half a degree Celsius for the temperature, as 2 volts expands to 100 degrees Celsius as shown in Figure 1.2 ($10^{-2} * 50 = .5$). An appropriate resolution to be used depends on the requirements of the application and the associated cost of the higher-resolution ADC in terms of processing, communicating, and storing additional temperature bits.

1.1.3 Error

Another issue that arises in this temperature sensor example is measurement error. Figure 1.3 shows the thermistor error as a function of temperature [1]. In sensitive applications,
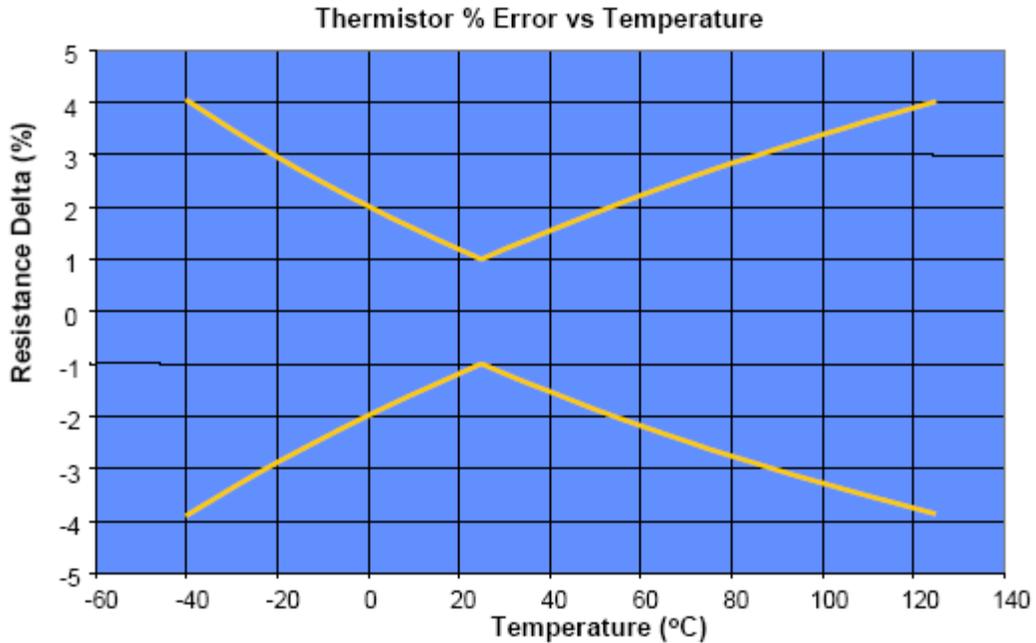
Figure 1.3 Thermistor error versus temperature. Note it is not constant. Error increases near the boundaries, and the optimal operating region is 25 °C.

such information might be useful, if not necessary. For example, if it is absolutely

necessary to be notified when the temperature in a particular region exceeds a threshold

even with low probability, then one might actually set an upper bound threshold at the

desired level minus the error of the thermistor at that temperature. More generally,

probabilistic models can be applied to the temperature readings and thermistor error

function by higher-level applications to estimate distributions over possible temperatures.

This type of data falls under the category of ancillary relational data consumed by higher-

level applications.

1.1.4 Delay

Another issue related to temperature measurement is delay. Suppose that the temperature in

proximity to a thermistor jumps immediately from 50 to 60 degrees Celsius. The time lag

before the thermistor resistance reflects the new temperature is related to the time it takes

for the heat to diffuse from the ambient atmosphere fully into the thermistor device itself.

That is, a diffusion process must occur between the atmosphere and the thermistor, before

thermodynamic equilibrium is achieved. Although this diffusion time is extremely small,

again, this delay may matter for highly sensitive higher-level applications. Furthermore,

there is delay for the ADC output to reflect the voltage change. However, the majority of

the delay occurs once the temperature reading has entered the digital world, for this binary

value must be sent somewhere to be stored and processed, whether this is on a CPU

embedded on the sensor or on a remote server.

1.1.5 Power

Yet another issue related to temperature sensing is power consumption. In the example

shown in Figure 1.1, power is consumed continuously, and the current drawn is clearly a

function of the temperature (alters thermistor resistance). This may be necessary to

arbitrarily sample temperature values. However, imagine we are interested *only* in sampling

temperature in a particular range. One could alter the circuit in Figure 1.1 by adding

multiple thermistors and transistors to cut current flow for invalid temperature ranges

(producing a 0 ADC output which must be properly interpreted).

In general, power is a scarce resource for sensors, and power optimizations are desired, but

again, there may be tradeoffs in the data available to higher-level applications and power

optimizations. Power is proportional to the number of samples taken multiplied by the cost

per sample. In building applications, one must consider appropriate sampling frequencies, or any non-constant sample rate optimizations, such as the one described above, or controller chips that switch the power supply on and off.

1.1.5 Drift

Another significant issue that arises with sensors is drift. That is, over time the measured value may drift from the actual due to some physical change to the sensor circuitry. The severity and nature of drift varies from sensor to sensor, and in general the only way to counteract drift is to periodically check sensor readings with a true reading and recalibrate as necessary.

1.2 Light Sensor

Thus far, we have walked through an example of a temperature sensor and measurement issues that arise. In fact, this example highlights general characteristics of sensors. In the case of light, we are measuring the intensity of electromagnetic radiation in a particular spectrum. A measurement device used here, analogous to the thermistor, is a photo resistor. An example is the VACTEC™ chip shown in Figure 1.4 [2]. Here, the voltage delivered to the ADC (not shown) is first amplified by an operational amplifier.
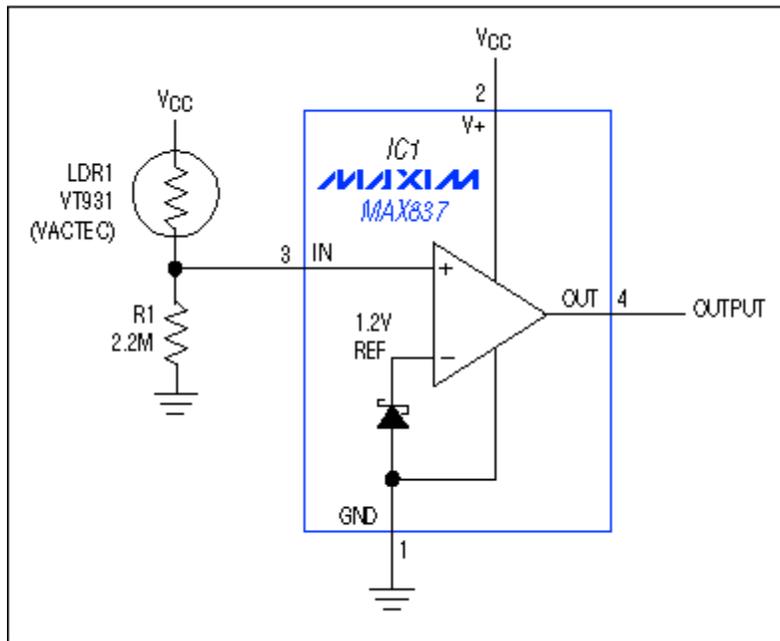
Figure 1.4 Circuit implementing a light sensor. The VT931 resistance varies with light intensity and provides a divided voltage to an OpAmp generating a unit-less measure of light intensity as output.

1.3 Generalizations

In addition to temperature and light, humidity, pressure, vibration, sound, and acceleration are all measured in a similar way. In fact, RFID readers are nothing other than sensors that measure coarse location of objects (tags). Readers emit electromagnetic waves that are reflected by passive tags. The reader senses which tags are present by the reflected waves and digitizes this output. In order to find an object's location, one may join a stream of RFID reads with a table mapping readers to locations. In general, at the lowest level an integrated circuit measures the attribute physically and alters a voltage input to an ADC. The ADC output represents a measurement of the attribute.

The issues of conversion, resolution, accuracy, delay, and power consumption are present

in each case. Data structures to represent sensor data must expose these practical issues as they effect higher-level applications. It is in the next chapter we construct a data structure to express RFID and telemetry sensor data.

Chapter 2

Data Layer

In this section, we describe a general framework for representing real-time streaming data. Many of the ideas here are borrowed from the Stanford Stream Project [3]. This framework is used to represent RFID and telemetry sensor data, in addition to associated relational data.

2.1 Tuples and Streams

We begin by defining a *tuple*. A tuple is a collection of typed elements. A *stream* S is a set of tuples (**s**, t), where **s** is added to S at time t. Element **s** represents a vector of attributes of primitive data types, such as booleans, integers, or doubles. If **s** has n attributes, then we represent tuples as $(s_1, \ldots, s_n, t)$, where s_i are in a set of primitive data types. Note that all tuples in a stream are of the same type.

To visualize a stream, imagine a semi-infinite conveyor belt onto which objects are placed. That is, the conveyor belt begins at one end and goes on forever in one direction. Objects are described by a set of attributes and placed at the edge of the conveyor belt, which moves away from the edge at some constant speed, representing the passing of time. At any point in time, the conveyor belt contains a set of objects placed at various points on the belt.

