Exploiting Event Stream Interpretation in Publish-Subscribe Systems

Yuanyuan Zhao
yuanyuan@cs.nyu.edu
IBM T.J.Watson Research Center
P.O.Box 704, Yorktown Heights, NY 10598
and
Courant Institute of Mathematical Sciences
New York University

Rob Strom
robstrom@us.ibm.com
IBM T.J.Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Publish-subscribe messaging middleware typically offers limited and low-level options for quality of service, such as best-effort delivery versus reliable delivery, or ordered versus unordered. We propose a new, high-level approach to specifying quality of service, in which the consumer specifies an *event stream interpretation* function that maps an event stream into a state that represents the consumer's semantics of the stream. Under this approach, the system may deliver either the subscribed event stream, or any alternative stream whose image under the interpretation function yields the same state. Event stream interpretation gives consumers the ability to more accurately specify the tolerable distortions of perfect message delivery, and gives middleware implementations the flexibility to use more efficient protocols for message delivery and failure recovery while preserving application safety.

We introduce an event stream interpretation language. We illustrate its utility by applying it to the problem of optimistic delivery of unlogged and out-of-order messages. We show how a publish-subscribe system can take an event stream interpretation program supplied by a subscriber and, using techniques derived from compiler technology, analyze it to determine which messages can be delivered optimistically, and which recovery messages must be delivered when an unlogged or out-of-order message is detected.

1. INTRODUCTION

1.1 Publish-subscribe systems

The background of this work is event distribution or "publish-subscribe" systems. Publish-subscribe systems improve flexibility by providing *anonymity* between producers and consumers. Producers (publishers) publish events (or messages) to a topic space, without specifying a particular destination; the messages go to all consumers who have subscribed to that topic space or (in content-based systems) to a selected subset of messages in that topic space satisfying a given predicate.

Although event distribution systems have evolved towards capturing more of the semantics of messages, the range of "quality of service" specifications still reflects their origin in low-level communications. In typical systems, such as the JMS model, subscribers may request such coarse qualities of service such as best-effort delivery versus guaranteed delivery. However, it is assumed that the application writer will supply a higher-level protocol to detect and recover from deviations from ideal delivery.

1.2 Event Stream Interpretation

In this paper, we propose a higher-level approach to quality of service, based on *event stream interpretation*. We assume that subscribers are aware of how to interpret the sequences of event messages they are requesting. We require the subscriber to convey to the message delivery middleware this interpretation through an interpretation function — a rule mapping an event stream to a state, which we call an *event stream interpretation*. This event stream interpretation determines the quality of service according to a *principle of equivalence:* the system is free to deliver any event stream equivalent to the subscribed stream, where event streams are defined to be equivalent if the interpretation function maps the streams to the same state.

1.3 Example

We illustrate our problem using the example of a stock trading application. In our example, multiple publishers report stock trades. We assume each stock trade event message is a simple pair of the form `<issue, price>`.

Suppose subscriber A is interested in events about all issues for the purpose of tracking the current price and the aggregate high of each issue. On the other hand, subscriber B is interested in the same events, but for the purpose of knowing the current price and the number of times the price dropped more than 5% in a single trade.

Suppose event stream *Stock-Events* is as follows:

```
(1) <IBM 150>
(2) <IBM 155>
(3) <IBM 145>
(4) <VZ 60>
```

(5) <VZ 58>
(6) <IBM 149>

Under ideal conditions, both subscriber A and B will receive the same stream, since their subscription predicates are the same. However, let us now consider "noisier" conditions, in which the network might drop messages.

Given the uses being made of these messages, we can see that a quality of service specification expressed in terms of low-level communication properties is not likely to be helpful. If A and B had to choose only between guaranteed delivery and best-effort, they would each be forced to choose guaranteed delivery, since A cannot afford to lose message (2), (4), (5), or (6), and B cannot afford to lose either (2) or (3), (5) or (6).

But in this example, Subscriber A can afford to lose messages (1) and (3), and Subscriber B can afford to lose (1) and (4). If we extend the possible distortions to include out-of order messages, either subscriber can permute an IBM message with an adjacent VZ message. However, subscriber A, but not subscriber B, can permute messages (2) and (3). Clearly it is not sufficient merely to specify whether total order is required or whether message reliability is required. Instead, we propose to use the equivalence principle to determine which permissible distortions of the actual event stream may be delivered.

In the paper, we argue that this is not only a theoretical benefit — the middleware, provided it is made aware of the interpretation function, can exploit it to perform optimistic delivery of messages.

1.4 Applying Event Stream Interpretation

In order for our proposed approach to be practical, the middleware must provide:

- A convenient language for expressing interpretation functions;
- An analysis tool for exploiting these functions to improve the performance (latency and/or bandwidth) of protocols based on the analysis.

In a previous paper [3], a restricted form of event stream interpretation was applied to the problem of recovering a meaningful event stream after a loss of connectivity. In this paper, we concentrate on how event stream interpretation can improve performance by enabling *optimistic delivery*.

Optimistic delivery means delivery before the system knows that the message it is delivering is the correct message to deliver next. We call such messages "in doubt."

