Relational Subscription Middleware for Internet-Scale Publish-Subscribe

Yuhui Jin
Department of Computer Science
Stanford University
Stanford, California, 94305
1-650-723-6805
yhjin@stanford.edu

Rob Strom
IBM T.J. Watson Research Center
PO Box 704
Yorktown Heights, NY, 10598
1-914-784-7641
strom@watson.ibm.com

ABSTRACT

We present a design of a distributed publish-subscribe system that extends the functionality of messaging middleware with "relational subscriptions", to support timely updates to state derived from published messages while preserving high throughput, scalability, and reliability.

Critical to our design is our service guarantee of "eventual correctness". Eventual correctness is a weaker guarantee than the ACID properties of conventional databases, yet is useful enough to deliver state that is "just consistent enough".

A key component of our design is a *monotonic type system.* All states delivered to clients represent facts that are permanently true, that may be refined by future updates, but will never become false. The monotonic type system is used both to formalize eventual correctness, and as a basis for our implementation, which generalizes the "Guaranteed Delivery" protocol previously implemented in the Gryphon system.

We discuss: (1) our monotonic type system and relational subscription language; (2) eventual correctness; (3) the architecture of our implementation; (4) potential optimizations that form a basis for future studies.

Categories and Subject Descriptors

D.3.3 [**Distributed Systems**]: Publish-subscribe

General Terms

Design, Reliability, Languages, Event Distribution Systems

Keywords

Relational subscriptions, continuous queries, monotonicity

1. INTRODUCTION

With the proliferation of computers and communication networks, timely access to information is fast becoming a reality. A growing number of applications, such as stock trade monitoring, sports reporting, traffic condition monitoring, etc., require the system to push information to users based upon a subscription, rather than reacting to repetitive polls. Content-based publish-subscribe systems support efficient and scalable message delivery for those cases where the subscription specifies a filtered subset of the published messages. We propose an approach to extending publish-subscribe middleware to include subscriptions to changes to state derived from event histories.

1.1 Why "stateful" middleware?

Consider a user who is monitoring stock market trades and wishes to be notified of changes to the hourly low, hourly high and current price of a selected set of issues as soon as such information is known. Suppose that publishers publish stock quote events following the schema: [time, issue, price, volume]. In a pub-sub system like Gryphon, the middleware filters and routes events, but cannot derive new events. We call this type of pub-sub system *stateless*, because the processing of each event depends only on the data in that event. Any additional state derivation would have to be performed either by publishers prior to generating events, or by subscribers in response to receiving events. In this example, the client would need to respond to each stock quote event by updating the relevant state: the high, low, and latest price for each hour. In contrast, in a *stateful* pub-sub system, the middleware would incrementally maintain the states and deliver the updates, in response to multiple client subscriptions each having a form such as "notify me of changes to the hourly max, hourly min, and current of selected issues".

There are a number of situations where a stateful system would be more efficient than the stateless one:

- **Reconnection.** In a distributed pub-sub system, messages can be intermittently dropped due to transient failures or congestion. With "best-effort" delivery, there could be inaccurate state if the lost message happened to be either a high, a low, or the last trade of the same issue. A more reliable quality of service supported by Gryphon, namely "guaranteed delivery", assures that messages are delivered exactly once and in publishing order. Although "guaranteed delivery" quality of service may be appropriate for dealing with transient losses of messages, it can be very inefficient when dealing with a longer outage. Suppose, for example, that a client disconnects for 30 minutes. Then upon reconnection, the client would be flooded with 30 minutes worth of missing messages. A stateful middleware, knowing how the messages are used, would just

DEBS '03, June 8, 2003, San Diego, California.

deliver a fresh update to the high, low, and current price of each issue. A similar problem arises if a client joins the system with his subscription in the middle of the trading day and needs to "catch up" the state.

- **Alerts.** In a related application, suppose a client needs to maintain a list of "high traders" stocks that have traded over 500,000 shares in the last hour. Now even in the failure-free case, the client will be receiving unnecessary messages that do not change the subscribed state. Instead of sending all the stock quotes, a stateful middleware reduces message traffic by sending only the information that a new issue has become a high trader or that an existing issue is no longer a high trader.
- **Dynamic Subscriptions.** In another application, suppose the client needs to track the stock quotes of issues on the 'buy' list of another organization, where insertions and deletions from this 'buy' list are published as events from a separate topic. Expressed as a stateless pub-sub application, this would be a rather complex program. One subscription would be needed to obtain the events that determined the current 'buy' list. Each time the content of the 'buy' list changed, the subscription to the stock quote topic would be dynamically withdrawn and a modified one issued. Extra care is needed to make sure that no events were lost or duplicated during this change. In a stateful pub-sub system, dynamic subscriptions of this sort are simple applications of joins of events from multiple publishers.