Consider the following example of a stream of temperature reads. Suppose we define a tuple of the form (temperature, temp_units, time, time_units) to represent the temperature

reading from some sensor at a particular time. Attribute "temperature" is a double precision floating-point data type, and "temp_units" is a character 'F' representing degrees Fahrenheit. Imagine that time is measured in seconds relative to some fixed point, and its units are represented by character 's' for seconds. Suppose that at time $t = 0$ the sensor reads 77.1 °F; at time $t = 1$ it reads 77.2 °F, and at time $t = 2$ it reads 77.2 °F. The stream representing this sequence of temperature reads is given by:

$$\{(77.1, \text{'F'}, 0, \text{'s'}), (77.2, \text{'F'}, 1, \text{'s'}), (77.2, \text{'F'}, 2, \text{'s'})\}.$$

2.2 Relations

A *relation* R is a set of tuples s. Analagous to a tuple of a stream, tuple $s = (s_1, \ldots, s_n)$, where attributes s_i are in a set of primitive data types. A relation can be thought of as a table in a standard relational database. An example of a relation is a table mapping EPC numbers to product descriptions:

EPC	Description
18.14829BD.174CC1.83B84FE89	Mach 3 Razor Head
52.018BC75.142D90.C97F2531B	Samsung Waterproof Headphones
11.A936B20.0D5695.16F8910AA	Dove Body Soap

ProductDescription Table

For our purposes, relations provide context and meaning to streams. For example, the ProductDescription table above provides additional information about a stream of EPC numbers generated by an RFID reader.

2.3 Operators

The next piece in our data framework is an *operator*. An operator O is an algorithm that operates on streams and relations. There are three types of operators we study:

1. O: R -> R
2. O: S -> S
3. O: RxS -> S

An operator of the first type takes a static relation as input and performs some algorithm on this relation to produce another relation as output. An operator of the second type takes a stream as input and produces a stream as output. The third type of operator takes both a stream and relation as input to produce a stream as output.

Operators form a closed algebra on the set of streams and relations, and we seek to describe a set of operators that is self-consistent and sufficiently expressive to construct various applications on the underlying data. In addition, we must allow users to add enw operators to an existing system, which we discuss in more detail in Chapter 4.

2.3.1 Relation-to-Relation Operators

In general, the standard query language (SQL) provides a set of operators of type 1 [4]. SQL is designed to manipulate and query relational databases. Elements may be added and deleted to a relation with INSERT and DELETE statements respectively. Elements may be modified with the UPDATE statement. Furthermore, a SELECT statement produces a relation as the result of a query on existing relations. As such, SQL provides a rich set of relation-to-relation operators and forms a closed algebra on relations.

Consider the ProductDescription table given above. A simple example of an operator that operates on this relation is the SQL statement "INSERT INTO ProductDescription VALUES ('EPC4', 'Good4')". This simply appends a new row to the ProductDescription table, and can thus be interpreted as $O(R) = R'$. We use a standard relational database such as Oracle, PostgreSQL, or MySQL to implement such operators. All of these support the necessary relation-to-relation operators.

2.3.2 Stream-to-Stream Operators

The Aurora Project describes a set of stream-to-stream operators proved sufficient to express a wide range of applications [5]. These operators include: *Filter, Map, Union,* and *Join.* The Filter operator produces a stream conditioned by predicates on tuples of the input stream. The Map operator applies a function tuple-by-tuple on a stream to produce a stream of projected tuples. The Union operator performs a set union on the streams and preserves total ordering. The Join operator produces a stream from stream(s) conditioned by predicates defined on the input stream(s). We seek to describe operators specific to RFID and sensor applications in the language of Aurora operators.

Two particular operators we study are *conversion* and *differentiation.* Consider a stream $S = \{(s, t)\}$. Conversion on stream S is $F(S) = F(\{(s, t)\}) = \{(f(s), t)\} = \{(s', t)\} = S'$. The function f transforms elements s to elements s'. Note that $|S| = |S'|$, where $|S|$ denotes the size of set S, but $s \mathrel{!=} s'$ in general. Conversion is nothing other than a map as defined in the Aurora paper.

An example of a conversion operator is transformation to degrees Fahrenheit on a unit-less temperature stream produced by a temperature sensor. Suppose that the manufacturer of a temperature sensor provides a function f that converts the unit-less readings x from the sensor to degrees Fahrenheit y ($f(x) = y$). Suppose a raw stream produced by the sensor is {(781, 0), (784, 1), (791, 2)}. Conversion on this stream might produce the stream {(77.1, 'F', 0), (77.3, 'F', 1), (78.0, 'F', 2)}, where tuples now contain the attribute 'F' corresponding to units of degrees Fahrenheit.

Differentiation is more subtle. It involves "differentiating" elements s on consecutive tuples (s, t) in a (time-ordered) stream. We define $D(S) = D(\{(s, t)\}) = D(\{\ldots (s1, t1), (s2, t2), \ldots\}) = \{\ldots d((s1, t1), (s2, t2)), \ldots\}$. The function d takes a "derivative" of two tuples. Clearly a derivative is ill-defined in general, as elements s take on arbitrary form, and multiple tuples may share the same time t. However, the idea is powerful in that it may condense streams while conveying the same amount of information. To define differentiation in terms of Aurora operators, take the union of the original stream and the stream shifted back by one time unit. Note we assume time-ordering of streams and regular time periodicity of tuples. The unioned stream is {... (s1, s2, t2), ...}. We now apply a map: $d(s1, s2, t) = (d(s1, s2), t)$. The result is a differentiated stream.

As an example, consider differentiation on RFID streams. Suppose that a reader produces tuples of the form ($tag_1, \ldots, tag_n, t$), where tag_i for $0 < i <= n$ are the tags in the reader's range at time t. We define the difference between two consecutive reads to involve two sets of tags: those present at time t2 but not at t1, and those not present at t2 but present at t1.

These refer to *entrance* and *exit* respectively. For example, suppose a reader generates the stream {(tag1,tag2, t), (tag1,tag2,tag3, t+1), (tag2,tag3, t+2), (null, t+3)}. The differentiation operator produces the stream {('enter', tag1, tag2, t), ('enter', tag3, t+1), ('exit', tag1, t+2), ('exit', tag2, tag3, t+3)}. The 'enter' and 'exit' fields indicate the type of event, and the following tags indicate which tags entered or exited. The differentiated stream clearly conveys the same amount of information more compactly than the raw stream, assuming tags stay within range of a reader for at least 3 polling periods and reads are not dropped at a high rate. Differentiation is explored in more detail in the following chapter.

2.3.3 Stream-and-Relation-to-Stream Operators

The most complex type of operator is one that uses relational information to condition a stream. A static relation containing information about elements of the stream is referenced for certain conditions. For a language to describe operators of this type, we again use a relational algebra, such as SQL, on the relation, and also on the stream up to the current time as if it were a relation. For each tuple that arrives, a SQL statement is run which has access to the relation and the stream. The query result is output each time the statement is run and thus forms a stream at the output.

Consider the following example. The union of a stream of temperature reads and a stream of RFID reads from a sensor and reader respectively at the same location and a static table of valid temperature ranges for objects is input to a *naïve cold-chain operator*. The naïve cold-chain operator checks that objects remain in their specified valid temperature ranges

by correlating the RFID and temperature streams from a particular loaction with the table of valid ranges for objects.

Suppose the RFID stream is given by:

{('enter', tag1, tag2, tag3, t), (null, t+1), ('exit', tag1, tag2, tag3, t+2)}.

The temperature stream is:

{(77, 'F', t), (76.8 'F', t+1), (76.9, 'F', t+2)}.

Note that in this example, the reader and temperature sensor are synchronized; that is, they both report tags and temperature respectively at regular synchronized time intervals. Lastly, suppose the table of valid temperatures is given by:

TagID	MinTemp (°F)	MaxTemp (°F)
tag1	32	90
tag2	32	75
tag3	80	100

ObjectTemperature Table

The naïve cold-chain operator performs the following SQL query each time a tuple from the stream is received:

1. SELECT 'violation', TagID FROM ObjectTemperature WHERE current_tag = TagID AND (current_temp > MaxTemp OR current_temp < MinTemp)

Note that the SQL query has access to the streams with the "current_temp" and "current_tag" variables. In general, the stream is represented as a relation with values up to the current time. Also, note that these queries are run for each tag present in an RFID tuple. The output of this operator in this example is:

{('violation', tag1, tag2, t), ('violation', tag3, t), ('violation', tag1, tag2, t+1), ('violation',

tag3, t+1)}.

Clearly this particular cold-chain implementation is a bit rudimentary to be applicable in a real supply chain; however, it demonstrates the use of streams, relations, and operators.

2.4 Queries

Lastly, we define a *query*. A query is a higher-level composition of streams, relations and operators. It is formally defined as a tree, where inner nodes (non-leaf nodes) are operators, leaf nodes are streams or relations, and edges transmit streams or relations. The output of the root node is the query result and may be either a stream or relation. At this level, a query tree specifies nothing about distribution on physical hardware. It simply describes an application built atop streams and relational data using a query tree as an abstraction. We discuss mapping queries onto physical hardware in Chapter 4.

