

Information Flow Based Messaging Middleware for Building Loosely Coupled Distributed Applications

Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao

Abstract. *Messaging middleware supports the integration of distributed components of an application by collecting messages from information producers and disseminating applicable messages to interested information consumers. In this paper, we present a flexible new model, the Information Flow Graph (IFG), for specifying and implementing the flow of information in a messaging application. We show that IFGs can be implemented efficiently by reducing them to a form that can exploit efficient multicast technology that has been developed for content-based publish/subscribe systems. Furthermore, we describe techniques for using stateful operations in IFGs to derive trends, summaries, and alarms from published events and to automatically generate equivalent sequences of events.*

1 Introduction

Messaging middleware is growing in importance with the need to glue together heterogeneous, distributed, and dynamically changing components of large information systems. Typically in these systems, there are multiple producers and consumers of messages. The middleware performs the function of collecting messages from producers, filtering and transforming them as necessary, and routing them to the appropriate consumers based on need. This approach is currently being applied in domains such as finance, process automation, transportation, and mergers and acquisitions.

The Gryphon project at IBM research (<http://www.research.ibm.com/gryphon>) is advancing the technology of messaging middleware by developing a system to test new models, algorithms, and tools. In this paper, we describe our approach to messaging middleware based on the concept of *information flow graphs*. To motivate the Gryphon model, it helps to look at the evolution of messaging middleware through its earlier stages of subject-based and content-based publish/subscribe systems.

Contact author: G. Banavar, banavar@watson.ibm.com,
Phone: +1-914-784-7755, Fax: +1-914-784-7455.

G. Banavar, M. Kaplan, D. Sturman, and R. Strom are at the IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532.

K. Shaw and W. Tao visited IBM Research during the summer of 1998 from the Departments of Computer Science at Stanford University and University of Utah respectively.

Please note that the contents of this paper are concurrently being patented, and is provided to the reviewers under the usual professional understanding of confidentiality.

Messaging middleware has its origins in multicast and publish/subscribe technology. In pub/sub systems, information producers publish events to the system and information consumers subscribe to particular categories of events within the system. The system ensures the timely delivery of published events to all interested subscribers. In addition to supporting many-to-many communication, the primary application requirement met by pub/sub systems is that producers and consumers of messages are *anonymous* to each other. This allows for components to be dynamically added and removed and for individual components to evolve without disrupting the entire system.

The earliest publish/subscribe systems were *subject-based*. In these systems, each unit of information (which we will call an event) is classified as belonging to one of a fixed set of subjects (also known as groups, channels, or topics). Publishers are required to label each event with a subject; consumers subscribe to all the events within a particular subject. For example a subject-based pub/sub system for stock trading may define a group for each stock issue; publishers may post information to the appropriate group, and subscribers may subscribe to information regarding any issue. The events themselves are treated by the system as uninterpreted bit strings. In the past decade, systems supporting this paradigm have matured significantly resulting in several academic and industrial strength solutions [5][10][13][14][16]. A similar approach has been adopted by the OMG for CORBA event channels [11].

Using subject-based pub/sub systems as a starting point, the Gryphon project has evolved messaging through the following three stages:

1. *Content-based pub/sub*. In order to give users the flexibility to select over not only the subject of an event but also the other pieces of content in the event, content-based pub/sub introduces the notion of information spaces that hold events of a particular type. In contrast to subject-based pub/sub systems, which only allow subscriptions based on the subject field of an event, this approach allows subscriptions to be expressed as predicates over all fields in an event. The Gryphon project has developed fast and scalable algorithms for matching events to content-based subscriptions [1][1][1], and for efficiently routing events from publishers to subscribers in a distributed implementation [3].
2. *Information flow graphs*. To support scenarios where events from multiple publishers are similar but not the same, Gryphon supports transformations on

events. This generalizes to a model where the flow of events in a heterogeneous system is specified as a graph of *selects* (predicates for filtering) and *transforms* (type or format changes). These operations are stateless in the sense that they do not store the histories of events.

3. *Stateful Information flow graphs.* To support subscribers who are interested not only in published events but also in derived events such as summaries, trends, and alarms, the Gryphon model supports several stateful operations as well.

This paper describes the approach of using Information Flow Graphs (or IFGs for short) for application integration. We show that IFGs are not only a flexible and powerful model for expressing event flows, but that they can also be efficiently implemented on a distributed network of event brokers.

The next section provides an overview of the IFG approach and provides initial motivating examples. Section 3 then describes our approach to mapping abstract IFG specifications to physical networks of brokers. Section 4 describes the use of state in IFGs to support sophisticated operations in support of the next generation of messaging applications. Section 5 discusses related work, the current status of this work, and concludes.

2 The Gryphon Approach

Subject-based publish/subscribe systems were limited in the selectivity of subscriptions. An emerging alternative to subject-based systems is content-based subscription systems [6][15]. In these systems, events are organized into *information spaces*. Each information space (I/S) holds a sequence (or stream) of events whose type conforms to an event schema defining the type of information contained in each event. Consider the stock trade example shown in Figure 1. In this example, a stock service is defined as a single information space with an event schema defined as the tuple [issue: string, price: dollar, volume: integer]. A content-based subscription is a predicate against the event schema of an information space, such as [issue="IBM" & price < 120 & volume > 1000]. We will refer to these predicates as *select operations* as in relational algebra.

