Subscription Propagation in Highly-Available Publish/Subscribe Middleware

Yuanyuan Zhao, Daniel Sturman, and Sumeer Bhola

{yuanyuan, sturman, sbhola}@us.ibm.com
IBM T.J.Watson Research Center,
Hawthorne, NY 10532, USA

Abstract. Achieving availability and scalability while providing service guarantees such as in-order, gapless delivery is essential for deploying publish/subscribe messaging middleware in wide area networks. Scalability often requires a publish/subscribe system to propagate subscription information and perform content matching across the network. Existing subscription propagation algorithms do not support in-order, gapless delivery in a redundant overlay network.
This paper presents a novel approach that utilizes virtual time (VT) vectors to convey temporal consistency in propagating incremental and consolidated subscription information. The VT vectors provide a means of testing sufficiency of filtering information, by comparing a broker's VT vector with that of a message. When the test fails, indicating insufficient broker subscription information, safety may be preserved by 'flooding' the message to all neighbors on a routing tree. This approach does not require subscription state agreement across redundant paths and hence is highly available. We present a detailed evaluation of the approach.

Keywords: publish/subscribe middleware, subscription propagation, high availability

1 Introduction

Content-based publish/subscribe(pub/sub) messaging is a popular paradigm for building asynchronous distributed applications. A content-based pub/sub system consists of *publishers* that generate messages and *subscribers* that register interest in messages. The interest is usually expressed through content filters in the form of Boolean expressions. The system ensures timely delivery of published messages to all interested subscribers, and typically contains *routing brokers* for this purpose. Publishers and subscribers obtain service by connecting to brokers and are decoupled with each other.

In many cases, publishers and subscribers also want strong service guarantees, such as in-order, gapless delivery [3, 4] (referred to as reliable delivery in the rest of the paper). These services are usually provided by the broker network and made available to clients through standard messaging interfaces, like the

Java Message Service (JMS) [1]. Reliable delivery guarantees that for every subscription s and a published message stream, the system finds a starting message m_0 and from m_0, delivers all and only those messages matching s in an order conforming to the original stream.

In addition to providing reliable delivery guarantees, commercial deployment over wide-area networks requires a pub/sub system to be scalable, highly available and utilize network bandwidth efficiently. In order to efficiently utilize network bandwidth, a pub/sub system usually propagates information about subscriptions across the broker network. A broker stores, for each of its neighboring part of the network, the information on what messages are needed by subscribers from the network. When a new message is published and routed, a broker filters out and does not send the message to parts of the network where no subscriber is interested. However, the amount of subscription information could get very large as it approaches the publishers. A scalable pub/sub systems should aggregate and only maintain a subset of subscription information for each routing direction of a broker as long as the subset of information is sufficient to match all messages that are needed by subscribers from that routing direction.

The combination of content-based routing and reliable delivery provides some unique challenges. Unlike in topic or group-based pub/sub systems, reliability cannot be based only on detecting gaps in publisher-assigned sequence numbers as each content subscriber may request a completely unique set of messages to be delivered. Reliable delivery protocols typically rely on brokers on the routing path to assist on detecting gaps. A routing broker with incorrect subscription information may decide not to forward on a message. Given that gaps cannot be detected by checking publisher-assigned sequence numbers, an end subscriber may never discover that a message was missed.

Numerous studies have shown that loss of connectivity is common in wide-area networks, due to hardware and software failures and network misconfigurations. Hence, pub/sub systems should be built on networks with redundant links. This further complicates subscription propagation as alternative routes with different subscription information may filter out messages matching subscriptions that are unknown to a route.

In this paper, we present a subscription propagation algorithm that supports reliable delivery in redundant overlay networks. Using this algorithm, a pub/sub system is able to (1) propagate and aggregate subscriptions; (2) support reliable delivery; (3) choose freely (based on other criteria such as traffic load, system faults) among multiple redundant links for data message routing; and (4) not require heavy consensus protocols among brokers that serve as alternatives or backups of each other. Our approach deals with link failure, message losses and re-ordering and broker crash failures. Byzantine failures are outside the scope of this work.