There are two cases that we will consider in this paper:

- **Unlogged/uncommitted messages:** It is possible for a message to arrive at a subscriber that is potentially a "bogus" message that is not part of the topic stream. One way that this could happen is if a failure causes a message to be lost before it is logged to stable storage, but it has been optimistically sent through the network towards the subscribers. Another way is if an event is published as part of a transaction that aborts after the published event is sent.
- **Out-of-order messages:** Suppose subscribers have requested uniform total order. It is possible, because of non-FIFO communication, such as messages being sent along multiple paths, or because messages are dropped and later retransmitted, for messages to arrive out of order. If a subscriber receives for instance, messages m1, m2, and then m5, then the subscriber may not know whether m3 and m4 are missing because they were filtered out, or because they were delayed or lost. If m5 arrives without confirmation that it is indeed supposed to be delivered next, it is "in doubt." If it is delivered anyhow, it may happen that later m3 will arrive. The case of *dropped messages* is handled by treating them as messages which are retransmitted and which arrive out-of-order. In some cases, it will be determined that physical retransmission does not have to occur because the dropped messages have already been superseded.

Without event stream interpretation, all in-doubt messages must be buffered until the doubt is removed via later completion of some protocol; otherwise there is the danger that an unlogged message may never be made stable, or that a message arrives that belongs earlier in the event stream.

In a system using the principle of equivalence, it may often be safe to optimistically deliver an in-doubt message without waiting. Such delivery is safe provided that if the message later turns out to be incorrect, there exists a *recovery sequence*. A recovery sequence is a set of messages which when appended to the delivered stream will yield the same semantic interpretation as if the "bogus" message had not been delivered or the out-of-order message had been delivered in the proper order. An analyzer enabling optimistic delivery must determine under what conditions a message may be optimistically delivered, and how to compute the recovery sequence in case the in-doubt message is determined to be erroneous.

As applied to our example, it is safe to optimistically deliver to subscriber A any possibly unlogged message corresponding to an event other than a new high; if it turns out to be unlogged, it can later be canceled by delivering a new message for the same issue with the correct current price. Notice that the set of messages that can be optimistically delivered to subscriber B is different: a message that triggers a 5% drop alarm cannot be delivered unless both it and its preceding message are known to be correct.

Similarly, with respect to potentially out-of-order messages, using subscriber A's interpretation function, it is always safe to deliver messages optimistically, although the recovery sequence depends upon which message is out of order. For example, if message (6) is received first and message (5) arrives late, the recovery action is simply to deliver message (5) — the two messages commute. If it is message (3) that arrives late, the recovery action is to discard the message — it is superseded. And if it is message (2) that arrives late, the recovery action is to deliver message (2) and then repeat message (6).

The reader may have noticed that the principle of equivalence only requires "eventual correctness" of the state after recovery is complete, and does not exclude the possibility that a partially delivered message stream produces a state that does not correctly correspond to any prefix of the sent message stream. This may seem problematic in the case where the subscribing application may wish to take some irreversible action on the basis of a partial message stream. In such cases, the application writer should specify an event interpretation program with a *monotonic* component of the

state. For example, he may specify that part of the state is to be computed as the count of the number of times that some critical sequence (such as a drop in price) has occurred. Because of this monotonic component, just enough optimistic delivery will be suppressed so that the application can safely take action when the count reaches a certain value, since no recovery actions can ever decrease the count.

The goal of this paper is to:

- introduce a language for specifying event interpretation functions,
- show how to build into the middleware a tool which takes such an interpretation function supplied by the subscriber, and analyzes it to
  - determine under what conditions messages that are possibly unlogged or possibly out-of-order can be optimistically delivered, and
  - for those cases, determine how to compute the correct recovery sequences to deal with the cases when the optimistic delivery turns out to be wrong

This paper deals only with how to analyze the conditions where optimistic delivery is possible. We do not address the trade-offs between the benefits gained by optimistic delivery and the cost incurred by recovery from an erroneous delivery. Intuitively, message delivery middleware will perform optimistic delivery when the delivery is highly likely to be right, and where timeliness and eventually accuracy are more important than continuous accuracy. The benefits gained vary according to the capabilities and the requirements of the client.

1.5 Summary

In section 2, we formalize the notions of equivalence and provide a language for specifying interpretation functions. In sections 3 and 4, we present techniques based upon compiler optimization for analyzing an interpretation function to determine under which conditions possible unlogged or out-of-order messages may be delivered optimistically, and what recovery sequences to generate. We then present related work and conclusions.

2. AN EVENT INTERPRETATION MODEL

In this section, we present a model that facilitates our higher-level approach to specifying quality of service through event interpretation and the principle of equivalence.

This model includes: (1) a type system for expressing the structure of events and event stream interpretations; (2) an imperative language for expressing interpretation functions as incremental programs that apply an event to a current state to produce a new state; (3) a notation for expressing annotated histories representing the execution sequences triggered by a set of events.

2.1 Data Type System

We assume that each message communicates a single event. Events and states have a type. In this paper, we restrict the possible types to the following:

1. Base types: the usual base types of a programming language – numbers, booleans, and strings, with the usual scalar operations
2. Tuples: declared by {`f1: type1, f2: type2, ...`}, with operations component selection `v.f1` and component assignment `v.f1=...`
3. Unbounded Arrays: declared by `elementType[indexType]` with operations component selection `v[i]` or component assignment `v[i]=...`. We assume that if an element is never stored, it has a default value of the element type
4. Variants: declared by `oneof` {`case1: type1, case2: type2, ...`} where each case is a distinct type, supporting selection `case case1 of v`{SS} (where SS is a sequence of statements, and assignment `v.case1=...`

Tuple and array types are "polymorphic" as there could exist multiple views on either the entire variable or any consecutive segment of it. The operations on them will not conflict as long as they are on distinct non-overlapping views.