A key challenge in implementing content-based subscription is that of matching, i.e., for each event, finding the set of subscription predicates that are satisfied by the event. The Gryphon system has developed a fast and scalable matching algorithm for content-based subscriptions [1].

Since subscribers may be distributed, matching needs to be implemented on a network of message brokers. The broker network is responsible for efficiently matching and routing (i.e., multicasting) relevant messages to interested subscribers. The Gryphon project has also developed an efficient protocol for content-based multicast [3].

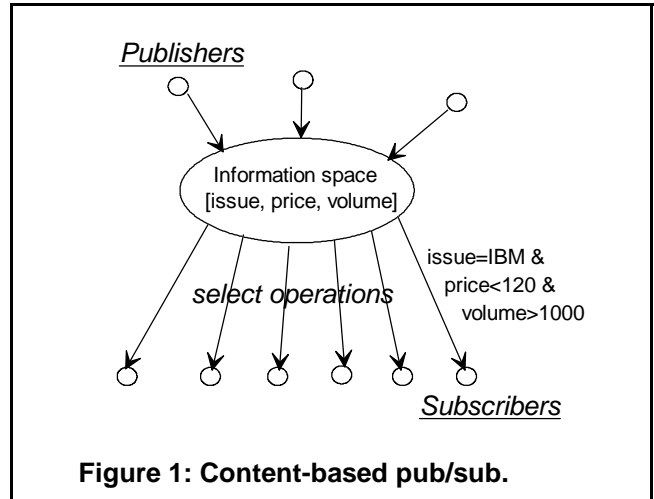


Figure 1: Content-based pub/sub.

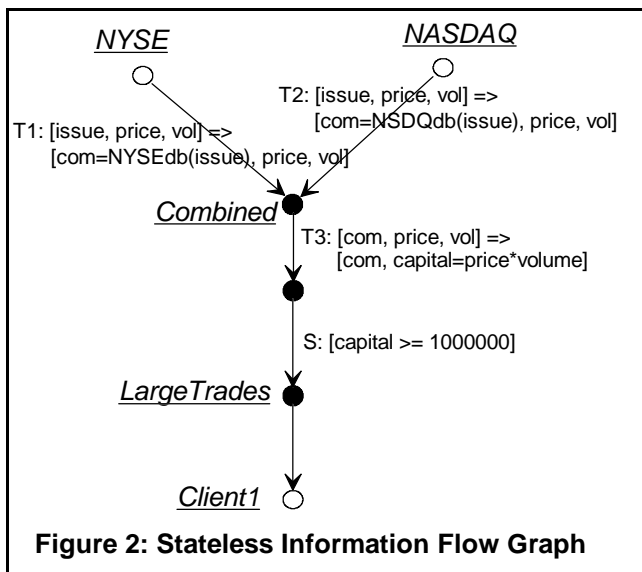
Stateless Information Flow Based Messaging

In the case where it is desired to combine the information from multiple content-based messaging systems, it can often happen that messages published from multiple producers are related, and sometimes only slightly different from what some population of subscribers expect. For example, in the stock market scenario, one publisher may publish a company's name while another may publish the company's ticker symbol. In support of such scenarios, the Gryphon model allows *event transformation* operations which convert events from one format to another by applying a function to the data in the event. .

In the Gryphon approach, select and transform operations are combined into an *information flow graph* abstraction that expresses how events flow from producers to consumers. In these graphs, nodes correspond to information spaces as before, and arcs correspond to *flows*, which specify an operation to perform. Such operations include select and transform on the events passing between information spaces.

Figure 2 shows an information flow graph in which stock trade events from two information sources, NYSE and NASDAQ, are combined, transformed, filtered and delivered to a client. Transforms T1 and T2 convert the "issue" name from the two exchanges to the more easily accessible company name by looking up the respective databases. Transform T3 computes the capital from price and volume, adding it as a field to the events, and the select S filters out events for trades with capital less than \$1,000,000.

Note that the above model is stateless, in that the system does not store any state that is derived from the history of events. The information flow graph abstraction allows users to specify with precision which events they are interested in, while giving the system the freedom to optimize the delivery strategy. In Section 3, we describe the event transformation language in more detail and show how we can implement information flow graphs efficiently,



by reducing them to a form that can exploit efficient content-based routing.

Stateful Information Flow Based Messaging

In many instances, subscribers may not be interested in published events, but rather in events that are derivable from the published events, such as summaries, trends, and alarms. In support of this requirement, Gryphon incorporates an enhancement to the information flow graph abstraction called *event stream interpretation*. Event stream interpretation converts an ordered sequence of messages to a state representing the significant information about the sequence of messages. The state can also be expanded back into a sequence of messages.

Event stream interpretation enhances the power of the Gryphon model in a number of ways:

1. By deriving events based upon past context: for example generating events containing the daily average price of IBM stock.
2. Allowing clients to subscribe to rare conditions without having to receive all events contributing to the condition: for example, one can look for stocks which have dropped more than 20 points in one day.
3. To weaken the requirement of exact delivery by allowing clients to use state to define "equivalent sequences": for example, a subscriber may specify that he cares only about the latest price and daily high price of each stock; the system is free to deliver stock events with missing, out of order, or duplicate events provided that the delivered events yield the correct state.

As with selects and transforms, the Gryphon system is free to optimize message delivery by combining the properties of multiple subscriptions. It may also exploit the freedom to deliver equivalent sequences. Section 4 gives examples of the "stateful" operations provided in the event interpretation language as well as an example of how

equivalent sequences can be used by a Gryphon implementation to intelligently compress event sequences.