2.4.1 Naïve Cold-Chain Query

Recall the so-called naïve cold-chain operator described in section 2.3.3. We now describe a *naïve cold-chain query*, which is constructed from the lowest-level streams. Imagine we have a reader that reports the set of tags in its range every 1 second, and a temperature sensor at the location of the reader that reports a unit-less temperature measurement (requires conversion to some known units) every second and is synchronized with the reader. The temperature sensor manufacturer gives us a function $f(x) = y$ that converts the unit-less temperature readings x into degrees Fahrenheit y. Suppose further that we have a

table mapping tag ID's to valid temperature ranges in degrees Fahrenheit.

We refer to the valid temperature table as T, the RFID stream as S_r, the temperature stream as S_T, the temperature conversion function as f, and lastly the cold-chain operator used above as C. We refer to F as the operator that applies f to the temperature readings from the stream S_T to produce a stream of Fahrenheit readings. Furthermore, we refer to D as the operator that applies differentiation to RFID stream S_r to produce entrance/exit events. We construct the following query tree shown in Figure 2.1 that represents the naïve cold-chain eventing application:
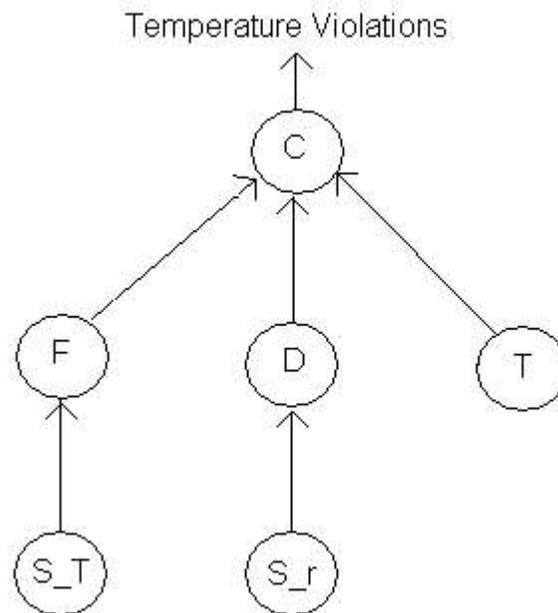

Figure 2.1 This query tree illustrates the naïve cold-chain query. Nodes S_T and S_r represent the temperature and RFID streams respectively. Node F performs conversion to degrees Fahrenheit on the unit-less temperature stream. Node D performs differentiation on the RFID stream. Node T is a relation mapping objects to valid temperature ranges. Lastly, node C is a cold-chain operator that correlates the RFID and sensor data with valid temperature ranges to generate temperature violation events.

This query tree illustrates the unit-less temperature stream S_T fed into the conversion

operator F. The output of this operator is a stream of temperature reads in units of degrees Fahrenheit. RFID stream S_r is fed into the differentiation operator D, whose output is a stream of RFID entrance/exit events. The two resultant streams from F and D, and the table T mapping objects to valid temperature ranges is used by cold-chain operator C, which computes violation events by checking that objects are not known to be exposed to unacceptable temperatures.

2.5 Data Model

We have presented a theoretical framework within which to describe and query streaming data. We move next to propose an explicit representation of RFID and sensor data in the framework presented.

2.5.1 Timestamp

We propose to use the ISO 8601 standard representation for the timestamp [6]. The ISO 8601 format describes a standard alphanumeric representation of date, time, and timezone. The format for a particular date is YYYY-MM-DD. For time of day, the format is hh:mm:ss. As for time zone, the above representation assumes some local time. To specify a time in Universal Time (UTC), which is equivalent to Greenwich Mean Time, append the character 'Z' to the end of the date time. To specify a local time zone relative to UTC, further append +hh:mm for time zones ahead of UTC, and for time zones west of the zero Meridian, append –hh:mm.

The particular time of 6:39 PM and 14 seconds on February 21, 2005, in Boston, MA, has

the following ISO 8601 representation:

2005-02-21 18:39:14 –05:00.

2.5.2 RFID Data

The Auto-ID Lab proposed a unique identifier for all products in the supply chain known as

the electronic product code (EPC). The EPC number is a 96-bit number that is unique for

all items. It has a specific format illustrated in Figure 2.2.

Version	Domain Manager	Object Class	Serial Number
8-bit	28-bit	24-bit	36-bit

Figure 2.2 EPC format has 8-bit Version slot for versioning, 28-bit Domain Manager for
specifying manufacturer, 24-bit Object Class for specifying product, and 36-bit Serial
Number for identifying product instance.

An example of an EPC number in hexadecimal is 21.203D2A9.16E8B8.719BAE03C. The

Version identifies the particular type of data structure behind the first 8-bits, and can be

used to introduce new structures into existing systems. The Domain Manager effectively

identifies the manufacturer. The Object Class is equivalent to a product number. The Serial

Number identifies the particular instance of the product [7]. We refer to the EPC number in

the format given in Figure 2.2 as a primitive data type.

Tags contain the EPC identifier for the item to which they are associated. Readers report

the set of tags (EPC numbers) in their range over time, though not necessarily at some

particular rate. A reader also has some unique identifier associated with it, which we call

the ReaderID and for our purposes is an integer data type. The ReaderID can be used to map a reader to particular information associated with the reader, such as its technical specifications, to which devices it is attached, where it is physically located, etc. This is nothing other than a relational join operator.

Formally we specify that the data stream readers produce consists of tuples of the form (EPC, ReaderID, timestamp). For simplicity, we propose that readers report only one tag per element in the stream. If multiple tags are present at the same instant, then multiple elements with identical timestamp are added to the stream. The details of how this stream is represented we leave to the section on implementation, but suffice it to say that it can be represented by XML or any other specific format, such as comma-separated text.

Consider the example where EPC number 21.203D2A9.16E8B8.719BAE03C is read by a reader with ReaderID 1005 at 6:39 PM and 14 seconds on February 21, 2005, in Boston, MA. The tuple corresponding to this read event is given by:

(21.203D2A9.16E8B8.719BAE03C, 1005, 2005-02-21 18:39:14 –05:00).

2.7 Telemetry Data

In this section, we begin by specifying the data types and units of attributes we wish to measure: temperature, light, sound, vibration, pressure, and humidity. We describe the data structure in the context of temperature. However, the structure we propose may be seamlessly extended to handle a broad class of attributes. The idea is that the attribute being measured is typically of the same data type, double-precision floating point, but the units of

measurement vary. For temperature, these units are Fahrenheit, Celsius, or Kelvin. Thus, we propose to construct a stream of tuples of the form (value, units, SensorID, timestamp), where value is a double-precision floating-point, and where units is a string representing the standard units of temperature measure: "F", "C", or "K".

Analogous to ReaderID, SensorID is a unique integer associated with the particular sensor that generated the reading. The timestamp is equivalent to the timestamp of the RFID stream. For each attribute there exists a table of known units. Figure 2.3 gives a unit table for temperature. Sensors typically produce some unit-less value which can be converted to one of the known units. The sensor manufacturer provides a conversion formula or table.

Units	Attribute
F	Temperature
C	Temperature
K	Temperature

Figure 2.3 Units table showing the standard units of a particular attribute, temperature. This table is used to interpret telemetry sensor streams.

A specific example of a temperature reading of 77.2 ℉ by a temperature sensor with SensorID 111 at time 6:39 PM and 14 seconds on February 21, 2005, in Boston, MA, is represented by the following tuple:

(77.2, 'F', 111, 2005-02-21 18:39:14 –05:00).

Note that conversion is handled outside of this data model. The idea is to decouple specifics of sensor hardware and meaningful data with associated universal units of measure. One may, however, represent unit-less measurements by a null value in the units field of the tuple, and then also supply a conversion funciton.

The tuple structure can be easily used to handle other attributes as well. Suppose we wish to construct a tuple for sound. If we measure sound in decibels, and add a row to the Units table in Figure 2.2 to include a unit type of "Db" for attribute "Sound", then the following tuple has explicit meaning:

(12.2, 'Db', 111, 2005-02-21 18:39:14 –05:00).

We have explicitly defined the structure of *ideal* tuples produced by RFID readers and telemetry sensors. These tuples may be extended to encapsulate some of the practical issues associated with sensors described in Chapter 1. For example, a delay or error attribute may be appended to the tuple. For example, if tuple $(s_1,\ldots, s_n, t)$ is added to a stream at time t, and we calculate a delay d from the time the tuple theoretically should have been added to the stream, then we may add tuple $(s_1,\ldots, s_n, d, t)$ instead.

We have described an architecture of tuples, streams, relations, operators, and queries that provide tools for building applications atop these data. We have also specified specific representations of RFID and sensor data in our data framework. In the next section, we describe how to construct these streams efficiently. That is, we discuss filtering techniques and sampling rates.

Chapter 3

Filtering Layer

We have proposed a way to describe and process data produced by RFID readers and sensors. The issue now arises of how to efficiently sample these data. That is, we are not passive listeners to RFID and telemetry sensor streams. We explore efficient ways to populate streams.

3.1 Inference and Differencing

Two general principles we wish to exploit are *inference* and *differencing*. The idea is to transmit only changes to existing state [8]. An example used is that of a car with certain velocity v. If one wishes to transmit the location of the car, one need only to report changes in its velocity, for it can be inferred by silence that the position is that of the last known position plus the integral of the last known velocity v. If the velocity is roughly constant, then we may substantially reduce the data transmitted while conveying the same amount of information.