2.2 Event Interpretation Programs

An event interpretation program is an imperative program that specifies how each event changes the subscriber's state. The interpretation program transforms the state from the previously received events, plus the current event into a new state. The event interpretation program is therefore an incremental expression of the interpretation function; the interpretation function consists of the repeated execution of the event interpretation program for each event, starting with some initial state.

The set of temporary variables the program uses combined with the subscription state is called the *augmented state*. Each operation f is a function from augmented states to augmented states, denoted as $f : S \rightarrow S$. We denote the result of applying operation f to augmented state S as $f(S)$. An operation can be a basic operation such as an assignment or increment, or it can be a complex function that operates on a number of arguments. The arguments can be any of the types such as an array or a segment of an array. We assume the arguments are denoted "IN", "OUT" or "INOUT" to specify the operation's use of it.

The following constraints are imposed on an operation:

1. an operation updates exactly one augmented state entity;
2. an operation reads at most one entity in the subscription state, but it may read an arbitrary number of temporary variables;
3. an operation can both update and read the same subscription state entity.

We define operations to *conflict* if they share some arguments and at least one of them updates a shared argument. As we will see later in the paper, conflicting operations are possibly not commutative.

Although each operation may access only one state variable, an arbitrary number of temporary variables may be used.

We are now in a position to formally introduce the notion of an *event interpretation program*. An event interpretation program is a sequence of statements. Each of the statements is either:

- An operation or a NOP (Null OPeration); or
- a conditional statement of the form:
  `if b then SS1 else SS2`

where SS1 and SS2 are sequences of program statements, and b is a predicate.

- a variant conditional of form `case x of v {SS}`, where SS is a sequence of program statements, and v is a variant having case x, meaning if v is in case x then execute SS in a context where v is known to have the type associated with case x.

Note that our language does not allow explicit loop constructs, however, complex functions can be used for tasks that need loops such as searching an array.

The following is the specification for subscriber A in Example Section 1.3:

EXAMPLE 1. *Subscriber A subscribed to events of the following schema:*

```
{issue:string, price:float}
```

The subscription state definition is:

```
s:{max:float, cur:float}[string]
```

The state specification tells that state variable s *is an array indexed by a string (the stock issue), each element having two fields* - max *and* cur. *The event interpretation program of subscriber A is:*

```
e:
  s[e.issue].max=max(s[e.issue].max, e.price);
  s[e.issue].cur=e.price;
```
◇

2.3 Histories

The actual execution history of event interpretation programs determines the client state change. We formally define a history as following:

DEFINITION 2.1. *A history $H = P(E, S_0)$ is a sequence of operations instantiated from event interpretation program(s) P under the direction of an event stream E and the starting state S_0; each operation is annotated by a condition clause under which the execution path is taken. We call H the derived history of E under S_0, and E the generating stream of H.*

The derived history of E must be *serial* in that the operations derived from different events do not interleave. We also use notation $H = \langle f_1, ..., f_n \rangle$ to denote a history H in which operation f_i precedes $f_{i+1}, 1 \leq i < n$, and applying H on initial state S_0, denoted as $H(S_0)$, is $f_n(f_{n-1}(...(f_1(S_0))))$.

Intuitively, a history records the sequence of operations that are actually executed and the conditions satisfied. The annotations are used to ensure that no transformation would violate the annotation. The default annotation of an operation is a boolean value "TRUE". Every "`if`" or "`case`" statement is instantiated to one of its choices, and the annotation is the logical "AND" with any nesting conditions. For example, from the following statement

```
if (expr1) then OP1
else OP2
```

two possible paths could be taken:

```
OP1 (expr1)
OP2 (not expr1)
```

Similarly statement `case x of v {SS}` generates history `SS (v.x)`.

Nested statement sequences, such as SS1, SS2 or SS are similarly resolved into sequences of operations until the history contains only annotated primitive operations.

Now we are ready to introduce a set of *principles of equivalence.*

DEFINITION 2.2. *Two histories $X = P(E_X, S_X)$ and $Y = P(E_Y, S_Y)$ are equivalent, denoted as $X \equiv Y$, if $S_X = S_Y$, and $X(S_X) = Y(S_Y)$.*

DEFINITION 2.3. *Two event streams E and E' are equivalent regarding an initial subscription state S_0, denoted as $E \overset{s_0}{\equiv} E'$, if for a set of event interpretation program P where E and E' are both defined, $P(E, S_0)(S_0)) = P(E', S_0)(S_0)$.*

The following corollary is immediate:

COROLLARY 1. *Two event streams E and E' are state-S_0-equivalent ($E \overset{s_0}{\equiv} E'$) iff their derived histories are equivalent, i.e., $P(E, S_0) \equiv P(E', S_0)$.* □

Using P to specify quality of service means that for initial state S_0 and event stream E, we are permitted to deliver any E' such that $P(E, S_0) \equiv P(E', S_0)$. The challenge is to find the equivalent event stream with desirable properties. For instance, for optimistic delivery, the problem is to determine whether for any E and any perturbation E_p that may include out-of-order or unlogged events, there exists an equivalent E' whose prefix is the stream E_p already delivered optimistically. Corollary 1 demonstrates that the search for such properties on an event stream is equivalent to a search on its derived histories. In the following sections, we will talk about the transformation techniques to generate equivalent histories with these properties.