If the service boundary between client and middleware were moved so that the responsibility for computing derived state falls to the middleware, there are multiple benefits:

- There can be reduced message traffic to clients. Depending upon the application, these benefits either apply during normal operation or during recovery.
- Client applications have less complexity with regard to initialization, recovery, or dynamic subscription changes.
- There are more opportunities for the middleware to optimize its workload based upon the consolidated set of subscriptions issued by all clients.

We therefore propose an extension to the Gryphon content-based pub-sub system to support *relational subscriptions*. A relational subscription is a continuous query over the event streams, expressed in a declarative relational subscription language. In addition, we propose techniques for our relational subscription middleware to carry over the design requirements of the original Gryphon system, namely, high data rates, many publishers and/or subscribers, widely distributed over an overlay network of brokers, and tolerance of broker and link failures.

1.2 Overview of Key Concepts

The key features of our model are: (1) our model in which published event histories are base relations and subscriptions are views; (2) a service guarantee called "eventual correctness" that maximizes the flexibility of the implementation by weakening ACID transactional properties yet promising just enough precision for the client; (3) relation values that represent *monotonic knowledge*. We give an overview and motivation for these key concepts in our design.

1.2.1 Relational Model

In our proposal, we formalize each publisher topic as a "source" relation representing an event history, where each tuple corresponds to an event. Each event represents a change to our knowledge of the world. Each source relation is keyed by ticks of "time."[1] Our type system is rich enough so that tuples can contain structured data such as nested embedded relations. Each subscription is a request to receive incremental updates to a derived view defined by a relational expression on one or more base relations or other views.

1.2.2 Eventual Correctness

Although some of the original published events in a pub-sub system may come from database systems (others may come from direct input from humans or sensors), the pub-sub system itself is not a database system. We do not want the system to implement heavyweight protocols in support of more consistency than the subscribers need. However, we want to promise enough reliable semantics so that applications can confidently react to the states they receive.

Consider an illustrative application – a sports scoreboard. (We have chosen hockey as the example; any sport with monotonically increasing score and some concept of "periods" will do for this example.) Reporters at each site distribute updates of game events; fans sitting in one stadium see a scoreboard in which the approximately current score of all games being played that day are available. Scores of different games may possibly be reported with different delays. This level of imprecision is acceptable to most fans, and is expected. However, some strong properties typical of database systems are not guaranteed:

- Scoreboards at different stadiums are not necessarily consistent.
- Scores of different games as seen on a single scoreboard are not necessarily synchronized. For example, if the NY-Detroit game score appears as 4-2, and the Chicago-Boston score as 5-3, there may never have been a time at which Boston had more goals than Detroit (because Detroit's third goal may have happened before Boston's third goal and its notification was received later).
- Not every intermediate state is guaranteed to be seen. For example, if the NY-Detroit score was 4-2, and I later see it as 5-4, I may not be able to answer the question of whether Detroit ever tied NY.

What this implies is that the full atomicity and consistency properties of database systems do not hold, and additionally that the properties of some continuous query systems such as the state as of each instant of time must be seen do not hold either. However, there are some properties that *can* be guaranteed in this application:

- The knowledge I receive is monotonic. For example, I know that if I see a score of 4-2, then at least 6 goals must have been scored in the game.
- Some scores may be labeled *final* and I will know that they will never change.

[1] This time typically represents the time at which the event becomes known to the system. When the events are themselves the result of transactions, they are typically the transaction commit times; for externally generated events, the times are assigned at the point of publication. The events might additionally contain attributes referring to other times.

- There may be some local consistency. For example, it may say that as of period 2, the score of the NY-Detroit game is (at least) 4-2, and I can rely on the fact that NY's fourth goal came at or before the second period.
- I will never see anything false. For example, I will never see a game go into overtime if it never did, or a score of 4-3 if the final score is 4-2.
- I will eventually see a state consistent with everything that happened. For example, if the score of the actual game now is 4-2, I will eventually see either 4-2 or some higher score. As a corollary, if the actual state is a *final* score, I will eventually see exactly that final score.

These properties promised to the viewer of a hockey scoreboard are called "eventual correctness" and will be formalized in a later section. Many stateful pub-sub applications have requirements similar to that of the sports scoreboard. (In fact, sports score reporting is one of the current applications of Gryphon.) For applications requiring more refined knowledge, it can be obtained by writing more refined queries. This is because all of the original events with their timestamps are in the source relations and can be extracted with an appropriate query. Usually such queries involve making explicit reference to time. For example, suppose that I actually did need to know whether there was a time when Boston had more goals in its game than Detroit had. Although the more usual "latest score of each game" query will not derive this knowledge, a more specific query "tell me at which times *t* (or at how many times *t*) was it true that Boston's latest score as of *t* exceeded Detroit's latest score as of *t*" will do.