However, silence, while it may be interpreted as the absence of change, may in fact be due to failure in communication. Perhaps the transmitter fails and reports no events, or perhaps the communication link breaks. In order to maintain symmetry of information, a *heartbeat* is disseminated. A heartbeat minimally conveys the continued functionality of a data source, and thus carries little overhead.

Note that the ideas of inference and differencing are independent of the data model. It is not necessary to describe them in terms of the data model. Rather, they are guiding principles in constructing data streams efficiently.

3.2 Filtering RFID Streams

We wish to apply the principles of inference and differencing to RFID streams in order to more efficiently communicate the data. Suppose a reader produces readings at a rate of 1/T. That is, every T time units, it reports the set of tags in its range or null if there are none. If tag x remains in a reader's range for much longer than T time units, then it becomes superfluous that the reader continue to report the presence of x. Rather, the reader might report its first detection of x (entrance), and then later report the absence of x (exit).

The entrance calculation is easy: maintain a set of tags currently in range; when x is in range but not in the set, add it to the set and report entrance. The exit calculation is analogous: on a reading, check the set of detected tags against the set of tags in range; the tags not detected but in the set of tags in range have in fact exited. In terms of operators, this is equivalent to differentiation on the RFID stream.

The problem with this simple approach to filtering RFID events is that readers tend to drop tags in their range. Suppose that for some properly functioning tag in a reader's range, the probability that the reader reports the tag on a read cycle is p. There is nothing to be done, given the accuracy of a reader, about delays in reporting entrances. If a tag is present but not detected by the reader, then there is unfortunately no way to otherwise discover the tag,

barring increasing the chances of a read by adding redundant readers, or improving the hardware itself.

Erroneous exit events, however, can be improved. Clearly a dropped read will result in a superfluous exit/entrance event pair. We wish to minimize these superfluous events by only reporting an exit when a tag is not reported for k consecutive read cycles. The value k is chosen to produce an acceptable probability of error, for the probability of not reporting a present tag for k consecutive read cycles is $(1-p)^k$ assuming the independence of consecutive read cycles. However, the obvious tradeoff introduced is accuracy versus the delay of k time units. In fact, the operation we are performing on the RFID stream is a specific instance of a low-pass filter. A general tradeoff of low-pass filters is smoothness versus lag.

We now proceed to describe this filter in terms of the operators presented in Chapter 2. We implement a low-pass filter in the following way: take the union of k+1 streams, where stream i is shifted back i time units. Apply a map to the k+1 streams which reports an exit if a tag is present at time t but not for any of t+i for $1 <= i <= k$. This produces a smoothed read stream. We now simply differentiate the smoothed read stream to produce a stream of entrance and exit events.

3.3 Temperature and Quality

A pertinent application in the context of temperature measurement is the cold-chain application. In a cold-chain, one wishes to ensure that temperature-sensitive goods such as

food and medicine are kept at appropriate temperatures as they traverse the supply chain. Supposing we can trace the space-time history of an object, and we also know the temperature in its vicinity at all times, how do we use this information to guarantee the quality of goods? What are the implications of our measurement of quality on the minimum amount of data necessary, and consequently how we sample temperature data?

A simple approach is the one used by the naïve cold-chain monitor. That is, some fixed thresholds are set for a particular object. Temperature is sampled at a constant rate, and whenever the temperature in the vicinity of an object violates these thresholds, an event is generated. The problem with this approach is that a short-lived spike or dip in temperature, perhaps caused by a transition, will trigger an unnecessary event. Moreover, if the temperature sits near but within a boundary, no event is ever fired. Food or medicine may be spoiling, albeit at a slower rate than were the temperature outside the bound. However, this fact is not reflected in the cold-chain events. The choking assumption is that there is no time-dependence on food quality. Goods within some hard temperature bound do not decrease in quality, yet any time the temperature exceeds the bounds, a good's quality is compromised and notification is necessary.

A more sophisticated model of food quality is the shelf-life model [9]. The shelf-life model estimates the quality of food as a function of temperature and time using Arrhenius kinetics. The result is a quality score $Q(t, T)$, a function of time t and constant temperature T. It turns out that for any constant temperature T, $Q(t)$ decreases as $1/t$ from 100 to 0. For larger T, the characteristic timescale of $Q(t)$ decreases; that is, it moves more quickly from 100 to 0. Explicitly, $Q(t, T)$ is given by:

$$eqx3\_2 := Q(t) = \frac{25}{432\ e^{\left(-\frac{15034}{5T}\right)} t + \frac{1}{4}}$$

Equation 3.1 Quality score as a function of time t and constant temperature T. Note the rational dependence on t ($1/t$) and exponential dependence on temperature ($e^{1/T}$).

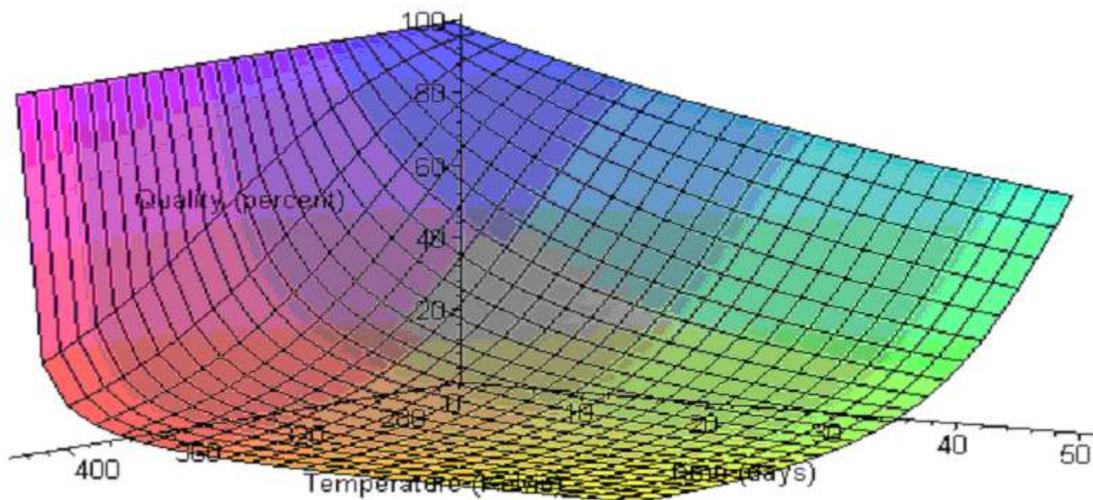Temperature T is measured in Kelvin and time t in days. Figure 3.1 shows a surface plot of Q(t, T).

Figure 3.1 Quality score as a function of time and temperature. Note the faster rate of decay as temperature increases.

The solution Q(t, T) as stated involves a constant temperature T. This has limited use, as temperature varies. We can construct Q(t, T(t)) with time-varying temperature by appending segments of Q(t, T) for different values of T. Specifically, if at time t, the current quality score is Q(t), and the temperature at time t + 1 is a constant T, then Q(t + 1) = Q(t) + Q(t + 1, T) – Q(t, T). The base case for constructing Q(t) is simply Q(0) = 100. The result is a function decreasing at $1/t$ with time-varying characteristic timescale.

For example, suppose a good has existed for 4 time units, and we have the 4 measured (constant) temperatures for those time units. The temperatures are given as:

Time	Temperature (Kelvin)
0	295.59
1	295.98
2	295.98
3	295.71

We wish to find the good's current quality score $Q(4)$. We assume $Q(0) = 100$ and proceed in the dynamic programming style described above:

$Q(0) = 100$.
$Q(1) = Q(0) + Q(1, 295.59) - Q(0, 295.59) = 93.81$.
$Q(2) = Q(1) + Q(2, 295.98) - Q(1, 295.98) = 88.27$.
$Q(3) = Q(2) + Q(3, 295.98) - Q(2, 295.98) = 83.36$.
$Q(4) = Q(3) + Q(4, 295.71) - Q(3, 295.71) = 78.99$.

It should be noted that temperature in fact depends on other factors as well, such as pressure. We are assuming a stationary physical state within which a continuous temperature measurement is valid. Furthermore, different goods have different physical properties such as freezing point, melting point, boiling point, etc. For example, the quality of oranges may behave according to the shelf-life model. However, once oranges freeze, thir quality effectively goes to zero. Such non-general behaviors must be accounted for as well.

Now that we have a measure of quality that varies with temperature and time, we discuss the implications on sampling temperature. Small-scale fluctuations in fractions of the timescale of Q (days in the above example) have little effect on the overall quality. An ephemeral temperature spike that quickly returns to its previous level does not significantly

decrease quality, as proposed by the shelf-life model. The naïve cold-chain algorithm, however, would trigger on such an event. The idea with the new model is to compute the quality score for goods as they traverse the supply chain and transmit data on the timescale of quality measure, not the timescale of low-level temperature fluctuations. This follows from the principles of inference and differencing, as it is the quality of food one cares about in a cold-chain application, not small-scale temperature fluctuations. Thus, data is transmitted at a rate minimally sufficient to reconstruct a quality score.