3 Stateless Information Flow Graphs

When integrating disparate applications, raw messages provided by one application must often be transformed or reformatted to be appropriate for consumption by another application. Furthermore, it is often desirable to select choice messages from various sources, convert these messages into a common format, and provide them as a value-added information service.

Consider the IFG for stock services shown in Figure 2. This IFG integrates two independent stock markets: NYSE and NASDAQ into the combined information space *Combined*. To do this, the messages from both sources must first undergo a lookup conversion. A *capital* field is then added to each message which is the product of the number of shares in the trade and the price per share. The new value-added information space *Large Trades* is derived from *Combined* by using a select operation, which selects those trades involving over a million dollars.

More formally, the nodes of an IFG is an information space containing events of a particular type. The arcs between nodes specify operations to be performed on events. In this section, we describe four such operations:

1. *Select*. This operation specifies a predicate on the attributes of an event which must be satisfied for the event to flow through the arc. For example:
 $(issue="IBM" \ \& \ price < 120 \ \& \ volume > 1000)$
2. *Transform*. This operation specifies a set of rules with an LHS and an RHS, such that if an event matches the LHS, the operation on the RHS is applied to it. The RHS describes an event computed from the event matching the LHS. For example:
 $[issue, price, volume] \Rightarrow [issue, capital:=price*volume]$
3. *Merge*. This is an implicit operation whenever multiple arcs lead into a node. This means that the events flowing on all the incoming arcs are combined, in a system defined order, at the outgoing arcs of the node.
4. *Copy*. This is an implicit operation whenever multiple arcs lead out of a node. A copy of each incoming event flows out onto each outgoing arc (but subject to the possibly different select and or transform operations which label each arc.)

An unlabelled arc implements the identity (*I*) function which can be considered as the select (*true*) that passes all events through or as the transform that simply copies events.

Note that the above operations are stateless in the sense that they do not store any information derived from the history of events passing through them.

3.1 Optimization of IFGs

IFGs are logical descriptions of the flow of events in a system. Ultimately, this description must be realized on a physical network of message brokers. The problem of mapping an arbitrary logical IFG to a physical broker network is non-trivial. If done naively, the performance of efficient content-based routing systems (such as the one in [3]) cannot be exploited at all.

Our approach to efficient realization of IFGs is to *reduce* an arbitrary IFG to one that can be efficiently implemented on a content-based routing system. The basic idea is to re-write the IFG so that all the select operations are lumped together and moved closer to publishers, and all the transform operations are lumped together and moved closer to the subscribers. (Because transforms may destroy information, they cannot, in general be pushed ahead of selects.) This will allow us to use the content-based routing protocols described in [3] to implement the select operations within the broker network, then perform the transform operation at the periphery of the broker network. Furthermore, we may be able to optimize away transform operations on events that would be eliminated by later select operations.

Rewriting the IFG can be done by an automated system so that, while users specify information flows as a series of selects and transforms that closely matches the way they think about the flow and processing of events, the system can optimize the processing of information flows.

The rules for rewriting graphs are described below.

Selects can be pushed ahead of transforms.

For any dataflow in which a transform TA is followed by select SA, there is an equivalent dataflow of the form SB followed by TB. To see how, observe that the predicate of SA must be a function of constants and the function outputs of TA. We can construct a predicate for SB that will choose the same messages as SA, by simply substituting the appropriate functions of TA for the attributes in the predicate of SA.

For example (in all of the examples in this section, the semicolon is used to mean “followed by” in an information flow):

TA: $[x1, x2] \Rightarrow [y1=f1(x1,x2,c1), y2=f2(x1,x2,c2)];$

SA: $(p(y1,y2,d))$

can be re-written as:

SB: $(p(f1(x1,x2,c1), f2(x1,x2,c2),d));$

TB: $[x1,x2] \Rightarrow [y1=f1(x1,x2,c1), y2=f2(x1,x2,c2)]$

where $x1,x2,y1,y2$ are attribute names, and $c1,c2,d$ are constants.

Selects can be combined.

Observe that a sequence of selects is just a conjunction of predicates. Thus, selects SA: $(p(...))$ followed by SB: $(q(...))$ can be rewritten as SC: $(p(...) \& q(...))$.

Transforms can be combined.

Observe that we can perform variable substitutions from a first transform into a second. For example:

$[x,y] \Rightarrow [y1:=f1(x1,x2,c1), y2:=f2(x1,x2,c2)];$

$[y1,y2] \Rightarrow [z1:=g1(y1,y2,d1), z2:=g2(y1,y2,d2)]$

can be rewritten as

$[x,y] \Rightarrow [z1:=g1(f1(x1,x2,c1),f2(x1,x2,c2),d1),$

$z2:=g2(f1(x1,x2,c1),f2(x1,x2,c2),d2)]$

The Select-Transform equivalence rule.

By applying the above rewriting rules, any sequence of selects and transforms can be reduced to a single select followed by a single transform. For example, starting with the sequence [T T S T S T], we push selects ahead of transforms to get [S S T T T], and combine selects and transforms to get [S T].