1.2.3 Monotonic Knowledge Model

The monotonic type system is the key component of our design and the basis for establishing the eventual correctness guarantee. All source relations, and all views derived from source relations are monotonic. In our model, monotonicity means monotonic knowledge, not necessarily arithmetic monotonicity. Relations and views represent eternally true facts, some of which are currently unknown because the future has not happened yet. Values of relations and views begin in a state of total ignorance and evolve towards a state of more precise knowledge. In the case of source relations, initially each tick has an "unknown" value for each non-key column of its schema. As time passes, the tuple at each tick evolves from "unknown" to either a "silence" tuple (meaning nothing happened at that tick) or to an "event tuple"). That tuple is now "final", meaning it does not evolve further. But in derived views, it is possible for values to evolve multiple times. Consider the application that tracks the hourly high, low, and current price of various stock issues. The column representing the "high" price is a simple monotonic type: each value has either the form "$\geq x$" or "$=x$". For example, a value of the first form "≥ 80" means that the high price is *at least* 80 (but might be higher later). It represents the result of the aggregate operator **max** applied to the prices in less than an hour's worth of ticks, some of which have an unknown value (because they lie in the future or have not been delivered yet). A value of the second form "=80" represents the result of **max** when the hour has ended and all ticks have a known value (either some price or silence). This second type of value is a *final* value, analogous to a final hockey score. Applications may need to distinguish the cases of "the high is at least 80 but possibly higher" and "the high is exactly 80". The column representing the "low" price is a similar type, where the values have the form "$\leq x$" or "$=x$". Finally, the column for the "current price" is the value of the aggregate operator **latest**. It has a special form "*x@t*", which is ordered by *t* and means "last changed at time *t* and had value *x*." In our system, the relational subscription compiler computes the type of each column of each view based upon the expressions used to derive these views.

Notice that Boolean predicates (e.g. max(price) < 100) on monotonic values can change from being *temporarily true* (e.g. a current high of at least 80 is *temporarily* less than 100) to *finally true* (when the final value is less than 100) or *finally false* (when the final value winds up being 100 or more).

Rather than thinking of views as tables that change through time, it is preferable to model them as increasingly precise views of information that will not be known until the infinite future.

Representing actions for deletion of rows (modeled as a query over two relations such as finding the available flights if they are in the **Booked** relation and not in the **Cancelled** relation") or subtraction of integers requires computing a value that must be interpreted mathematically as monotonic. This turns out to be non-trivial and is not discussed here for space limitation. For more detail, please refer to our technical report [19].

1.3 Structure of the paper

The rest of the paper is structured as follows. Section 2 gives a more rigorous description of the relational subscription model, including the monotonic type system, the subscription language, and the definition for eventual correctness. Section 3 presents the architecture and summary of the protocols of the implementation. Section 4 describes related work, and Section 5 concludes with future work.

2. RELATIONAL SUBSCRIPTION MODEL

We formalize the relational subscription model as follows. Each publisher corresponds to a source relation representing an event history. An event history is an append-only relation keyed by ticks of real or virtual "time". As time passes, the unknown value will change either to a silence or to some event value. A subscription is a request to receive incremental updates to a derived view defined by a relational expression on one or more source relations and/or other views.

2.1 Relations

All relations in our model are (following Darwen and Date [11]) *relation variables*, i.e., entities containing relation values that can change over time. All relation variables are statically typed. A relation type determines:

- A *schema*: a collection of column names associated with column types. Each row of a relation is a tuple consisting of one value of the appropriate type for each column. Some types may include one or more special values with interpretations such as "unknown", "silent" or "deleted". The algebraic operations on all types are fully defined over all values including these special values.
- *Key* and *non-key columns*: The columns defined by the schema are partitioned into a set of key columns and a set of non-key columns. Non-key columns may be of relational type (that is, there is no "first normal form" requirement).
- Optional other constraints. Type analysis may impose additional constraints on values of a relation variable. An example of a constraint is the *history* constraint, obeyed by all source relations. The history constraint specifies that if the non-

key values keyed by time tick t are "unknown", so are the non-key values at all ticks keyed by tick t' where t' > t.

Mathematically, values of a relation variable are total functions from the Cartesian product of the domains of the key columns to the Cartesian product of the domains of the non-key columns. A consequence of totality is that there exists a tuple for every possible value in the domain of the key columns. What is normally thought of as an "absent row" is just a row all of whose non-key columns have values such as "unknown" or "deleted". In our formalism, insertions are modeled as the replacement of an unknown value with an actual event; and deletions are modeled as the replacement of an actual event with a deleted value. At the implementation level, special data structures are used to avoid physically storing these unknown or deleted values.

2.2 Relational Expressions

The relational subscription language is derived from traditional relational algebraic operations (**select**, **project**, **join**, **extend**, etc., together with usual aggregation operators such as **sum**, **max**, **min**, **count**, etc.) We also support specialized operations such as **merge** (a kind of union), and a selector for the "best-k" elements.