One approach to estimating the quality score of a good is to compute the average value of its temperature on the timescale of the quality function and then use this average to compute the quality drop from one time unit to the next. The question is how to accurately compute this average by minimally sampling temperature within a time unit. To do this, we use the Nyquist criterion, which states that in order to fully reconstruct a function f(t), the minimum sampling frequency is twice the highest frequency component of f. We estimate the highest frequency present in temperature fluctuations along a supply chain, sample at the appropriate frequency, and integrate over the time unit to compute the average. If we attempt to sample below the Nyquist frequency, we may miss high-frequency spikes in temperature that affect the average.

In terms of a query tree, the data source S_r produces a highly sampled temperature stream, which is input to an averaging operator A. Operator A outputs a single value per time unit of Q, and it may be implemented as a map over a window of tuples in S_r. Note that the output stream is simply a down-sampled temperature average. It has nothing yet to do with quality score. This average is then fed to a Quality operator Q. Operator Q performs a map

on the temperature over time implementing the dynamic generation of $Q(t)$. In order to do this, it must of course have the state of $Q(t-1)$. This is simply a map over a window of streams $Q(i)$ and $Q(i-1)$.

In this section we have proposed efficient ways to construct the low-level data streams to be used in supply chain eventing applications. We have discussed how to filter RFID and temperature streams to condense superfluous information. We move next to describe an architecture for these applications built atop low-level data streams, which maps the query trees described in Chapter 2 onto physical machines.

Chapter 4

Application Layer

We have defined the data structure of the low-level streams produced by RFID readers and telemetry sensors, which are the basis for higher-level applications. These mechanisms are described as query trees. We have also proposed how to efficiently construct low-level RFID and sensor streams. That is, how to sample and filter the data in terms of query trees. We move now to describe a system architecture that maps query trees onto physical hardware to implement real applications.

4.1 Query Trees and Machine Graphs

Recall the query tree described in chapter two. It is a hierarchical structure, where sub-streams in sub-queries produced by sub-trees are conjoined at parent nodes to produce query results.

We define a *machine node* as the processing medium on which operators perform algorithms on streams and relational data. We distinguish between *query nodes* and machine nodes in that query nodes exist in the context of a query and implement an operator in the query tree. Machine nodes, on the other hand, represent the physical hardware on which query nodes exist, such as servers, PCs, laptops, or sensor boards.

Machine nodes may contain multiple query nodes (i.e., one machine implements multiple operators). For simplicity, if a single query node is in fact spread across multiple physical

machines, we refer to the set of machines as a single machine node, and thus query nodes do not contain machine nodes.

We define a *machine edge* as the physical communication medium between two machine nodes, such as a TCP/IP socket or SOAP web service. Analogously, we refer to a *query edge* as the edge connecting two query nodes. As such, edges between query nodes contained by different machine nodes are in fact contained by machine edges that implement the data being transferred from query node to query node.

A *machine graph* is the graph of machine nodes and edges. Note we cannot necessarily specify a machine tree, for there are no restrictions on which query nodes exist on which machine nodes. As such, the machine graph may not be a "tree" in terms of data flow.

Consider the example of the naïve cold-chain query described in Chapter 2. Suppose a single laptop has attached to it a reader and a temperature sensor. The laptop collects RFID and temperature streams and performs differentiation on the RFID stream and conversion to degrees Fahrenheit on the temperature stream. Suppose further that a centralized server contains a database mapping objects to valid temperate ranges. The laptop forwards the differentiated RFID stream and converted temperature stream to the server over a TCP socket.

The server performs a cold-chain eventing algorithm and produces cold-chain violation events on screen. Note that the server behaves as a client in this context, for it consumes low-level data provided by the laptop to serve cold-chain events to the end user. Figure 4.1

shows the machine nodes "Laptop" and "Server" which contain query nodes of the cold-chain query shown in Figure 2.1. Keep in mind that this architecture is the base case for a centralized architecture in which multiple laptops or machines with attached readers and/or sensors forward data streams to a centralized server that contains a database of global information.
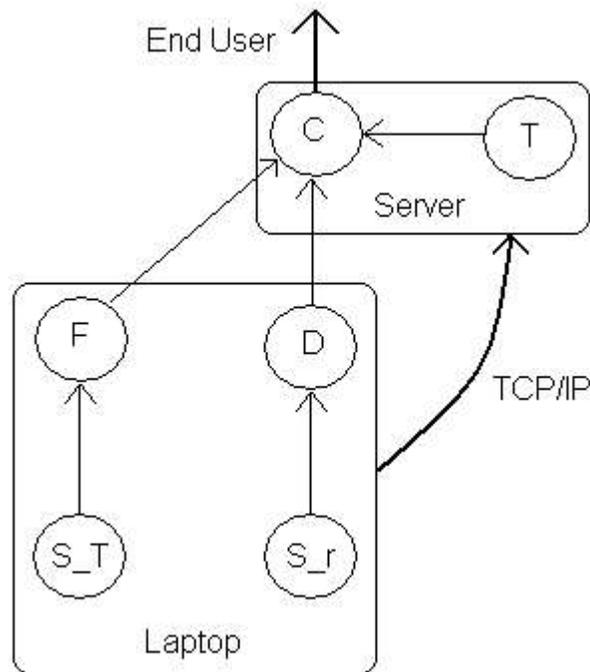

Figure 4.1 System architecture for naïve cold-chain application. Laptop machine node contains query nodes S_T, S_r, F, and D, which provide filtered RFID and sensor streams to the Server machine node. The Server contains global database T and cold-chain eventing application C.

Note that machine nodes are square and query nodes are round. Machine edges are bold. The Laptop machine node contains query nodes F, D, S_T, and S_r. Machine node Server contains query nodes C and T. The edges from F and D to C are implemented by the machine edge TCP/IP. The machine graph contains only the Server and Laptop nodes, connected by an edge.

4.2 Query Optimizations

By constructing a mapping from a query tree to a machine graph, we have specified the physical structure of a query spread across disparate data and machines. The issue arises of how to efficiently and appropriately distribute the computation.

The metrics we use for measuring architecture performance and feasibility are *bandwidth*, *computational capacity*, and *transparency*. In our framework of a machine graph, we define bandwidth to be the sum of data traffic along machine edges in a machine graph. Computational capacity is the capacity of machine nodes to perform computations (i.e., contain query nodes) and is related to the processing power, memory, and storage on a machine. We say that query nodes use capacity, and a machine node may contain as many query nodes whose sum of capacity usages does not exceed the capacity of the machine. Lastly, we define transparency to be the degree to which higher-level applications have access to any sub-query in the query tree. That is, consider the *level* of a node in a query tree, which is the length of the path from that node to root. Transparency is the ability of applications to have access to data at various levels in a query tree.

Before we explore optimizations of these metrics, we define some terminology. A *consumer* is a machine node that contains query nodes that are parents in the query tree of query nodes of another machine node. A *producer* is a machine node that contains query nodes that are children of query nodes of another machine node. A machine node may be both a producer and a consumer. In the machine graph shown in Figure 4.1, the Laptop is a

producer, for it provides RFID and sensor data to the Server. The Server is both a consumer and a producer, for it consumes RFID and sensor data provided by the Laptop and produces cold-chain events for the end user (node not shown).

Suppose for now that we have infinite machine capacity, and we wish to minimize bandwidth. Note that we could analagously optimize load balancing subject to bandwidth and capacity. The solution techniques are similar, and here we focus on the former. For a given query tree and set of machine nodes, it is clear that in terms of minimizing machine edge traffic, we wish to push the query tree down into producer nodes where possible. This idea is closely related to pushing logic to the edge of an overall system. The assumption is that query nodes tend to reduce data traffic from input to output. Typically, query nodes are presented with data covering a realm of possibilities, and they perform filtering and/or correlation to condition the data to a more meaningful and compact form. Thus, by pushing query nodes down into the machine graph, we reduce traffic along machine edges. Note that we may also reconfigure the query tree to reduce bandwidth along query edges, but we assume an optimal query tree to be mapped onto machine nodes.

Consider Figure 4.1. Imagine that query node D is contained by the server node. Since the output stream of D is more compact than its input stream, there will now be more traffic along the machine edge. By keeping D in the laptop, we have more efficiently implemented the same query.

However, certain query nodes are fixed to machine nodes by construction. In Figure 4.1, query nodes S_T and S_r are fixed to the laptop, for the reader and sensor are physically

connected to the laptop. In a centralized architecture where T is a global database and eventing application C feeds events to listening clients, query nodes C and T are fixed to the server node. Thus, our only freedom is with nodes D and F. By pushing these nodes down to the laptop, we have optimized this machine graph in terms of bandwidth.

There are, however, restrictions we must obey. Machines have finite and varying capacity. In fact, machines toward the edge of a system (i.e., motes, readers, and laptops) are likely to have less capacity than machines on the backbone (i.e., servers). The distribution of machine capacity opposes our goal of pushing logic to the edge of a system.

Consider that the laptop in Figure 4.1 has some limited capacity. If the laptop has enough capacity to run one of D and F, but not both, which do we choose? Clearly we choose the one whose reduction in data rate from input query edge to output query edge is greatest. Since F merely converts the unit-less temperature to units of degrees Fahrenheit, whereas D compresses the raw RFID streams, we choose to place D on the laptop. This minimizes machine edge traffic.