Once a dataflow has been reduced so that all paths from publishers to subscribers may be represented by a [Select;Transform] pair, the single select can be further optimized. A straightforward application of the re-writing rules may cause many common subexpressions to appear within the combined predicate. A smart implementation can discover those, just like any good compiler, and avoid re-computation of sub-functions. Likewise, the final, single transform will likely contain many common subexpressions and many of those will have already been computed for the select. A smart implementation can cache them and/or tag each selected message with them as auxiliary attributes.

Externalizing I/S nodes as terminal nodes.

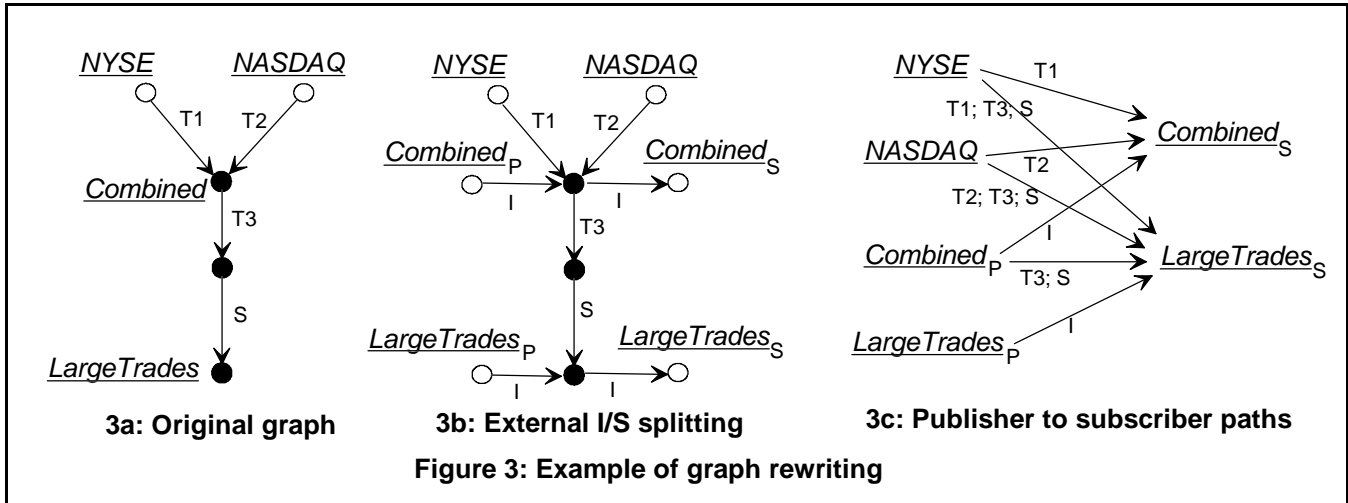
As we combine and rearrange the select and transform operations specified by an IFG, we may eliminate or change the meaning of the non-terminal I/S nodes. However, the users who specify IFGs may wish to use a non-terminal or internal I/S node as a publication and/or subscription point. (e.g., the node labelled “Combined” in Figure 2.) To avoid losing such I/S nodes due to rewriting an IFG:

1. For each internal node that may be used as a publication point we add an explicit terminal node with an identity arc that connects to the internal node.
2. For each internal node that may be used as a subscription point we add an explicit terminal node with an identity arc from the internal node to the terminal node.

Thus, all publication and subscription points are represented by the terminal nodes of the IFG. All arcs and internal nodes can be subjected to rewriting rules and optimizations.

Rewriting the entire IFG.

Consider an IFG, G1, with all interesting publication and subscription points externalized. We can construct an equivalent IFG, G2, as follows. For each publication and subscription point in G1, add a like-named publication or subscription point to G2. For each possible pair (p,s) of publication and subscription points in G1, if there is a path



from p to s in $G1$ consisting of arcs labelled with transforms and/or select operations:

$$p \rightarrow (ts1; ts2; \dots; ts_k) \rightarrow s$$

then add a single arc from p to s in $G2$ that is labelled with the (select;transform) pair of operations that is equivalent to $(ts1; ts2; \dots; ts_k)$, as given by the above rewrite rules.

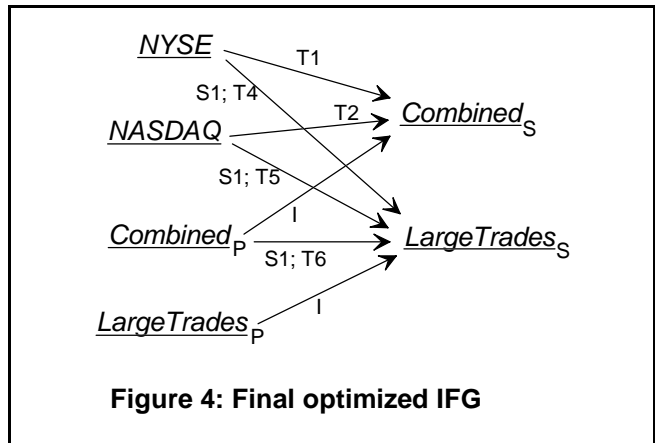
3.2 Example of Graph Rewriting

In Figure 3, we demonstrate how the rewrite rules above may be used to convert a stateless information flow into one where selects are combined and pushed towards the publisher and transforms are moved towards the subscribers. In this example, we begin with the graph of Figure 2, simplified as shown in Figure 3a, and perform the following operations on it.

As shown in Figure 3b, we split each externally visible I/S into two: one for publishers and another for subscribers, e.g., *Combined* is split into *Combined_p* and *Combined_s*. This step is necessary to ensure that, after transform, all the advertised content is still available to dynamically joining publishers and subscribers.