We have implemented our algorithm in the Gryphon system. Gryphon is a content-based pub/sub system designed for high-volume, low latency, Internet-scale distribution. Gryphon can be deployed using multiple geographically-distributed application-level routers (called brokers), with tens of thousands of clients and

with tens of thousands of messages being delivered across the system each second. Our algorithm, however, applies to a wide range of messaging systems.

Previous pub/sub systems have mainly focused on efficiently utilizing network bandwidth and on scalability issues concerning propagating and aggregating subscriptions. However, few of these systems efficiently exploit the network redundancy and recover from failures in a timely fashion. Indeed, existing pub/sub systems are usually unable to utilize the redundant links or do not support reliable delivery. Simply extending existing work that utilizes only a single path requires consensus between multiple brokers on alternative paths and will incur a lot of overhead.

The remainder of this paper is organized as follows: Section 2 presents a redundant overlay network model. Section 3 describes the basic algorithm, and then discusses a number of variants. Section 4 discusses liveness and failure recovery, and Section 5 describes our implementation and experimental results. Section 6 is an overview of related work and we conclude in Section 7.

2 Context: Redundant Overlay Network

In this section, we describe a simplified topology model of a redundant overlay network of messaging brokers. We assume a redundant routing tree where each tree node may contain more than one broker/server(Figure 1). We further restrict the edge nodes (e.g. N_{11}, N_{12}, N_{13}, and N_{31}) to contain only one broker. We assume there are bi-directional links between any two brokers in neighboring nodes. Furthermore, we assume brokers residing in the same tree nodes are fully connected when there is no failure. The algorithm, however, does not depend on the full connectivity.

As we describe the protocol from the standpoint of a single publisher connecting broker, we arbitrarily designate an edge node as the root of the tree and other edge nodes as the leaves. Publishers connect at the root broker (also called *publisher connecting broker*) and subscribers at leaf brokers (*subscriber connecting brokers*). The full algorithm with publishers/subscribers connecting at any edge node is a repetition of this simplified description with repetitive information avoided. As a convenience, we refer to where the root node resides as upstream, and leaf nodes as downstream.

The singleton broker in an edge node can be replicated but requires strong consistency to ensure client connection and failover from one replica to another does not cause message loss or duplication.

Figure 1 shows a redundant routing tree that contains: 2 non-edge nodes N_{21} and N_{22} each with two brokers; 1 root node N_{31} with broker PB_1 for publisher connections; and 3 leaf nodes N_{11}, N_{12} and N_{13} each with one broker (SB_1, SB_2 and SB_3, respectively) for subscriber connections.

This model may seem a bit restrictive at first glance. However, the brokers referred here are logical brokers. A broker process may represent multiple logical brokers and hence participate in more than one node and have virtual links between its multiple presence in neighboring tree nodes. Furthermore, publishers

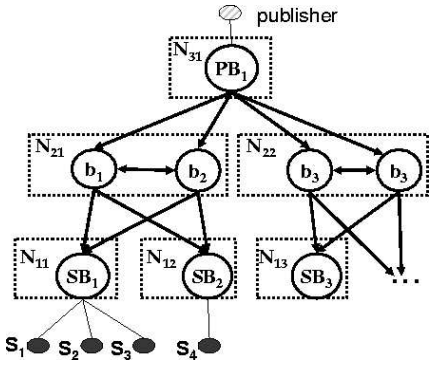


Fig. 1. A Redundant Routing Tree

and subscribers can connect to any physical brokers. The physical broker in this case implements a logical leaf broker. Thus we abstract the topology to have publishers and subscribers connecting only at the edge nodes. How to map physical brokers to logical brokers, and what brokers reside in which tree nodes is an important part of the Gryphon technology. We do not describe it here.

3 Algorithms

3.1 Solution Intuition

To properly introduce the intuition behind our algorithm, we revisit in detail the challenges facing subscription propagation. We use the example network shown in Figure 1 to illustrate.