We formally consider the bandwidth optimization problem with finite machine capacity and fixed query nodes, setting aside transparency considerations for now. Consider query tree T of query nodes n_i with associated capacity usages u_i and data rate reductions d_i, for $0 < i \leq |T|$. Note that $u_i, d_i \geq 0$. The data rate output by a query node is simply d_i multiplied by its input data rate. (Note that if $d_i > 1$, the node *increases* data rate from input to output).

The set of machine nodes onto which T is mapped is given by $M = \{m_j\}$, where each

machine node m_j has associated capacity c_j for $0 < j \leq |M|$. Note that a machine node m with capacity c may contain as many query nodes with usages whose sum is less than or equal to c. There is also a mapping of query nodes fixed to particular machine nodes. Note also that the mapping of fixed query nodes to machine nodes is onto. That is, every machine node m in the set M has some fixed query node. We describe a greedy algorithm whose goal is to minimize bandwidth given capacity constraints and fixed query nodes:

1. Assign all fixed query nodes to their corresponding machine nodes.
2. Order the machine nodes by depth of deepest query node assigned to the machine node.
3. In reverse order of machine nodes:
    4. Find the set of query nodes that are parents of query nodes in the machine node and not yet assigned to another machine node.
    5. Order this set of query nodes by d_i in ascending order.
    6. For each query node in order:
        7. If the query node's usage does not exceed the remaining capacity of the machine, then add the query node to the machine and subtract from the machine's available capacity.
8. If the algorithm fails to assign all query nodes to machine nodes due to capacity constraints, then use additional machine nodes to contain unassigned query nodes.

The algorithm basically assigns query nodes to machine nodes far down the query tree while attempting to minimize machine edge traffic and obeying the constraints of fixed query nodes and machine capacity. The algorithm is not necessarily optimal, but the problem as stated is difficult to solve. Figure 4.2 illustrates the algorithm on a simplified problem in which we assume no effects due to machine capacity.
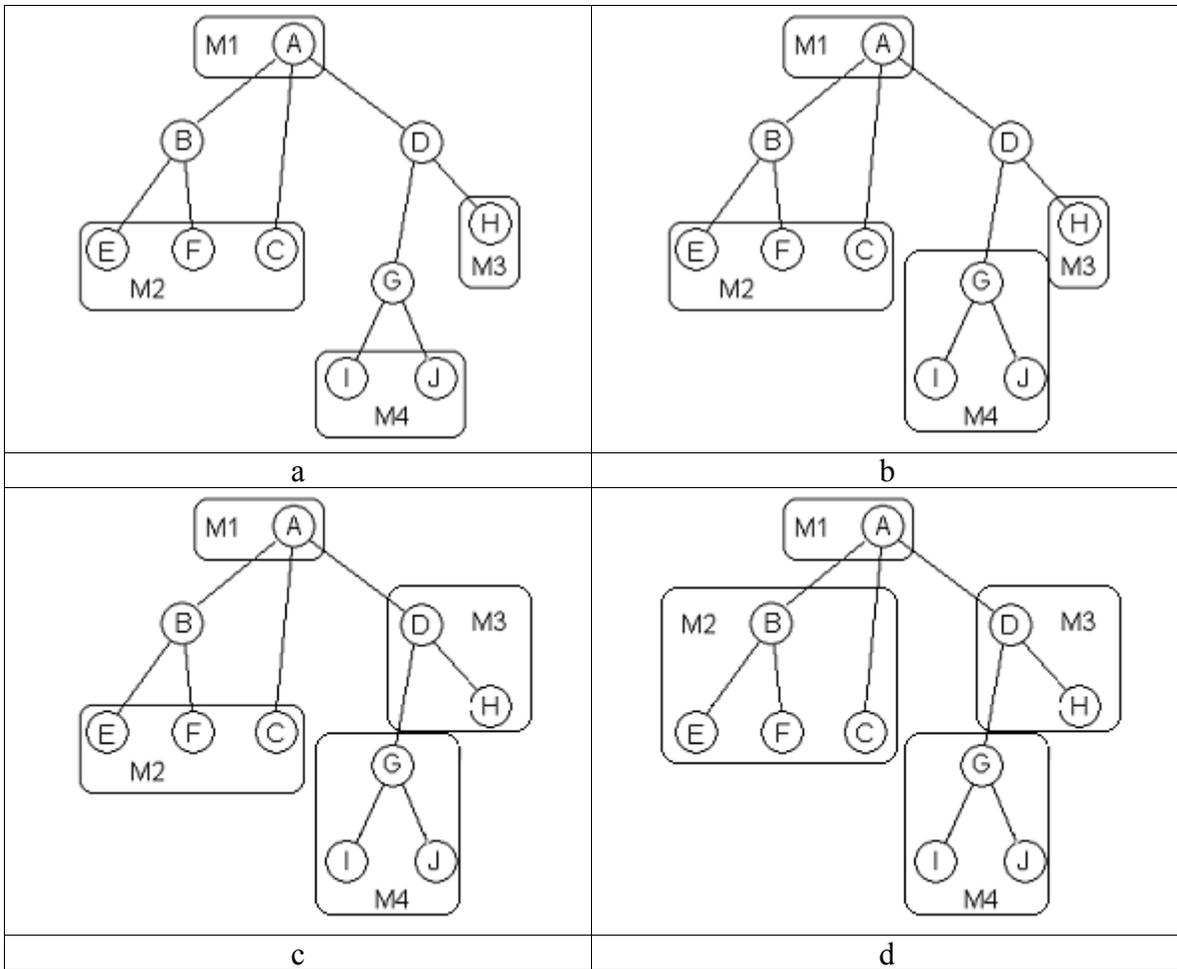
Figure 4.2 Structural improvement algorithm operating on a query tree. Machines furthest down the query tree consume unassigned query nodes in an attempt to minimize machine edge traffic while obeying query node and machine capacity constraints.

Figure 4.2a shows a query tree with nodes A,B,… J, four machines M1,M2,M3, and M4, and node A fixed to M1, nodes E, F, and C fixed to M2, node H fixed to M3, and nodes I and J fixed to M4. Step 1 of the algorithm is complete. Now step 2, looping through machines furthest down the query tree first, 4.2b shows M4 consuming G, 4.2c shows M3 consuming D, and lastly 4.2d shows M2 consuming B. The machine graph in Figure 4.2d is in fact optimal in terms of bandwidth given the constraints.

4.2.1 Transparency and Reconfiguration

The bandwidth optimization problem was considered in the context of a particular application running on a particular set of distributed machines and data. Transparency and reconfiguration come into play when new models (i.e., new operators or query sub-trees) are developed, and one wishes to deploy these new models in existing applications or super-impose new applications onto existing machine infrastructure.

The bandwidth optimization technique of pushing query nodes (operators) down the machine graph by construction results in pushing logic toward the edge of the system. However, this may become a problem if one wishes to have access to low-level data layers, or if one wishes to change the logic. In a sense, to optimize transparency, one might consider doing the opposite of the bandwidth optimization algorithm. Consider the following architecture shown in Figure 4.3 for the naïve cold-chain query described in Chapter 2.
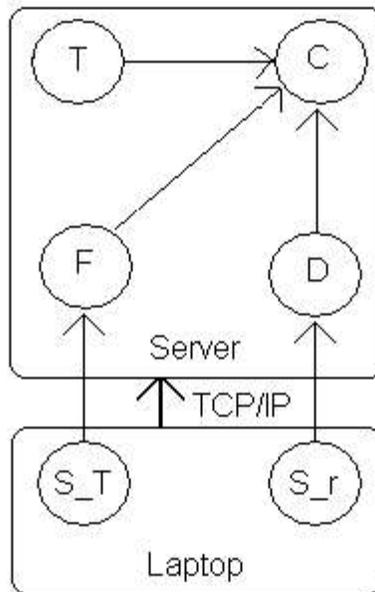
Figure 4.3 Naïve cold-chain machine graph with all of the logic at the Server. Laptop simply forwards lowest-level streams S_T and S_r to Server.

This implementation uses the laptop as a dummy conduit to the server for the low-level data streams. If one now wishes to have access to low-level stream S_r to use a better differentiation algorithm D' on RFID streams, for example, then one only has to change the code on the server. Presumably the laptop is out in the field gathering appropriate RFID and temperature data and updating server code is easier. However, the query tree is not optimally partitioned in terms of bandwidth. In general, bandwidth-optimal architectures have much of their logic toward the edge of a system, which opposes transparency and reconfiguration.

Our proposed solution to this problem involves using a bandwidth-optimized architecture. However, we propose to use a set of services running on machine nodes that allow users to gain access to hidden query layers and also to dynamically reconfigure the query tree structure. Given that these applications are widely distributed, we must enable *remote* layer access and reconfiguration. The machine services may be implemented as SOAP web services [10] or any other such protocol, but for our purposes these implementation details are not pertinent. It must be mentioned that there are dimensions of ownership, permission, and cost associated with these services, but these issues are beyond the focus of this thesis.

We begin by defining the services that implement machine edges and comprise the query. For each machine edge, the tail of the edge is associated with a *publish service,* and the head is associated with a *subscribe service.* A publish service registers that a producer machine node has data available, and the subscribe service registers that a consumer

machine node consumes data along a machine edge. In this way, the queries are constructed as a set of publish/subscribe communication mechanisms. Note that there may be multiple input/output specifications for a single machine node. The same query node may be used in multiple query trees for multiple applications, and thus one set of machine input/output specifications is used for one application, while another set of specifications is used for another.