2.4 Transformations

As established in Section 2.3, the task of finding an equivalent event stream can be achieved by transforming a history to an equivalent one. We provide a set of transformations that can implement this process.

The first transformation, adapted from [2], deals with dead operations. Specifically, a NOP is equivalent to any dead operation.

DEFINITION 2.4. *A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. An operation is dead if the variable it updates is dead.*

TRANSFORMATION 1. *Given initial state S_0, history $H = \langle f_1, ..., f_n \rangle$, and its generating event stream $E = \langle e_1, ..., e_m \rangle$, let $P = \langle f_{i_1}, ..., f_{i_1+k} \rangle$ be a segment of $k+1$ operations generated by interpretation program of e_i in E, If all the operations in P are dead, then $H \equiv H'$ and $E \overset{s_0}{\equiv} E'$, where $H' = \langle f_1, ..., f_{i_1-1}, f_{i_1+k+1}, ..., f_n \rangle$, $E' = \langle e_1, ..., e_{i-1}, e_{i+1}, ..., e_m \rangle$. E' is the generating stream of H'.*

The second transformation deals with the commutativity of two events within a history.

DEFINITION 2.5. *Given stream E and its derived history $H = \langle ..., f_i, s_1, s_2, ... \rangle$ where s_1 and s_2 are derived from e_1 and e_2 respectively, event e_2 is commutative with e_1 in H if each operation in s_2 is commutative with all those in s_1, and all annotations stay true.*

TRANSFORMATION 2. *Given initial state S_0, history $H = \langle ..., s_1, s_2, ... \rangle$ and its generating stream $E = \langle ..., e_1, e_2, ... \rangle$, where s_1 and s_2 are generated by e_1 and e_2 respectively, let $H' = \langle ..., s_2, s_1, ... \rangle$, and its generating stream $E' = \langle ..., e_2, e_1, ... \rangle$. If e_2 is commutative with e_1 in H, then $H \equiv H'$ and $E \overset{s_0}{\equiv} E'$,*

EXAMPLE 2. *A subscription to deposit/withdraw events of type* `oneof{d:int,w:int}` *has the following* interpretation program*:*

```
case d of e{s.balance += e.amt;}
case w of e{if (s.balance >= e.amt) then
               s.balance -= e.amt;
            else NOP}
```

For event stream `e1=d(25), e2=w(10), e3=d(15)` *and initial state* `s.balance=0`*, we have history*

```
s.balance += e1.amt; (e1.d)
s.balance -= e2.amt;
(e2.w AND s.balance >= e.amt)
s.balance += e3.amt; (e3.d)
```

By definition 2.5, event e3 commutes with e2. Applying Transformation 2, we have

```
s.balance += e1.amt; (e1.d)
s.balance += e3.amt; (e3.d)
s.balance -= e2.amt;
(e2.w AND s.balance >= e.amt)
```
◇

3. OPTIMISTIC EVENT DELIVERY

We model optimistic delivery as follows: Events e may arrive in doubt either as to their logging status or their correct sequence. In-doubt events that can be proven to be recoverable are delivered; other events are held.

Eventually, for a possibly unlogged message, a protocol will deliver either a positive or a negative *log acknowledgement* (LACK). A positive log acknowledgement means that the message is logged (or alternatively, that the transaction that published the message committed); a negative log acknowledgement means that the message is lost (or that the transaction that published the message aborted).

For a possibly out-of-order message, a protocol will deliver either a positive *sequence acknowledgement* (SACK), or a *negative acknowledgement* that some event e' should have been delivered before e. In this discussion, we assume that we can tell the order between any two arrived messages by comparing their sequence numbers. We also assume every message possibly knows the sequence number of its predecessor, otherwise, we call that message possibly out-of-order. The arrival of a message with a sequence number earlier than one previously received is implicitly a negative acknowledgement.

Positive acknowledgements are used to release held messages, and to free space; negative acknowledgements cause recovery actions to be taken.

3.1 Compensating Events

The simplest recovery action we consider here is the delivery of a *compensating event*. A compensating event of e, denoted as $\bar{e}$, is an event whose derived operations undo those of e.

For some event types, a number of techniques can be used to generate their compensating events given the interpretation programs. However, in the general case where the primitive operations involve arbitrary functions, it is necessary for the analysis tool to have access to a table of rules defining compensating operations. From such a table, the tool can then decide whether it is possible to generate compensating events. An operation might not compensatible as the action is irrevocable. Hence, an event does not have a compensating event if it incurs any non-compensatible operations. For such an operation, there is not an entry in the table. The compensating events should have the following properties:

- Compensatability - for event e and its compensating event $\bar{e}$ and any event stream E, $P(E, S_0) \equiv P(Ee\bar{e}, S_0)$;
- Conformability - $\bar{e}$ conforms to the schema of one of the event types the client subscribes to.