2.3 Subscription Graphs

In our relational subscription model, a set of relational subscriptions can be represented logically as a *subscription graph* – an acyclic directed hypergraph where each node is a relation and each hyperedge represents a view operation – a relational expression with one or more relations as input, and a single relation as output. The leaves of this graph represent the subscribed views. Figure 1 illustrates a subscription graph for a simple airline reservation notification application with two subscriptions, together with typical values of the relations. (In this picture, we depart from the usual depiction of relations, in which rows that are not present are not shown, in order to distinguish between "silent" rows shown with gray shading, and "unknown" rows shown with '?'. The invisible silent and unknown values have different mathematical treatments: for instance, an unknown value can evolve, whereas in this example a silent value cannot.)

In this example, there are two source relations: **Flights** gives various information about today's flights including the maximum seating. **Booked** shows the flight name and number of booked seats for each booking. A hyperedge connects **Flights** and **Booked** to the intermediate view **Available**. Its operation joins **Flights** and **Booked**, computing the number of available seats, and selecting flights that have at least one available seat. Notice that the flight NW44 has been "deleted" from **Available** because all 10 of its seats have been booked. Two hyperedges connect **Available** to subscribed views: **Cheapest** is a subscription for the 3 cheapest available flights to London Heathrow (LHR) – currently there is only one flight available. **Planned** is a subscription tracking the availability of flights AA141 and UA23.

The execution paradigm for the relational subscriptions is as follows. Initially, the value of each column of the tuple for each tick in which the value of the row for every tick is "unknown". As time goes on, tuples in each source relation evolve from an unknown value to a known value – either silence or an event. The derived views are then incrementally re-evaluated to reflect the new events, and these changes are propagated down towards subscribed views at the leaves.

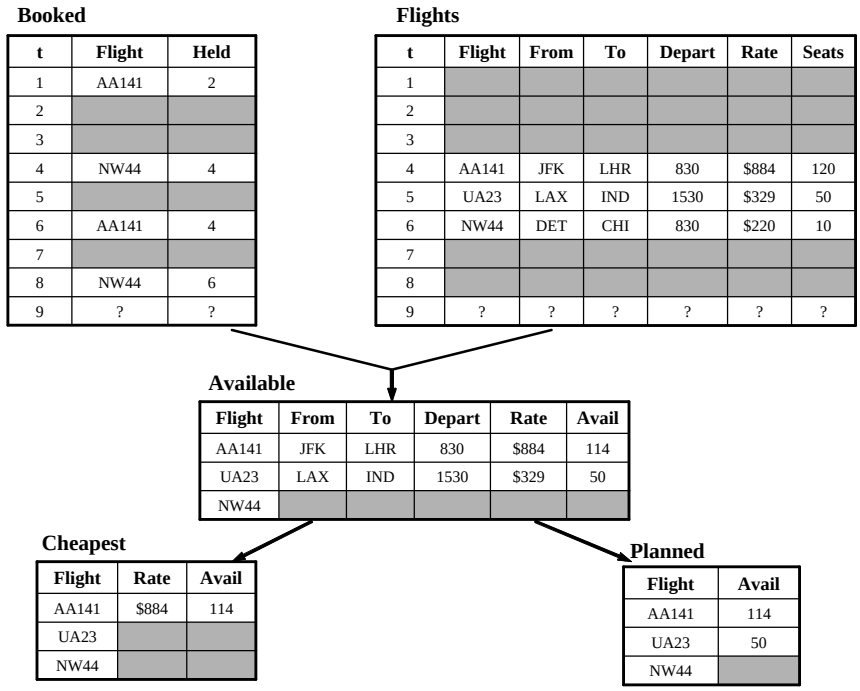

Figure 1. A subscription graph showing two source relations, an intermediate view, and two subscriptions

2.4 Monotonicity

Monotonicity is central to the design of the language, the type system, and the correctness specifications in our model. The underlying definitions and principles are the following:

- Any column of any tuple that can change value over time belongs to a monotonic type. The domain of every type includes a partial order relationship $\preccurlyeq$, interpreted as "may evolve to". Types whose values can't change are non-evolvable types. For these types, $v \preccurlyeq v'$ iff $v = v'$.
- Each scalar monotonic type is defined with its own partial order $\preccurlyeq$, and with a unique bottom value, the "unknown" value.
- Any value v in the domain of a type T, with the property that there exists no distinct value v' in the same domain such that $v \preccurlyeq v'$ is called a *final* value in T. A domain may contain many final values.
- A value v can only change by evolving to a higher value v'. Final values never change.
- Every relational type is monotonic. Given two values R and R' of a relation variable, $R \preccurlyeq R'$ iff for each tuple r in R, $r \preccurlyeq r'$, where r' is defined as the tuple in R' with matching key to r. The algebraic operations that derive views from relations preserve monotonicity.
- Every relation with an evolvable key column is also a monotonicity-preserving function. That is, if k and k' are values of a key column and $k \preccurlyeq k'$, (and all other key columns are identical), then for each non-key column, if v (respectively v') is that column's value in the row with key k (respectively k'), then $v \preccurlyeq v'$.