4.2.1.a Discovery Service

In addition to the publish/subscribe services which implement a particular query, machines support general services necessary for reconfiguration. The first such service that machine nodes must support is a *discovery service*. The discovery service simply reports the machine node input/output specifications and also the input/output specifications of query nodes contained by the machine, in addition to the query sub-tree structure(s) on the machine. Note that machines may contain disjoint query sub-trees within the machine but which are connected in the entire query tree; however, this is inconsequential. This service is passive; machine nodes simply report their own structure. Again, as for specific formats, XML or comma-separated text suffices, but these are implementation decisions beyond the scope of this thesis.

4.2.1.b Reconfiguration Service

The second type of service is a *reconfiguration service*. The reconfiguration service allows for two types of reconfigurations: query sub-tree reconfiguration, and machine node

input/output specification modification. The first type of reconfiguration involves specifying new query nodes and (possibly) a new query sub-tree structure to exist on the machine node. This reconfiguration must obey the machine input/output specifications. The idea is to upload code or binaries that implement query nodes and to specify the query sub-tree structure so that a machine may automatically reconfigure itself to implement the new sub-query.

For example, suppose we wish to use D' instead of D for differentiating RFID streams as shown in Figure 2.1. We use the discovery service to learn the input/output specifications of D on the laptop, and we use this information to implement D'. We then use the reconfiguration service to swap D' for D on the laptop machine, and thus we have seamlessly reconfigured the application.

The other type of reconfiguration involves changing machine specifications. Suppose we wish to replace operator F shown in Figure 2.1, which converts unit-less temperature streams into temperatures of degrees Fahrenheit, by operator Ce, which instead converts to degrees Celsius. Implicitly, the server is effected by this change, for cold-chain operator C expects degrees Fahrenheit, not degrees Celsius. Not only must we swap Ce for F, using the same method described above for swapping D' for D, but we must also modify the laptop and server machine specifications to output and input, respectively, temperature streams of degrees Celsius, rather than degrees Fahrenheit. To do this, we use the reconfiguration service to modify the output specifications of the laptop and the input specifications of the server.

We have proposed a general architecture for constructing arbitrary queries on disparate data streams and relations over multiple machines. In addition, we have formalized the notion of pushing logic to the edge of a system. Furthermore, we have described a way to reconfigure the distributed query so that new models may be seamlessly deployed across the distributed system, and multiple applications may be implemented over the same data and hardware infrastructure. We move next to demonstrate an actual implementation of the proposed architecture, specifically providing Track and Trace and Cold-Chain applications using TagSense RFID readers and Crossbow telemetry sensors.

Chapter 5

Implementation

We have thus far discussed much of the theory behind building applications on RFID and telemetry sensor data. We now demonstrate an actual implementation of the Track and Trace and Cold-Chain applications using readers, sensors, a laptop, and a server.

5.1 RFID Readers

The readers we use are 1356-MINI readers from TagSense [11]. These readers operate at 13.56 Mhz and read tags at a close range of roughly 1 cm. They support the EPC protocol and output data in RS-232 format through an adapter to a USB cable. The USB cable is connected to my Dell laptop running Windows XP.

The laptop runs a Windows application written in C# called ReaderInterface that interfaces with the reader by listening on the appropriate COM port. The reader forwards tag ID's as it detects them to this COM port. The ReaderInterface program constructs the tuple (EPC, ReaderID, timestamp). The tuple is specifically represented by the string "(EPC, ReaderID, timestamp)". The EPC number is in the format described in Chapter 2. The ReaderID is an integer, and the timestamp is the string representation of a C# System.DateTime object. These tuples are the elements of the RFID stream described in Chapter 2. It represents the node S_r of the query tree shown in Figure 2.1.

5.2 Telemetry Sensors

The telemetry sensors boards we use are MTS300CA from Crossbow [12]. These sensor boards have the following functionality: microphone, sounder, light sensor, temperature sensor, 2-axis magnetometer, and a 2-axis accelerometer. The sensor boards are powered by two AA batteries, and as a unit we refer to the sensor board and attached power supply and antenna as a *mote*. This mote communicates wirelessly using a pre-zigbee proprietary protocol. The motes run TinyOS locally, a primitive operating system designed to run on sensor boards with limited computational resources.

A *host sensor* connects to the serial port of my laptop. The host sensor is the gateway to the laptop for a set of wireless motes. The motes form a mesh network amongst themselves automatically. The host sensor distributes TinyDB queries over the motes. TinyDB is a program running on TinyOS on the motes that implements a SQL-style query, which allows for streaming data in addition to static relational data, across a network of motes. The query is formed, distributed, and then collected by the host sensor. Query results are reported over a COM port to the laptop. For details of this process, refer to Sam Madden's Ph.D. thesis: "The Design and Evaluation of a Query Processing Architecture for Sensor Networks" [13].

A Java Program called SensorInterface running on the laptop is analogous to the ReaderInterface program. The SensorInterface program interfaces with the host sensor to inject TinyDB queries over the sensor network and collect query results. We focus on the example of temperature. To collect temperature readings from the sensor network every second we inject the TinyDB query: "SELECT nodeid, temp FROM Sensors SAMPLE

PERIOD 1s". The SensorInterface program collects the results of this query and forms tuples of the form (temperature, SensorID, timestamp). Note that the temperature is unitless; it must be converted to known units. The tuple is specifically represented by the string "(temp, SensorID, timestamp)". The temp and SensorID values are integers, and the timestamp is the string representation of a Java Date object. The stream generated by SensorInterface represents the node S_T of the query tree shown in Figure 2.1.

5.3 Filtering and Conversion

We have described our implementation of data sources S_r and S_T. We now describe our implementation of query operators D and F for the naïve cold-chain application.

A windows application written in C# running on the laptop called FilteringAndConversion (FC) implements query nodes D and F. This program listens on a specified port for RFID and sensor events in their string representations described above. That is, whenever ReaderInterface or SensorInterface generate a tuple, it opens a TCP socket on the specified port to localhost and sends the tuple string. On the other end, the FC module spawns a new thread to handle the tuple. As such, an internal TCP socket implements the query edges (S_r, D) and (S_T, F).

To implement query node D, the FC module performs differentiation on the RFID stream. For every tag seen it keeps a boolean tagx_in_range and a DateTime tagx_last_seen in memory. For every incoming RFID event, the tagx_last_seen variable is updated for the appropriate tag. If a tag is seen and its tagx_in_range variable is false, then the boolean is

switched to true, and a tuple representing tag entrance is constructed. On a separate thread, every t seconds, for each tag where tagx_in_range is true, if the tag was last seen more than t seconds ago, the tagx_in_range variable is set to false, and a tuple representing tag exit is constructed with the current time. Tag entrance and exit tuples have the following form: "('enter', EPC, readerID, timestamp)" and "('exit', EPC, readerID, timestamp)".

To implement query node F, the FC module converts the integer representation of the temperature read directly from the temperature sensors to units of degrees Fahrenheit of data type double precision. To perform the conversion, we use an explicit formula given by Crossbow: $1/T(K) = a + b*\ln(Rth) + c*\ln^3(Rth)$, where $a = .00130705$, $b = .000214381$, $c = .000000093$, and $Rth = 10000*(1023 - x)/x$, where x is the integer unit-less temperature reading. Refer to pp. 9-10 of [12]. Note that $T(K)$ is temperature in Kelvin. We convert to Celsius by $T(C) = T(K) - 273.15$, and finally to Fahrenheit by $T(F) = 1.8*T(C) + 32$.

After the conversion is performed, a new tuple is constructed. This tuple is given by "(temp, 'F', SensorID, timestamp)". The difference between the output tuple and the input tuple is that temp is in degrees Fahrenheit; the character 'F' denotes this. Figure 5.1 illustrates the software architecture on the laptop and attached devices.
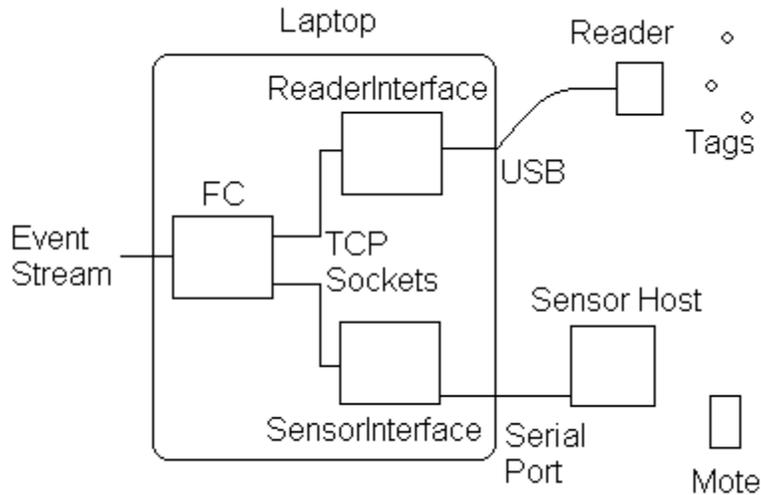
Figure 5.1 System diagram for software running on laptop interfacing with reader and sensor host. The ReaderInterface module collects RFID events from the reader over a USB port, and the SensorInterface module collects sensor events from the mote network via the sensor host over the serial port. These streams are forwarded to the FC module for filtering and conversion.