Next, we identify all paths in the graph of Figure 3b to arrive at Figure 3c. For each path that has more than one select or transform, we then apply the rewrite rules so that 1) selects are moved before transforms and 2) a series of selects or a series of transforms are combined into a single instance of each. In this example, we simplify three such paths:

1. NYSE to *LargeTrades_s*: T1; T3; S, which reduces to:
 $S1: (price \cdot vol \geq 1000000);$
 $T4: [issue, price, vol] \Rightarrow$
 $[com=NYSEdb(issue), cap=price \cdot vol]$
2. NASDAQ to *LargeTrades_s*: T2; T3; S, which reduces to:
 $S1: (price \cdot vol \geq 1000000);$
 $T5: [issue, price, vol] \Rightarrow$
 $[com=NASDdb(issue), cap=price \cdot vol]$
3. *Combined_p* to *LargeTrades_s*: T3; S. This reduces to:
 $S1: (price \cdot vol \geq 1000000);$



$$T6: [com, price, vol] \Rightarrow [com, cap=price \cdot vol]$$

With this, each path from a publisher to a subscriber is of the form select followed by transform, as shown in Figure 4.

The selects can now be implemented by an efficient content-based routing system, and the transforms performed before delivering to subscribers. Going one step further, we can combine the individually derived paths back into a single I/S, which may be implemented as a content-based publish/subscribe system. These paths can then be split by adding an additional select based on message source, then tagging the transforms with a source. Before an event is delivered to a subscriber, it is transformed based on the I/S to which the client subscribed and the source of the message. Tagged transforms can be stored in a table for lookup and execution before the system delivers a message to a client. New subscriptions coming into any of the subscription points (*Combined* or *LargeTrades*) have their content filters modified based on the filter arcs out of the root into these spaces using a simple application of the rewrite rules.

4 Stateful Information Flows

Event stream interpretation captures the fact that not only do individual events have meaning, but also streams of

events have meanings. Additionally, an event’s meaning may depend upon the context of some or all of the events that preceded it. As with content-based subscription, event stream interpretation can increase the flexibility of end-user subscription, and can be exploited by the system to improve performance. In particular, subscribers can generate context-dependent derived events, such as averages, trends, or alarms. Subscribers can also give the system the freedom to substitute an equivalent sequence mapping to the same state in place of the original sequence.

4.1 Enhancements to IFGs

To support event stream interpretation, we introduce a new kind of node in the information flow graph: a *state*. Like an information space, a state has a schema. Unlike an information space, a state is not constrained to grow monotonically as the system evolves. We also introduce two new kinds of flow operations: a *collapse* operation, and an *expand* operation.

1. *Collapse*. This operation converts an event stream to a state. It is parametrized by a *summarization function* whose signature is $list\ of\ E \rightarrow S$, where E is the base event type of the event stream, and S is the type of the state. In some contexts, it is useful to express the summarization function in *incremental form*, that is, by defining how to convert an existing state and a new event into a new state. When defining the summarization function in incremental form, one specifies an initial state, and an incremental operation with signature $E \times S \rightarrow S$. The result of the collapse operation is simply the state resulting from applying the summarization function to the event sequence.
2. *Expand*. This operation inverts the *collapse* operation --- that is, converts a state into an event stream. It is also parametrized by a summarization function. The result of the *expand* operation is one of the possible inverses of the summarization function --- that is, an event sequence which will yield the given state after applying the summarization function. Notice that the result of *expand* is non-deterministic and may even be undefined. We only deal here with cases where there is guaranteed to be at least one possible defined outcome of *expand*. The non-determinism of *expand* gives the system the freedom to deliver different equivalent event sequences.

4.2 Examples

Bank Account: Derived Events

The first example will illustrate derived events. Let us assume that in a bank account, each event is a tuple $\langle D, amt, id \rangle$ for desposits, or $\langle W, amt, id \rangle$ for withdrawals. To simplify the example, we will assume that the system is dealing with a single account. To interpret these events, we

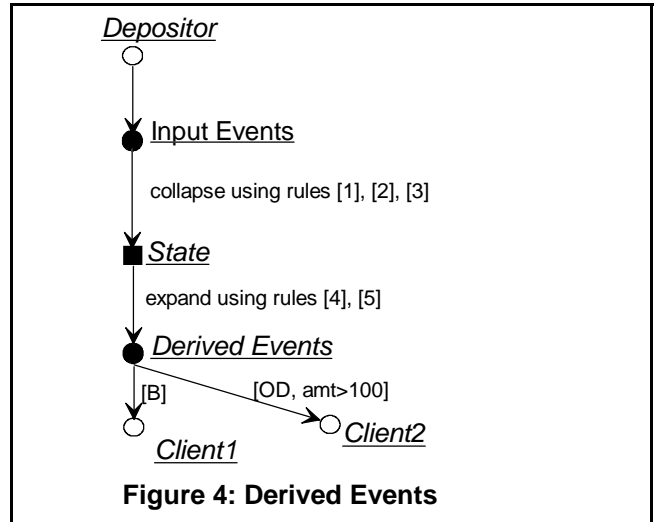


Figure 4: Derived Events

assume that the application only cares about the net balance and about attempted overdrafts. Our state, therefore, is a tuple $\langle bal: integer, ov: list\ of\ \langle amt, id \rangle \rangle$. From an initial state of $\langle bal: 0, ov: () \rangle$, we specify the summarization function incrementally using a set of *replacement patterns*, in which the first pattern whose left side matches the event and current state triggers the replacement of the state with the right side:

- [1] $\langle D, amt, id \rangle, \langle bal, ov \rangle \rightarrow \langle bal+amt, ov \rangle$
- [2] $\langle W, amt, id \rangle, \langle bal \geq amt, ov \rangle \rightarrow \langle bal-amt, ov \rangle$
- [3] $\langle W, amt, id \rangle, \langle bal < amt, ov \rangle \rightarrow \langle bal, ov + \langle amt, id \rangle \rangle$

The first rule is matched for any deposit event and simply increments the balance by the deposited amount. A withdrawal event will match rule [2] if there is a sufficient balance, and will decrement the balance. If there is not a sufficient balance, rule [3] will fire, leaving the balance unchanged, and appending the unsuccessful transaction to the list of attempted overdrafts.

Having collapsed a set of bank account deposit and withdrawal events to a state, we generate our derived events by re-expanding the state into an event stream using a summarization function over a different set of events, namely events of the form $\langle B, balance \rangle$ and $\langle O, withdrawal, id \rangle$, and a similar initial state:

- [4] $\langle B, bal \rangle, \langle bal, ov \rangle \rightarrow \langle bal, ov \rangle$
- [5] $\langle OD, amt, id \rangle, \langle bal, ov \rangle \rightarrow \langle bal, ov + \langle amt, id \rangle \rangle$

Rule [4] simply replaces the balance; rule [5] appends to the list of overdrafts. The following table shows one possible result of publishing a particular event sequence:

Input Event	State	Derived Event
[D, 10, 0001]	<10, ()>	[B, 10]
[D, 20, 0004]	<30, ()>	[B, 30]
[W, 25, 0008]	<5, ()>	[B, 5]
[W, 10, 0030]	<5, ([10, 0030])>	[OD,10,0030]

What is shown above is the most “natural” sequence of derived event; however the non-determinism of the *expand*

operation permits the system to deliver other sequences. For example, all the balance updates except the last can be omitted or permuted without changing the final state. So if the system load is very high, the delivery system is free to omit event [B, 30] without affecting the final state. (Notice however that if the event interpretation had tracked, for instance, *maximum* balance, then the delivery system would no longer have been free to omit this event.) As we shall see below, it can be useful to exploit the ability to deliver equivalent sequences, even if one is not using the state to derive new events.

Stock Prices: Equivalent Sequences

Suppose the subscriber collapses an event stream using a summary function, and then re-expands to the original type of event stream using the identical summary function. This is an important special case of event interpretation called *equivalent sequence generation*. Notice that it is always correct to treat the collapse and expand as an identity function. However, the non-determinism of *expand* gives additional flexibility to the delivery system, permitting it to suitably alter the delivered event stream, but only in ways which do not affect the specified interpretation. For example:

1. As discussed above, under conditions of high load, the system may be able to omit events or to combine certain events.
2. If a subscriber is offline, and goes back online after a large number of events have been delivered, it may not always be necessary to deliver all these events upon reconnection: the system may be able to perform *intelligent compression* and deliver a possibly much shorter sequence which collapses to the same state. An example of this is shown below.
3. The delivery system may be able to use *optimistic* delivery protocols that exploit the ability to deliver permuted or distorted equivalent sequences.

Here is an example illustrating how clients can exploit the ability to specify equivalent sequences. The events are stock trades of the form $\langle \text{issue}, \text{price} \rangle$, and the state is of the form *bag of* $\langle \text{issue}, \text{max}, \text{cur} \rangle$ (keyed by issue). To simplify the example (and avoid dynamically growing structures), we assume an initial state of $\langle \text{issue}, 0, 0 \rangle$ for each issue.

In this example, we use the same notation for expressing the summary function in incremental form:

$$[6] \quad \langle i, p \rangle, \langle i, \text{max} < p, \text{cur} \rangle + s \rightarrow \langle i, p, p \rangle + s$$

$$[7] \quad \langle i, p \rangle, \langle i, \text{max} \geq p, \text{cur} \rangle + s \rightarrow \langle i, \text{max}, p \rangle + s$$

Rule [6] applies when the new price is strictly greater than the current maximum price; both the maximum price and the current price are updated. In all other cases, rule [7] applies, and only the current price is updated.

Suppose a mobile client subscribes to the IBM events from the information space of equivalent events derived by the above rules. After receiving a number of events, and arriving at a state in which IBM's maximum price is 160

and its current price is 140, the client disconnects. While the client is disconnected, a long series of events is published, arriving at a new state with maximum price of 200 and current price of 120. The mobile client then reconnects to the system. If the system is able to exploit the knowledge of the client's interpretation of event sequences, it should be able to deliver just the two events [IBM, 200] and [IBM, 120] rather than the much longer sequence of published events. The following table shows the published events, the generated state (event stream interpretation), and the compressed set of delivered events.