Because values evolve only in a single direction, and only to those values related by the partial order $\preccurlyeq$ of the type, it becomes possible for external observers of the system to interpret the tuples as "persistent knowledge".

Here are examples of some simple monotonic types:

Figure 2(a) shows a type of a column in a source relation that takes on a value from 0 to 3. The unknown value is represented as "?"; and the silence value is represented as "S".

Figure 2(b) shows a type of applying the *max* operator to a column whose type is shown in Figure 2(a). As each of the column values evolves from "?" to "S" or a number in [0..3], the result of *max* evolves monotonically. For example, the result is "≥1" if one of the values evolves to 1, and there is at least one other row whose for which the column values is still unknown; the result is "=2" when all the values are evolved to known values and at least one of them is 2. Notice that although conventionally, silent tuples and unknown tuples are both considered "absent" tuples, yet the result of max("S", "?", 2) is "≥2," while the result of max("S", "S", 2) is "=2."

Figure 2(c) represents the monotonic "integer range" type between 0 and 3. More specifically, this is the result type of applying the *count* operator to a column whose type is similar to the type shown in Figure 2(a); and where we know from the key that there are three rows in the group to be aggregated. For example, the bottom value [0..3] is produced when all three values are "?"; and the value [1..3] results from one of the values evolves from "?" to any non-silent value. The value [1..2] results if a second value evolves from "?" to "silent".

All relational operators *F* are monotonicity-preserving, that is, for any input v and v, if $v \preccurlyeq v'$, then $F(\ldots, v, \ldots) \preccurlyeq F(\ldots, v', \ldots)$. Details about the formal representation of the monotonic types can be found in [19].

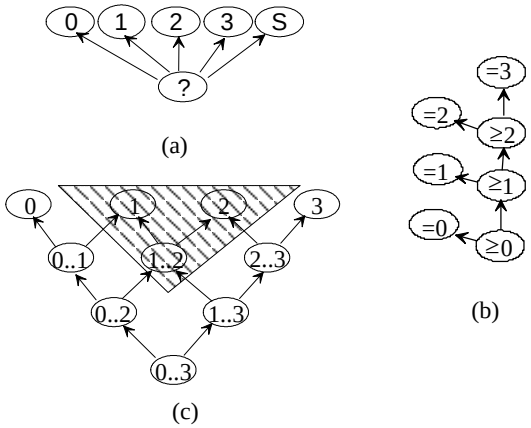

Figure 2. Examples of monotonic types

2.5 Eventually Correct Delivery

The eventual correctness property is a separate promise to each subscriber, not a global consistency property. Suppose S is a subscribed view, and its subscription depends on source relation variables R_1, ... R_n via a relational expression F (F is a composite expression from the expression for S and all the intermediate view expressions). We express eventual correctness in a distributed manner, i.e., in a way that does not require knowing a single time at two distinct sites. Let $r_i(t_i)$ denote the value of relation variable R_i at some local time t_i. Then the following properties hold:

Safety: *The subscriber never learns anything false*. Let s(seen) be a value seen at S. As the set of source relations continue evolving, applying F to the source relations will always yield a value that implies s(seen), i.e., a value higher than or equal to s(seen). In Figure 2(c), if s(seen) is [1..2], these values are in the shaded area constituting the "future light-cone" of [1..2]. Formally, there exist $t_1, \ldots t_n$, such that for all $t_1' > t_1, \ldots t_n' > t_n$, $s(seen) \preccurlyeq F(r_1(t_1'), \ldots r_n(t_n'))$.

Liveness: *The subscriber eventually learns everything that is true*. As before, let $r_1(t_1)$, ... $r_n(t_n)$ be values of relation variables R_1, ... R_n when it is time t_1 at R_1, ... t_n at R_n. Let $s = F(r_1(t_1), \ldots r_n(t_n))$. Then *eventually*, there will exist a value s(seen) at the subscriber in the "future light-cone" of s, that is, $s \preccurlyeq s(seen)$.

A consequence of the above rules is that if the publishers quiesce, (i.e., their values stop changing), all subscribers will eventually see the correct result.

3. ARCHITECTURE AND PROTOCOLS

The architecture for a relational subscription system is a generalization of the architecture of the Guaranteed Delivery (GD) service in the Gryphon system. The system is implemented on an overlay network of *brokers*. Although any broker may play any role, we distinguish the roles of *publisher-hosting* broker (PHB), *subscriber-hosting* broker (SHB), and *intermediate* broker. Events published at any of the PHB's are propagated through *knowledge graphs* where the resulting states are derived and delivered to the interested subscribers.