5.4 Aggregation and Dispatching

Once the FC module on the laptop has generated the filtered and converted data streams, it forwards them to a server. The server runs a Windows application written in C# called EventListener, which listens for tuple strings. Its first action upon receiving a tuple is to store it in an ADO.NET SQL database. An entire historical record of events is kept in this database. In addition, the database contains relational information about EPC numbers, readers, sensors, and locations. After a temperature event is stored, it is dispatched to a cold-chain monitor to determine potential cold-chain violations. This is described in more detail later in this chapter. Figure 5.2 illustrates the software architecture on the server.
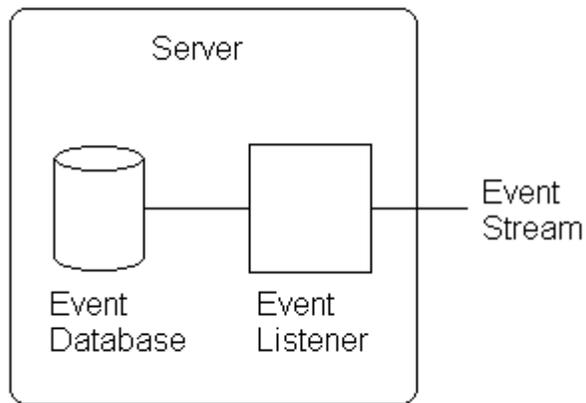
Figure 5.2 Server collects event stream from laptop. EventListener module first stores data into EventDatabase, and then performs real-time processing to generate cold-chain events.

5.5 Database Schema

The database on the server represents the state of the world as we can measure it, and all of

its measured previous states. The question is how to represent the world as a database

schema. A stripped down version of the database structure is shown below:

Location
       location_id (int)
       description (varchar)

Object
       object_id (varchar -- EPC number, will get exact format)
       description (varchar)

Reader
       reader_id (int)
       current_location_id (int)

Sensor
       sensor_id (int)
       current_location_id (int)

RFIDEvent

rfid_event_id (int)
reader_id (int)
timestamp (DateTime)
location_id (int)
object_id (EPC)

SensorEvent
sensor_event_id (int)
sensor_id (int)
timestamp (DateTime)
location_id (int)

Temps
sensor_event_id (int)
temp (double)

Lights
sensor_event_id (int)
light (double)

The Location table represents locations that can be either fixed in space or fixed to some reader or sensor that is mobile. Objects refer to things that have attached RFID tags, and they are uniquely identified by the EPC number on the tag.

The Reader and Sensor tables simply identify readers. The reader_id and sensor_id uniquely identify readers and sensors respectively. The curr_location_id field identifies the *current* location of a reader or sensor. That is, a reader or sensor may be moved to another location.

The RFIDEvent table captures the RFID tuple (EPC, ReaderID, timestamp). Note that the location_id field is the location of the reader generating the event at the time that the event is generated. To capture the location history of a particular reader, query the RFIDEvent table matching the reader.

The SensorEvent table is analogous to the RFIDEvent table; however, since sensors in our implementation can measure multiple attributes, such as temperature and light, etc., then we use a generalized SensorEvent table as a place-holder for the more specific Temps and Lights tables. This extends obviously to an arbitrary number of attributes.

5.6 Track and Trace

Given this database schema, a SQL query tracing the known history of object x is given by: "SELECT reader_id, timestamp, location_id FROM RFIDEvent WHERE object_id = x ORDER BY timestamp DESC". To obtain reader and location descriptions, simply query Reader and Location tables on reader_id and location_id respectively. A track of object x is simply the first record returned by the above query. Thus, a single SQL query statement implements Track&Trace as we have defined it. Figure 5.3 illustrates the query tree and machine graph for Track&Trace.
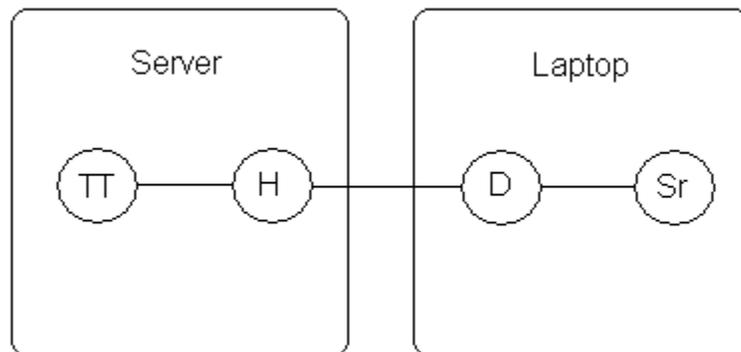

Figure 5.3 System architecture for Track&Trace application, showing query tree mapped onto machine graph. RFID stream S_r forwards read events to differentiation operator D on the laptop. Filtered read events are shipped to the server to be stored in historical database

H. The Track&Trace application is represented by query node TT, which queries H.

Query node S_r is the RFID stream collected on the laptop; D is the differentiation operator which forwards the filtered RFID stream to the laptop; H is the EventListener on the server that stores RFID events in an historical database. Note that H does not produce a stream, rather it makes available a relation of historical RFID events. Query node TT implements the Track&Trace query which makes calls to historical database H.

5.7 Cold-Chain

Cold-chain differs from Track&Trace in that events are pushed to the user. In Track&Trace, data is being gathered and stored, but results are only reported when a user explicitly executes a query. This type of query generates a set of results at one time each time the query is executed. With cold-chain, on the other hand, a user specifies a *continuous query* to the system, and results are reported as they occur. For example, if a user wishes to know if object x is ever exposed to a temperature above 75 °F for any period of time, then he or she specifies this to the system, and the user is notified of query results (i.e., temperature violations for object x) as they occur.

We implement a continuous query in the following way. A user specifies an algorithm A to run when condition C is met on the event stream. In addition, for this particular continuous query, a user has memory M shared by A and C. In our case, the event stream is the input stream to EventListener, which consists of filtered RFID events and converted temperature events. Both A and C are source code. They have access to the EventDatabase and live

event stream and may store information in memory M. For every tuple that arrives on the input, C is run and returns a boolean. If C, then run A. A returns query results and reports them to the user.

Consider the naïve cold-chain application. Suppose a user wishes to know if object x is ever exposed to a temperature greater than 75 ºF for any period of time. The only type of tuple on the input stream that can allow for such an event is the tuple ((enter, x, ReaderID), timestamp). The condition code C simple checks for tuples of this type and sets a boolean stored in memory M, tagx_visible = true. C also sets a variable tagx_location_id = location of the reader to which x is visible. C returns true in this case. In addition, if the tuple received is a temperature event from the location of tagx_location_id and tagx_visible = true, then C returns true as well. Otherwise, C returns false.

When C returns true, A is run. In our naïve cold-chain implementation, A checks if sensors in the current location of object x report temperatures greater than 75 ºF. If so, A reports a violation.

Conclusion

We have walked through a hierarchy of layers that comprise a system built atop RFID readers and telemetry sensors. We began with an exploration of the physical layer, discussing some physical properties of sensors and alluding that these properties effect data at higher layers. We next proposed a data structure to describe and query streaming and static data, which we refer to as the data layer. The data structure allows for general data models and can be made to encapsulate the physical realities of sensors.

These discussions are of course only the groundwork for the more novel research relating to the filtering and application layers. In the filtering layer, we introduce the notions of inference and differencing as a way to condense data streams and minimize communication. Furthermore, we propose an efficient way to sample temperature data in the context of the Cold-Chain eventing application. In the application layer, we describe how to build distributed systems atop RFID and sensor data, and we explore optimizations of our architecture in terms of bandwidth, computational capacity, and transparency (reconfigurability). Lastly, we illustrate our research with a specific implementation of the Track and Trace and Cold-Chain applications.

It is clear that building systems structured around RFID and sensor data is quite involved. These systems span a broad hierarchy of layers and are highly distributed by nature, which render them challenging to build. Our ultimate goal is to build robust and modular systems that can adapt to chaotic business environments and provide the desired functionality to end users. A layer above the application layer is the *user interface layer*, which was not

discussed in this thesis. The user interface layer is very complex, for it interacts with humans as part of an extraordinarily complex business environment. For a more detailed discussion of the UI layer, refer to the thesis of Chaitra Chendreshekhar [14].

Bibliography

[1] http://www.national.com/appinfo/tempsensors/files/TinyTempSensors.pdf

[2] http://www.maxim-ic.com/appnotes.cfm/appnote_number/237

[3] Stanford Stream Project

[4] SQL source

[5] Aurora paper

[6] http://www.cl.cam.ac.uk/~mgk25/iso-time.html

[7] http://www.itsc.org.sg/synthesis/2004/3_EPC.pdf

[8] Brock

[9] Shelf-life model

[10] SOAP source

[11] http://www.tagsense.com/ingles/products/products/mini-1356-datasheet.pdf

[12]

http://www.xbow.com/Support/Support_pdf_files/MTSMDA_Series_User_Manual_7430-

0020-03_A.pdf

[13] http://astragalus.lcs.mit.edu/madden/html/thesis.pdf