	Published Events	States $\langle \text{issue}, \text{max}, \text{cur} \rangle$	Delivered Events
	[IBM, 150]	$\langle \text{IBM}, 150, 150 \rangle$	[IBM, 150]
	[IBM, 160]	$\langle \text{IBM}, 160, 160 \rangle$	[IBM, 160]
	[IBM, 140]	$\langle \text{IBM}, 160, 140 \rangle$	[IBM, 140]
disconnect:			
	[IBM, 200]	$\langle \text{IBM}, 200, 200 \rangle$	
	[IBM, 180]	$\langle \text{IBM}, 200, 180 \rangle$	
	
	[IBM, 120]	$\langle \text{IBM}, 200, 120 \rangle$	
reconnect:			
			[IBM, 200]
			[IBM, 120]

4.3 Implementing Expansion

Given a current state, a goal state, and a summarization function, the *expansion problem* is defined as the generation of the most economical sequence of events which in the context of the current state, yields the goal state. The expansion problem can be converted into a *shortest path graph search problem*. We represent the states in S as vertices in a graph, and define each possible event transition as an edge. We then label these edges with a cost. For the purpose of this paper, we will assume each event has unit cost 1. Figure 5 shows a fragment of the state transition diagram for the stock prices example whose rules are shown above (for just IBM events).

To solve the shortest path problem, we use the known A* algorithm [9][4]. This algorithm requires an estimator function h , where $h(s)$ is a lower estimate of the shortest path from state s to the start state s_0 . Working backward from the goal state g , toward s_0 , we keep a set of candidate paths. We sort these paths based upon the actual length from g to the end s of the path plus the estimated length $h(s)$ from s to s_0 . Beginning with the node n at the end of the best candidate path, of length $f(n)$, we locate that neighbor n' of n that minimizes $(f(n) + 1) + h(n')$. We extend the candidate path in the direction to n' . (We ignore other neighbors unless and until all candidates of at least this distance have been explored.)

The problem is to find suitable estimator functions $h(s)$. One can obtain an estimator $h(s)$ for a particular graph by

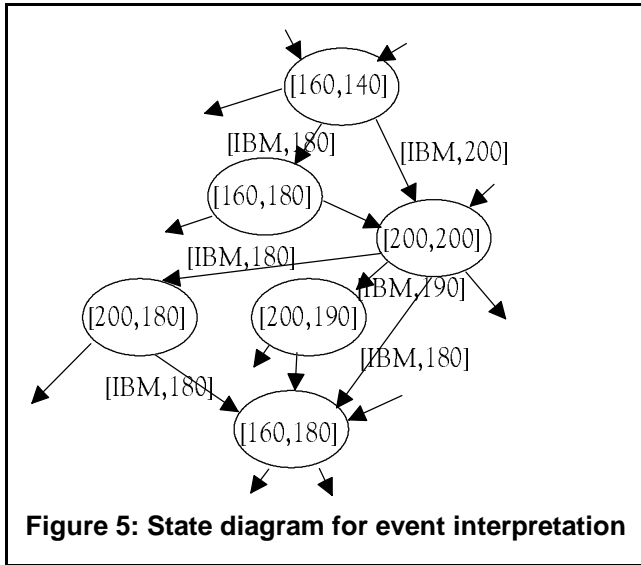


Figure 5: State diagram for event interpretation

constructing an exact solution for an extended graph with strictly more edges. We have developed a strategy for finding h for an important subclass of summarization functions by solving a hierarchy of problems.

Let us assume that the incremental formulation of the summarization function can be put in the following form:

	s1	s2	s3
e1(a,b,c)	a		b
e2(a,b,c)	a	a	
e3(a,b,c)		b	c

Each row of the table corresponds to a particular collection of events meeting a particular condition, and having parameters, e.g. a, b, and c. Each column of the table corresponds to a component of the state. Each blank entry in the table indicates that the event in the corresponding row leaves the corresponding component of the state unchanged; each non-blank entry indicates that the event in the corresponding row replaces the corresponding component of the state. Not every summarization function can be put in this form; however many can, such as the stock example illustrated above. (We call this the *replacement form*.)

If a summarization function is in replacement form, and it contains no rows such as the second in the example above, in which two or more columns are constrained to be replaced by the same value, then it is in *unconstrained* form. If a summarization function is in unconstrained form, and each row changes k columns, and for any k columns there is a row which changes those columns, then it is in *uniform unconstrained* form. The exact solution to a problem in uniform unconstrained form requires a number of events equal to $\text{ceil}(m/k)$, where m is the number of state components in which the start and goal states differ. Any problem in unconstrained form but not uniform unconstrained form can be solved by extending it to uniform unconstrained form, and using the exact solution

to the extended problem as an estimator for the original problem, and then applying A^* . Similarly, any problem in constrained form can be extended to unconstrained form by assuming that all steps except the first (for which the constraints are known) may follow new rules in which additional parameters have been added as necessary to eliminate constraints. The optimal solution to the unconstrained problem serves as an estimator for the constrained problem.

For example, the stock price example can be put into the form of a constrained problem as follows:

	Maxprice	Curprice
$p > \text{Maxprice}$	p	p
$p \leq \text{Maxprice}$		p

This problem can be solved by using the corresponding unconstrained problem as an estimator. In this case, the estimation function is straightforward: the estimated distance from start to goal equals the number of issues for which the current state differs in cur price or max price from the goal state.

In this example, finding a path from [IBM, 160, 140], to state [IBM 200, 120] the search turns out to be so easy that the estimators would not really be needed. Starting at goal state [200, 120], we find that only the second row of the matrix leads to a predecessor state, which has the form [200, *] For any value of the second state except 200, the first row is blocked and therefore the estimated distance from the start state is 2. For the state [200, 200] the estimated distance is 1. We therefore choose state [200, 200] as the best candidate to continue the search. In fact, this state satisfies the conditions for firing row one of the matrix to reach a predecessor [*, *], so we can insert the start state [160, 140].