A knowledge graph is created after compiling the subscriptions. It serves as the execution plan for the subscription graph. A *relation* object corresponds to a node of the subscription graph, a *transform* object corresponds to a hyperedge. Figure 3

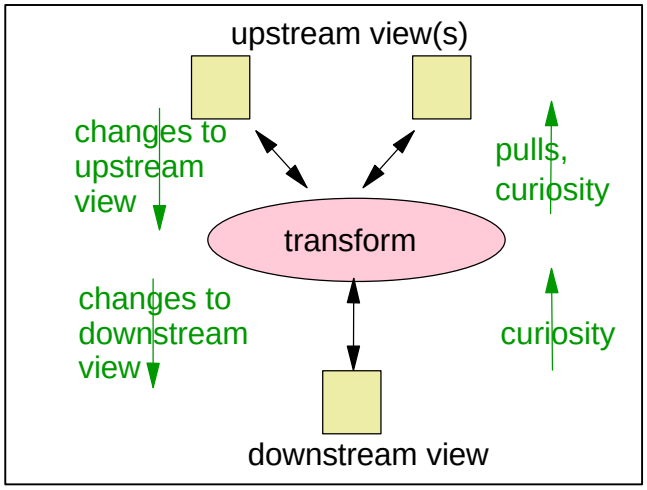

Figure 3. View and Transform Objects and the flows between them

illustrates the relationship between two upstream view objects, a downstream view object, and a transform, for a subset of a knowledge graph (e.g. the derivation of **Available** from **Booked**, **Flights** in Figure 1). In our prototype implementation, we used JavaCC to generate a compiler that reads a set of relational expressions, analyzes their types, and produces the knowledge graph. Because views and transforms communicate via asynchronous messages and the protocols tolerate loss and reorder of messages, these objects can be distributed across brokers without affecting the correctness of the protocol, although some placements will be more efficient than others.

The protocol for the relational subscription system consists of two parts: the basic protocol adapted from the GD protocol and an extended protocol addressing the broader set of relational operators. Guaranteed delivery for stateless subscriptions can be thought of as a special case of relational subscriptions in which the only operations are **select** and **merge**.

3.1 The Basic Protocol

At execution time, published messages enter the system at a publication endpoint in the PHB. Each message received from a publisher is assigned a tick number at the PHB. Adjacent ranges of silence ticks before and (optionally) after this tick are also generated. After logging the value to stable storage, each message is delivered to the source relation object associated with the event's topic. PHBs do not need to share a common clock, although it is advantageous for the clocks to be approximately in step. If events from multiple PHBs are merged, the merged relations might contain a range of unknown ticks between ranges of known ticks. This could cause delay in delivering messages to those clients that require messages in tick order, but will never cause safety or liveness violations.

Within the knowledge graph, view objects hold the state of derived relations ("knowledge"). Knowledge increments are passed downstream in the knowledge graph, incrementally updating downstream view objects according to the transform associated with each operator. Since in GD the only operations are **select** and **merge**, it follows that the only knowledge increment is to replace an unknown tick with a silence or a value tick, and that all views deriving from a source relation or from several merged source relations have the same signature. The transforms are straightforward: **select** is a filter, and **merge** simply passes value ticks through, but delays passing silence until all inputs have sent silence.

View objects also keep information about how urgently they need to receive knowledge. This information is called curiosity: *positive curiosity* indicates that there is or may be lost, missing, or incomplete information that is needed from upstream; *negative curiosity* indicates that certain kinds of information is no longer needed, or is temporarily not needed. Curiosity is passed upstream in the knowledge graph. An upstream object satisfies curiosity by delivering the requested state if it has it, or else by propagating the curiosity further up. Ultimately the logged source relations can always satisfy curiosity.

Because value ticks are sparse relative to silence ticks, each knowledge increment message consists of either a range of silence ticks, or of a single value message coupled with ranges of past and future silence ticks. A *doubt horizon* separates past ticks (all values or silence) from future ticks (all unknown). Because messages can be lost or reordered, gaps (unknown ticks between value or silence ticks) can appear. The protocol will initiate positive curiosity when gaps are detected, or whenever the doubt horizon has not advanced for a designated period of time. As values are delivered and acknowledged by subscribers, they are no longer needed, and negative curiosity is passed upwards. The data structure for storing value, silence, and unknown ticks is optimized for the case where gaps are infrequent and located close to the doubt horizon time.