Compensatability only holds when the paired events are delivered together. If a new event intervenes between the unlogged event and its negative acknowledgement, it may not be safe to deliver this new event unless it commutes with the compensating event.

Conformability is required so that the action of optimistic delivery and recovery could be made transparent to the subscriber. From the subscriber's point of view, it simply receives an event stream it can interpret. The subscriber has no way to tell whether it's an optimized stream or the original stream. The compensating event should be reasonably light weight. For example, in some systems there may be an event which sets the entire state. Such an event can be used to compensate for anything. However, if the state is very large, it might be too expensive to include it as a compensating event unless the negative acknowledgements are rare.

In the remainder of this section, we will discuss techniques to statically analyze the event interpretation program. Static analysis determines, given the text of the event interpretation program, under what conditions an event is compensatible and if so what the compensating event is.

3.2 Analyzing Event Compensatibility

To analyze event compensatibility, the tool needs to know a table of compensating rules and the event interpretation program. These rules define the compensating operations for a set of basic operations and complex functions. For example, the compensating operation for a simple assignment is just another assignment of the previous value. If the initial value of state variable `s` is `v0`, no matter what the assignment is, the compensating operation is simply `s=v0`. For an operation of add-and-assign to a state variable, the compensating operation is a corresponding deduct-and-assign. It is trivially true that a proper assignment of the original value is always a compensating operation to any operation.

Figure 1 lists some of the compensating rules.

As a special case, if the compensated-for operation is dead, it does not need to be compensated for; alternatively, its compensating operation could be a `NOP`. Conversely, a `NOP` compensating operation means the compensated-for operation is dead.

To analyze the compensatibility of an event, its interpretation program is first factored out to a set of history trees, where each node represents an operation and the edge represents the condition under which this path is taken. Stored along with the operation in the node are its compensating operation and the condition clause of the compensating operation. If we are unable to find a compenating operation, that particular path is trimmed. This process is illustrated

$s.s1 == v0;$	$s.s1 = \_;$	$\_$	$\rightarrow$	$s.s1 = v0;$	(3.1)
$s.s1 == v0;$	$s.s1 \mathrel{+}= \_;$	$\_$	$\rightarrow$	$s.s1 = v0;$	(3.2)
$s.s1 == \_;$	$s.s1 \mathrel{+}= e.a1 + c;$	$\_$	$\rightarrow$	$s.s1 \mathrel{-}= e.a1 + c;$	(3.3)
$s.s1 == v0;$	$s.s1 \mathrel{*}= \_;$	$\_$	$\rightarrow$	$s.s1 = v0;$	(3.4)
$s.s1 == \_;$	$s.s1 \mathrel{*}= e.a1 + c;$	$e.a1 \neq 0$	$\rightarrow$	$s.s1 \mathrel{/}= e.a1 + c;$	(3.5)
$s.s1 == v0;$	$s.s1 = max(s.s1, e.a1);$	$s.s1 \geq e.a1$	$\rightarrow$	NOP	(3.6)
...					

Note: The first column on LHS of $\rightarrow$ specifies the current value, the second is the compensated-for operation, the third the constraint, and the RHS of $\rightarrow$ is the compensating operation that should be generated. $\_$ is wildcard.

Figure 1: Compensating Rules

as procedure $Expand(P)$ in Section *3.2.1* and in an example in Figure 2. To simplify the exposition, we show the whole tree, but in practice the tree can be partially generated as each path is analyzed, and each path can be discarded after it is no longer needed.

For each node in the history tree, there exists a compensating operation. However, the existence of compensating operations does not ensure the existence of a compensating event. For now, we only consider the case in which a compensating action is a single event. In order to derive this compensating event, all the compensating operations have to be pattern-matched to those of a single event. Each compensating operation requires the bindings of a number of free variables. For each leaf, the AND of the conditions along the edges from the root to that leaf represents the conditions that must have been satisfied whenever this path is taken. The AND of the condition clauses in the nodes on this path defines the necessary conditions under which a compensating event exists. A compensating event could only exist if these two logical ANDs do not conflict, i.e., the logical AND of them, which we call the *test set*, is satisfiable. A compensating event is generated using the bindings of free variable in the compensating operations and bindings performed during the satisfiability testing. This test set and compensating event are used at run time to determine whether an event can be delivered optimistically. The runtime test executes in constant time given the interpretation program and a compensating rule library.

For simplicity, the algorithm shown in Section *3.2.1* assumes there exists at most one compensating operations for each operation. In the case that the compensated-for operation is compensatible by more than one operation choices, i.e. multiple rules apply to a single compensated-for operation, the rules are taken in the order of their appearance to generate all possible compensating events.

Section *3.2.1* lists the analysis algorithm.

3.2.1 The Static Analysis Algorithm

The algorithm contains two main procedures. Procedure "Expand" factors out the event interpretation program to a set of history trees, where each path from the root of each tree to each leaf represents an execution possibility of the event's interpretation program. Procedure "GenCompensatibilityTest" takes the trees generated in the first process, and performs satisfiability test on each path from a root to a leaf. If the conditions are satisfiable, a compensatibility test is added into the predicates set. For any event arriving at runtime, it's compensatible if it satisfies any one of these tests. To simply exposition, we used a tree structure where many of the nodes are replicas of each other, and we used a batch like procedure where all the trees are expanded first, and satisfiability test is performed later. In practice, the trees can be expanded as each path is analyzed for satisfiability, and discarded when the analysis is completed.