5 Discussion

5.1 Related Work

Many concepts in Gryphon have been synthesized from a large base of computer science work in group communication, databases, programming languages, and software engineering. As mentioned in Section 1, existing publish/subscribe technologies were our starting points.

One of the bases for the work described in this paper is an efficient and scalable protocol for content-based routing within a network of brokers [3]. The content-based subscription systems that have been developed to date do not yet address wide-area, scaleable event distribution, i.e. although they are *content-based subscription* systems, they do not support efficient *content-based routing*. SIENA [6] allows content-based subscriptions to a distributed network of event servers (brokers). SIENA filters events before forwarding them on to servers or clients. However, a scaleable matching algorithm for use at each server has not

been developed. The Elvin system [15] uses an approach similar to that used in SIENA. Publishers are informed of subscriptions so that they may “quench” events (not generate events) for which there are no subscribers. In [15], plans are discussed for optimizing how Elvin performs event matching by integrating an algorithm similar to the parallel search tree. This algorithm, presented in [8], converts subscriptions into a deterministic finite automata for matching. However, no plans for optimizations such as those discussed in [3] are discussed.

The basic idea of representing system behavior in terms of the flow of data from inputs through functional modules to output, is used extensively in software design, see for example [7]. It is also common practice to allow software designers to depict the structure of a system using a high-level visual representation similar to data flow diagrams. The tool then converts the high-level representation into lower-level executable code.

Ideas for graph rewriting and analysis of data flow graphs have their origins in very early work in programming languages and code optimization in compilers [2].

Some systems commercially available today (e.g., NEON, [12]) claim to support arbitrary filtering and transforming operations. However, there is scant literature on the exact technical nature of these operations. It is not clear that these systems support a systematic approach to specifying the flow of events. Furthermore, there is no evidence that these systems support efficient routing of events by optimizing IFGs and mapping them to distributed broker networks.

5.2 Current Status and Future Work

We have developed a Gyphon system prototype that supports content-based publish/subscribe via efficient matching and multi-broker networks. We have also developed initial prototypes of tools for supporting stateless and stateful information flows. One tool supports the visual specification of information flow graphs and applies the rules of Section 3 to rewrite the graph. Another tool supports a restricted language to specify the meaning of event sequences, using which the tool generates equivalent, but shorter event sequences.

Several directions of work are ongoing and appear promising. Besides extending our graph rewriting techniques to encompass a more expressive language, we are working on ways to efficiently map optimized IFGs onto physical broker networks of various configurations. We are also incorporating protocols for reliable and ordered delivery within this framework of messaging middleware.

We believe that stateful operations within IFGs are the next major step in the functionality of messaging middleware. We are currently exploring the breadth of applicability of derived events and equivalent event sequences. Finally, we are beginning to deploy this new

generation of messaging middleware in real-world application integration scenarios.

6 Bibliography

- [1] Marcos Aguilera, Rob Strom, Daniel Sturman, Mark Astley, Tushar Chandra. 1998. Matching Events in a Content-Based Subscription System. Upcoming IBM Technical Report, available from <http://www.research.ibm.com/gryphon>.
- [2] Aho, A., Sethi, R., and Ullman, J. 1985. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley publishing, Reading, MA.
- [3] Banavar, G., Chandra, T., Nagarajarao, J., Mukherjee, B., Strom, R., Sturman, D. 1998. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. Upcoming IBM Technical Report, available from <http://www.research.ibm.com/gryphon>.
- [4] Barr, A., and Feigenbaum, Edward A. 1986. *The Handbook of Artificial Intelligence*. Volume 1. Addison-Wesley Publishing, Reading, MA.
- [5] K. P. Birman. “The process group approach to reliable distributed computing,” pages 36-53, Communications of the ACM, Vol. 36, No. 12, Dec. 1993.
- [6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. “Design of a Scalable Event Notification Service: Interface and Architecture,” unpublished. Available from <http://www.cs.colorado.edu/users/carzaniga/siena/index.html>
- [7] Ghezzi, C., Jazayeri, M., and Mandrioli, D. 1991. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ.
- [8] John Gough and Glenn Smith. “Efficient Recognition of Events in a Distributed System,” Proceedings of ACSC-18, Adelaide, Australia, 1995.
- [9] Hart, P.E., Nilsson, N. J., and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on SSC*. SSC 4:100-107.
- [10] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs, Dept. of computer science, The University of Arizona, TR 91-32, Nov. 1991.
- [11] Object Management Group. CORBA services: Common Object Service Specification. Technical report, Object Management Group, July 1998.
- [12] New Era of Networks (NEON). <http://www.neonsoft.com>.
- [13] Brian Oki, Manfred Pfluegl, Alex Siegel, Dale Skeen. “The Information Bus - An Architecture for Extensible Distributed Systems,” pages 58-68, Operating Systems Review, Vol. 27, No. 5, Dec. 1993.
- [14] David Powell (Guest editor). “Group Communication”, pages 50-97, Communications of the ACM, Vol. 39, No. 4, April 1996.
- [15] Bill Segall and David Arnold. “Elvin has left the building: A publish/subscribe notification service with quenching,” Proceedings of AUUG97, Brisbane, Australia, September, 1997.
- [16] Dale Skeen. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview, <http://www.vitria.com/>