3.2 The Extended Protocol

In a relational subscription system, the protocol becomes richer. Specifically:

- The relation objects can have different schemas. This is because we not only support select and merge, but also join of multiple relations or views.
- The transform objects need to be stateful or else need to read state from relations. The compiler generates the "partial derivative" for each expression relative to changes to the relevant columns of its input.
- Because we have to tolerate lost or duplicate messages, the state kept by the transforms needs to account for not only the current partial results, but also the input state that it reflects. For example, if a transform is tracking the sum of some column from a source relation, it needs to track which range(s) of rows are being summed over, so that: (1) it knows where gaps are, and (2) it can ignore retransmissions of rows that have already been included in the sum.
- The incremental changes to knowledge may be not only replacing unknown values with silence or data, but also advancing values along the monotonic partial order. They are often, but not always O(1).
- Curiosity and negative curiosity can pertain not just to time ticks, but also to values satisfying predicates. Changes to values of a relation can cause it to need to receive changes to only certain changes to upstream relations. For example, if all seats on a flight are booked, then the **Available** view does not need to track changes to other parameters of the flight.
- Negative curiosity can be temporary. For instance, in the predicate 'a and (b>0)', if 'a' is temporarily false, then changes to the value of 'b' are temporarily irrelevant and optionally may be suppressed by negative curiosity to save message traffic, but

then re-requested if 'a' should later become true. For this reason, the system needs to be tuned to provide an appropriate balance between the costs of wasted messages that will be ignored versus the delays of having to pull for information when it is not readily at hand.

- Some operations, e.g. "cheapest 3", will use a combination of push and pull strategy when distributed, because the algorithms that implement them "guess" that certain values will not be needed, and pull for them when they are later needed. The implementation of the view might decide to track changes to the cheapest 5 flights, for example, so that if two of the cheapest 3 become unavailable, it can immediately display the next two in order. But if a third flight becomes unavailable, it then will need to "pull" to find a new third-cheapest flight.

4. RELATED WORK

The work described in this paper extends our previous research on the Gryphon content-based pub-sub system with continuous queries over streaming data

The Tapestry system [20] introduces the notion of standing queries with *continuous semantics*. Continuous semantics is a stronger notion than the one presented here: under this semantics, a query is treated as if it were re-executed at every instant of time. Tapestry also introduces the notion of monotonic relations and monotonic queries based upon an ordering with the subset relationship. This is a more specialized definition of monotonicity than ours. They transform queries into a "standard form" and then convert queries that are non-monotonic to the narrowest query that is monotonic. This differs from our system in which the type system is expanded so that all queries are monotonic using a more generalized partial order. Their standard form involves selections without aggregations.

The Chronicle model [13] introduces the view update problem for views derived from append-only ordered sequences of tuples (chronicles) and traditional relations. Chronicles correspond to the source relations of our model. The emphasis is not so much on timely or fault-tolerant delivery as on accuracy and incremental computational complexity.

Recently, there are an increasing number of continuous query systems being designed and implemented. The OpenCQ system [14] explores system support for efficient evaluation of continuous queries driven from event streams. The NiagaraCQ system [9] allows a very large number of continuous queries registered over distributed XML data and apply dynamic re-grouping to share computation and provide scalability.

The CACQ system [18] extends an adaptive query processing framework called Eddies [6] to execute the disjunction of all continuous queries posed by the clients of the system. Psoup[8] extends the mechanisms developed in CACQ, allowing queries over historical data and intermittent connections. The idea is to treat query processing as a symmetric join between data and queries. The focus of both systems is main-memory query optimization under failure-free conditions.

The STREAM project [15] aims at developing a general-purpose Data Stream Management System (DSMS). Its focus has been resource allocation to approximate the answers under limited resources and algorithms to determine the memory requirement of queries.

The Aurora system [7] is designed to support stream monitoring applications. A recent effort [10] extends the centralized architecture of Aurora to distributed settings: the Aurora* architecture addresses a small-scale distribution within a single administrative domain; the Medusa architecture addresses a large-scale distribution across administrative boundaries. The optimization they proposed for load balancing is to repartition the network by sliding and splitting the trigger boxes across different machines.

The Cougar sensor database system [5] builds upon the previous work on sequence query processing and view maintenance over sequence data. Fjords [16] is an architecture for building query plans for a sensor network with a mixture of push and pull connections between modules, and provides a set of non-blocking and windowed operators to execute the plans. Madden et al [17] and Yao and Gehrke [21] propose algorithms and routing protocols to support in-network aggregations. Partial aggregation and packet merging are performed in a distributed yet synchronized fashion to reduce communication cost and save power.

In all the centralized continuous query systems studied so far, there has not been an issue of a correctness specification for query results – at the time of computing the updates, all the relevant information to this time point is available to the system; users either receive the precise answer or an approximation with a provable bound. However, in a distributed setting, because messages can be lost or out-of-order, a query at a site is not guaranteed to see all the necessary input data at the time of evaluation. Thus, an appropriate correctness guarantee such as ours is essential to achieve high scalability.