Procedure Expand(P)

```
;;Expand program P into a history tree
;;return a set of (cond, node)
;;each represents the subtree taken
;;under certain condition
   s ← remove first statement in P
   ifP is not empty then
     children ← Expand(P)
     ifchildren is empty set then
       ;;remaining program is not compensatible
       ;;hence, whole P is not compensatible
       return empty set
     end if
   end if

   if s is an operation then
     make new node n
     n.op ← s
     ;;using rule table find a compensating operation
     o̅p ← find compensating operation of s
     if o̅p is NULL then
       ;; operation is not compensatible
       return empty set
     else
       n.o̅p ← o̅p
     end if
     n.children ← children
     return (true, n)
   else if s of form ``if b then SS1 else SS2'' then
     children1 ← Expand(SS1)
     foreach (cond, node) in children1 do
       (cond, node) ← (cond AND b, node)
     end for
     children2 ← Expand(SS2)
     foreach (cond, node) in children2 do
       (cond, node) ← (cond AND (NOT b), node)
     end for
     foreach leaf (cond, node) in
       trees (children1 ∪ children2) do
```

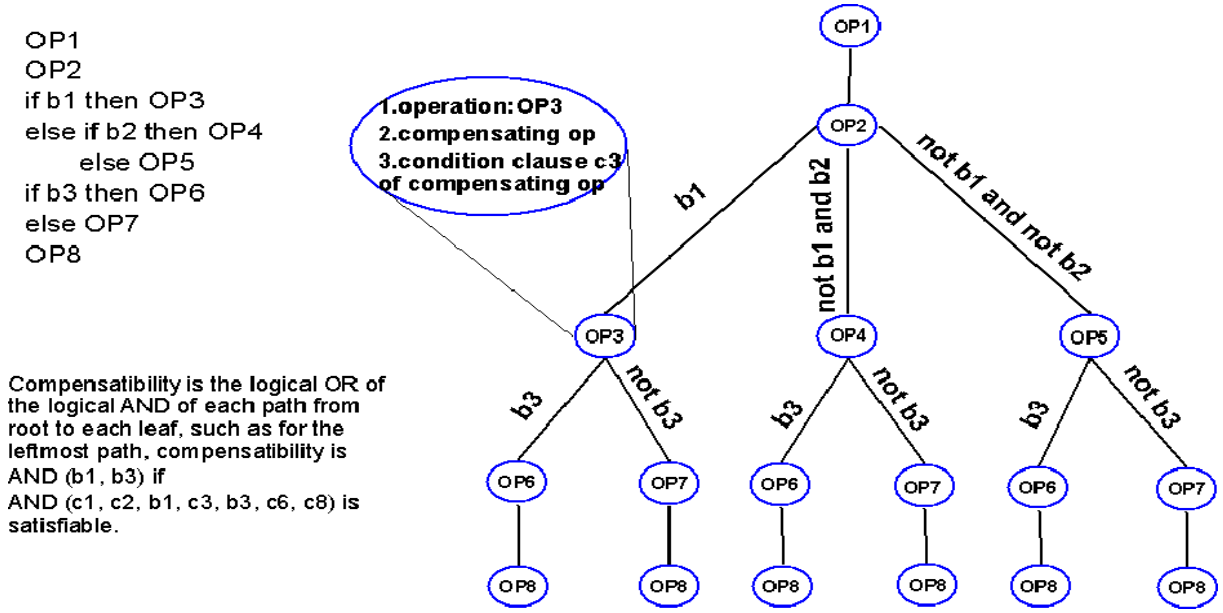

Figure 2: Analysis Tree of an Interpretation Program

```
      node.children ← children
    end for
    return children1 ∪ children2
  else if s of form ''case x of vSS'' then
    children1 ← Expand(SS)
    foreach (cond, node) in children1 do
      (cond, node) ← (cond AND e.x, node)
    end for
    foreach leaf (cond, node) in trees children1 do
      node.children ← children
    end for
    return children1
  endif
end Expand
```

```
Procedure GenCompensatibilityTest(Ts)
    foreach tree T in Ts do
      foreach path p from root to a leaf in T do
        conds ← AND (conditions of all nodes,
                     conditions of all ōp)
        if(conds is satisfiable) then
          initialize ē with the bindings
                     in compensating operations
          preds ← preds ∪ (conds, ē)
        end if
      end for
    end for
    return preds
  end GenCompensatibilityTest
```

A similar approach can be applied to generate commutativity information between events.

Let us look at how a static analysis algorithm would work as applied to the event interpretation program of subscriber A in example 1. Since there is no conditional statement, there is only one unconditional segment of the program. The compensating operations are `NOP` $(s.max \geq e.price)$ and `s.cur=s.cur0`, here `s.cur0` denotes the initial state value. The compensating operation set matches the pattern of the only event in this system, namely the event is compensatible with the predicate $s.max \geq e.price$, and the compensating event is an event with $e.price$ bound to $s.cur0$. This means that events with $(s.max \geq e.price)$ can always be optimistically delivered; others may not.

3.3 Optimistic Delivery and Recovery

In this section, we describe an algorithm (Figure 3) that performs optimistic delivery using the information obtained from the static analysis process, and recovers in case of a negative aknowledgement. The recovery incurs costs, thus an efficient system should perform optimistic delivery only when negative acknowledgements are rare. The criterion for estimating the tradeoffs between optimism and pessimism is a separate issue relating to performance and will not be discussed here.

In the algorithm, we divide the events into three categories. An event is called "potential out-of-order" when the system does not have positive knowledge of its immediate predecessor, i.e., the system cannot be certain there will not be a gap between this event and the received event with the greatest sequence number that is earlier than this event. An event is called "potential unlogged" when the system lacks a positive logging acknowledgement. An event is called "normal" otherwise. Note that a "normal" event here could still be out-of-order if one of its predecessors is out-of-order, but we call this event normal since it introduces no new gaps in the stream.

The algorithm operates on two stacks and one queue, s_ooo for delivery of potential out-of-order events, s_unlogged for potential unlogged events, and heldQ for events that cannot be optimistically delivered and hence must be held. The heldQ are maintained according to event sequence numbers. To simplify the presentation, we assume that an event is missing at most one of the acknowledgements. In case of an event with neither acknowledgement, the algorithm

performs both out-of-order and unlogged delivery, but the data is recorded at most once in the stacks or queues, and the event is delivered optimistically only when both of the out-of-order and unlogged handling will deliver the event. The algorithm buffers an event in heldQ that cannot be optimistically delivered and all events with a later sequence number. The algorithm also maintains the current state of the interpreted stream as well as state associated with events on the out-of-order stack. For reasons of space, we omit the updating of this state. For each event, the algorithm operates in linear time with regard to the number of elements in each stack.

4. SPECIAL HANDLING OF OUT-OF-ORDER EVENTS

There are situations where either a compensating event for an arbitrary "bogus" event does not exist, or where delivering it could be too expensive. In some cases, it is possible to determine a simpler recovery sequence for out-of-order messages.

The problem is formally represented as follows:

For starting state S, and an event stream $E = \langle e_1, ..., e_k, e' \rangle$, and $H = P(E, S)$, does there exist a history $H' = P(E', S)$, where E' begins with e', and $H \overset{s}{\equiv} H'$?

Intuitively, E' could be constructed from E by pushing e' ahead as long as e' forward commutes with other events in the path. Since $e_1, ..., e_k$ are unknown, they have to be considered as events of any possible types and values.

The histories derived from different orders of events could be totally different from each other because of different resolution of the conditional statements in the interpretation program. However, the behavior of programs with no conditional statements other than the variant case of the event is much more predictable. In the following we will introduce a set of transformations for these programs.

The commutativity transformation 2 could be used in the case of out-of-order events. However, it often happens that events only partially commute yielding a non-serial history in which some operations of an event are ahead of some events, where others are after. We introduce two transformations called idempotency and dead-operation injection in order to solve this problem.

As conventionally defined, idempotent operations are those that have the same effect when they are applied one or more times. Operation `s[issue].max=max(s[issue].max, e.price)` in example 1 is such an example.

TRANSFORMATION 3. *Given history $H = \langle s_1, f, s_2 \rangle$, and its subsequences s_1, s_2 and operation f, if f is idempotent, then $H \equiv H'$, where $H' = \langle s_1, f, f, s_2 \rangle$.*

TRANSFORMATION 4. *Given history $H = \langle s_1, s_2 \rangle$ and its subsequence s_1 and s_2, $H \equiv H'$, where $H' = \langle s_1, f, s_2 \rangle$, if f is dead in H'.*

In example 1, `s[e.issue].cur=e.price` can be inserted into any place before another `s[e.issue].cur=e.price`.

From the above mentioned transformations, we derive the following theorem:

THEOREM 2. *Given event e and its interpretation program P_e, where P_e contains no embedded conditionals, if there exist subsequences s_1 and s_2, such that $P_e = \langle s_1, s_2 \rangle$, and the following are true:*

1. for each operation f in s_1, f is idempotent and commutes with all operations in P_e;

2. for each operation f' in s_2 and any e' that could possibly precedes e, f' is a dead operation in $\langle f', F(e'), s_2 \rangle$ then e could be delivered out of order, and recovered by $e'e$.

EXAMPLE 3. *As an example, for the subscription system described in example 1, we have the following process:*

```
e': s[e'.issue].max =
      max(s[e'.issue].max, e'.price);
e': s[e'.issue].cur = e'.price;
e:  s[e.issue].max =
      max(s[e.issue].max, e.price);
e:  s[e.issue].cur = e.price;
```

Case 1: `e'.issue` $\neq$ `e.issue`
Event e could be delivered with recovery sequence of only e'.
Case 2: `e'.issue = e.issue`
By theorem 2, e could be delivered before any event, and recovered by `e'e`. ◇

5. RELATED WORK

The traditional approach for event delivery is through group communication, such as Virtual Synchrony [5] implemented in ISIS [4] and a number of other systems. In such systems, messages are considered as bit strings without semantic significance. A number of different delivery protocols are supported, each with different properties regarding order and consistency.