Previous research on information dissemination and pub-sub systems has been focused on content-based filtering of messages or documents. Xfilter [2] is an XML document filtering system that indexes XPath queries after converting them into a Finite State Machine representation. Febret et al. [12] describe an efficient algorithm that groups subscriptions based on their schemas and re-optimizes the grouping dynamically upon changes in subscriptions and event patterns. Finally, content-based pub-sub systems such as Gryphon proposed an efficient matching algorithm for event matching using a parallel search tree (PST) [1]. In [3], a distributed extension of the matching algorithm is discussed which serves as the basis of a multicast protocol for a network of brokers.

5. FUTURE WORK

A prototype implementation exists for a subset of our relational language. However, the interesting open questions concern how to exploit the flexibility implicit in our weak guarantee to enable the middleware to optimize delivery. Here are the main areas we are examining:

- Tradeoffs of pushing and possibly storing messages that may not be needed versus pulling, e.g., the example of the cheapest-3 transform in which a choice must be made to keep more than 3 rows and pull less often, or to keep 3 rows and pull whenever any of the 3 cheapest rows is deleted.
- Consolidation techniques for large numbers of subscriptions.
- Placement of relation and transform objects on brokers.
- Exploiting "selective curiosity" to quench message streams, or to convert stateful subscriptions to dynamic stateless subscriptions. (A typical example is: "show me events near my

current location," which can be implemented as a stateless subscription for events near my location until it changes.)

- Exploiting the ability to selectively checkpoint intermediate views so that in cases of "catch-up" or recovery, the message logs at the publisher are the *last* resort rather than the default.

6. REFERENCES

[1] Aguilera, M., Strom, R., Sturman, D., Astley, M., and Chandra, T., "Matching Events in a Content-Based Publish-Subscribe System", Proc., Principles of Distributed Computing, pp. 53-61, May, 1999.

[2] Altinel, M., and Franklin, M.J.: Efficient Filtering of XML Documents for Selective Dissemination of Information. In VLDB 2000.

[3] Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., and Sturman, D. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS), 1998.

[4] Bhola, S., Strom, R., Bagchi, S., Zhao, Y., and Auerbach, J., "Exactly Once Delivery in a Content-Based Publish-Subscribe System", *Proc. International Conference on Dependable Systems and Networks*, June 2002, Washington D. C.

[5] Bonnet, P., Gehrke, J., and Seshadri, P. Towards sensor database systems. In 2nd International Conference on Mobile Data Management, Hong Kong, January 2001.

[6] Avnur, R., and Hellerstein, J.M. Eddies: Continuously adaptive query processing. In ACM SIGMOD, Dallas, TX, May 2000.

[7] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring Streams – A New Class of Data Management Applications. VLDB 2002.

[8] Chandrasekaran, S., and Franklin, M., Streaming Queries over Streaming Data, In Proc. of the 28th VLDB Conference, Hong Kong, China, 2002.

[9] Chen, J., DeWitt, D., Tian F., and Wang, Y. NiagaraCQ: A scalable continuous query system for internet databases. In ACM SIGMOD, 2000.

[10] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. Scalable Distributed Stream Processing. In Proc. of the 1st Biennial Conference on Inovative Data Systems Research (CIDR 2003), Jan 2003, Asilomar, CA.

[11] Darwen, H., and Date, C.J. Foundation for Object/Relational Databases: The Third Manifesto. Addison-Wesley. June, 1998.

[12] Fabret, F., Jacobsen H-A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In Proc. of SIGMOD 2001.

[13] Jagadish, H., Mumick, I., and Silberschatz, A. View Maintenance Issues for the Chronicle Data Model, ACM PODS, pp. 113-124, 1995.

[14] Liu, L., Pu, C., and Tang, W. Continual queries for internet-scale event-driven information delivery. IEEE Knowledge and Data Engineering, Special Issue on Web Technology, 1999.

[15] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In Proc. of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003), Jan 2003, Asilomar, CA.

[16] Madden, S., and Franklin, M. Fjording the stream: An architecture for queries over streaming sensor data. ICDE. San Jose, CA, February 2002.

[17] Madden, S., Szewczyk, R., Franklin, M., and Culler, D. Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks. Workshop on Mobile Computing and Systems Applications, 2002.

[18] Madden, S., Shah, M., Hellerstein, J., and Raman, V. Continuously Adaptive Continuous Queries over Streams. In Proc. of SIGMOD, 2002.

[19] Strom R., Extending a Content-Based Publish-Subscribe System with Relational Subscriptions, IBM Technical Report. At http://www.research.ibm.com/gryphon

[20] Terry, D., Goldberg, D., Nichols, D., and Oki, B., Continuous Queries over Append-Only Databases. In ACM SIGMOD, pp. 321-330, June, 1992.

[21] Yao, Y., and Gehrke, J. Query Processing in Sensor Networks. In Proc. of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003), Jan 2003, Asilomar, CA.