There are numerous emerging content-based event systems (e.g. [9]), in which messages have a defined type and subscriptions can query fields in these messages. In these systems, the middleware can exploit the common tests in these subscriptions to speed up the matching of events to subscribers [1]. Sun's JMS standard [10] supports a content-based interface. However, the qualities of service offered in these systems still reflect the same coarse range of choices as in earlier systems in which events were completely opaque bit strings. That is, users can specify, for instance, best effort versus guaranteed delivery.

Flexible notions of equivalent streams are exploited for supporting different qualities of service in specialized domains such as video streaming, but not for general event delivery systems as would be used in applications such as stock trading or electronic commerce.

Our earlier work [3] introduced the notion of an event interpretation function. The techniques presented in that work applied to a restricted form of language called *replacement form*, and only dealt with compression and not with optimistic delivery.

The concept of optimistic delivery and compensating events in our work is inspired by earlier work on compensating transactions that appeared in a number of works on database concurrency control and recovery such as in [7], [6]. However, they did not address the static analysis for generating compensatibility information, nor did they address the issue on performing optimistic delivery and recovery in conjuction with interfering events.

The problem of recovering from out-of-order events delivered optimistically has been dealt with in specialized con-

Normal Event e: Event not in Doubt

```
if s_ooo and s_unlogged both empty then
   deliver e
   return
end if
if s_unlogged not empty and e not commute
   with an ē on s_unlogged then
   put e on heldQ
   return
end if
if s_ooo not empty and e is compensatible then
   push (e, ē) onto s_ooo
   deliver e
   return
end if
put e on heldQ
return
```

Potential Out-of-Order Event e:

```
if e is compensatible and (s_unlogged empty or
   e commute with each ē in s_unlogged) then
   new segment marker on s_ooo
   push (e, ē) onto s_ooo
   deliver e
   return
end if
put e on heldQ
return
```

Potential Unlogged Event e:

```
deliverable ← false
if e is compensatible and commute with
   each ē in s_unlogged then
   deliverable ← true
   push (e, ē) onto s_unlogged
   deliver e
end if

if deliverable and s_ooo not empty then
   push (e, ē) onto s_ooo
end if
if not deliverable then
   put e on heldQ
end if
```

LACK on Event e:

```
Positive:
  if e in s_unlogged then
    remove (e, ē) from s_unlogged
  else ;; e in heldQ
    remove & deliver all e in heldQ that
           are no longer in-doubt
  end if

Negative:
  if e in heldQ then
    remove e from heldQ
  else ;; e must've been optimistically delivered
    remove (e, ē) from s_unlogged
    recovery by delivering ē
    if s_ooo not empty then
       remove (e, ē) from s_ooo
    end if
  end if
```

SACK on Event e:

```
Positive:
  if e on heldQ then
    remove & deliver no-longer in-doubt e in heldQ
    return
  end if
  seg←segment contain (e, ē) on s_ooo
  if seg is in bottom of s_ooo then
     remove seg
  else
     combine seg with previous segment
  end if

negative with e' before e:
  m←index of (e, ē)
  i ← m
  while e' commute with s_ooo[i].ē
    i ← i + 1
  end while
  deliver recovery stream ∑_{j=top}^{i} ē + ∑_{j=i}^{top} e
  handling e' and events after it
  if m is bottom of s_ooo then
     remove segment (e, ē)
  end if
```

Figure 3: Optimistic Delivery and Recovery

texts, such as computer-supported cooperative work systems. For example, the work of Ressel et al [8] deals with group document editors where events are insertions and deletions to segments of text, and where it is important that all users reach consistent states, despite the fact that some users may receive messages in an inconsistent order. In this work, it is necessary for the middleware to generate transformed operations to make the state consistent. These transformed operations behave like our compensating events. Unlike our work, this system exploits the fact that all consumers of events have a common application and a common event interpretation.

6. CONCLUSIONS

In this paper, we present an approach to more flexible quality of service in general-purpose event distribution (publish-subscribe) systems through the use of *event interpretation functions* and the *principle of equivalence*. The event interpretation function maps a subscribed event stream to a state representing the information that the subscriber wishes to maintain. The principle of equivalence allows the system to deliver a "distorted" event stream provided that it maps to the same state.

We show that this principle can be exploited in a number of ways. In particular, events can be delivered without waiting for logging acknowledgements or sequencing acknowledgements, provided that the event interpretation function allows those distortions in which unlogged or out-of-order events are followed by appropriate compensating events. We show how an analysis tool can examine an event interpretation function and determine under what circumstances such optimistic delivery is possible, and how to compute compensating events.

7. REFERENCES

[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on the Principles of Distributed Computing, Atlanta, GA*, May 1999.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison Wesley, 1986.

[3] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information flow based event distribution middleware. In *Proceeding of ICDCS'99 Middleware Workshop*, 1999.

[4] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM, 36(12):36-53*, 1993.

[5] K. Birman and J. T. Exploiting virtual synchrony in distributed systems. In *Proceeding of 11th ACM Symposium on Operating Systems Principles*, 1987.

[6] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems, 8(2):186-213*, 1983.

[7] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceeding of 16th International Conference on Very Large Databases*, 1990.

[8] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM 1996 conference on Computer supported cooperative work*, Boston, MA, 1996.

[9] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings, AAUG97*, September 1997.

[10] Sun Microsystems. *Java Message Service*. http://java.sun.com/products/jms.