One challenge is to decide the delivery starting point of a subscription, that is, to find a position in a published message stream from which the system can deliver messages in order and without gaps. This problem is complicated because the new subscriptions may partially overlap with existing subscriptions. Suppose in the network in Figure 1, subscriber/subscription s_1 of $stock = nyse : ibm$ has been propagated to all brokers in the tree nodes on the path from N_{11} to N_{31}, that is, broker SB_1, b_1, b_2 and PB_1. When subscriber s_2 submits subscription $stock = nyse : *$ (which matches all messages matching s_1 and more), broker SB_1 propagates this information toward PB_1. However, when message m_1 with content $(nyse : ibm, 92)$ arrives, broker SB_1 does not know whether it should deliver it to subscription s_2 as well. It might be an error to deliver m_1 because m_1 arrives only as a result of matching subscription s_1. If this is the case, there is no guarantee that a later message m_2 of $(nyse : t, 19)$ will be routed by the system and thus result in a gap in the messages delivered to subscription s_2.

Another challenge arises when messages that are part of the same ordered stream travel along different paths. This might be a result of a load sharing

decision or a failed link. Furthermore, the message delivery path could be different from the path subscription has been propagated. For example, subscription s_2 might have been propagated from SB_1 to b_1 and PB_1 but not yet to b_2. If message m_1 is routed through broker b_1 and m_2 through b_2, b_2 would not send m_2 to SB_1 without knowing subscription s_2. Delivery for s_2 might have already started and SB_1 would not detect the gap created by missing m_2. This kind of gap can not be detected by the reliable delivery protocol because it relies on *correct* subscription information.

Our algorithm solves these problems by making two types of information explicit, i.e., the set S_m of subscriptions a message should be matched against and the set S_b of subscriptions a broker has information of and uses for message matching. Using this information, delivery starting point of a subscription can be detected by choosing the first message whose S_m set contains the subscription. In addition, the system guarantee that for every message published in the same stream after the first message, its S_m set (eventually) includes the subscription until it is unsubscribed. How and when the S_m set of a message is assigned and changed is a dimension in which an algorithm can vary.

To prevent a broker (e.g. b_2) to filter out a message by mistake, the broker compares its S_b set with the S_m set of the message. The broker contains information of the subtree where a subscription in S_b set is connected and thus can project/partition the S_b set onto each subtree rooted at the tree node where the broker resides. The broker can also project the S_m set onto each subtree but only partially because the S_m set may contain subscriptions that is not in the broker's S_b set. The comparison of S_m and S_b is thus performed for each subtree rooted at a child node (e.g. N_{11}). It is possible that in subtree N_{11} there is a subscription that is contained in S_m but not in S_b, and the subscription matches the message. However, since the broker does not have information about the subscription, its filtering result will probaby indicate that the message does not need to be sent to subtree N_{11}. Our algorithm takes the comparison result of S_m and S_b as an indicator that whether the broker matching/filtering result should be used. If S_b is the same or a superset of S_m for a subtree, the broker contains *sufficient* information to perform matching and can decide whether or not to send the message to a subtree based on the existence of matching subscriptions. Otherwise, as a conservative measure, the message should be sent to the subtree regardless of the matching result. We call this comparison of S_b and S_m sets the *sufficiency test*.

With this type of algorithm, there is no need to maintain consistent states of subscription information between redundant path brokers. As a result, the solution is light-weight and highly available.

3.2 Solution Overview

Obviously, concise representation of S_m and S_b sets and efficient computation of the sufficiency tests are crucial for system performance. In our approach, each leaf broker maintains a virtual time clock. We assign each subscription a virtual start time (vst) at its subscribing time and a virtual end time (vet) at its

unsubscribing time using the current value (VT) of the virtual clock at the leaf broker to which the subscription connects. We call a subscription that has not unsubscribed *active*. Such a subscription has a *vet* that is greater than the VT of the leaf broker to which the subscriber is connected. The virtual clock advances whenever necessary and more than one subscription/unsubscription can occur in a single virtual time tick.

Thus, we can represent the set S_m of a message as a vector V_m, with one element per leaf broker. Assume a V_m vector $(SB_1 : v_1, \cdots, SB_n : v_n)$, the set S_m of the message is "any active subscription at broker SB_1 with *vst* no later than v_1 and *vet* later than the current virtual time VT_1 , $\cdots$, plus any active subscription at broker SB_n with *vst* no later than v_n and *vet* later than the current virtual time VT_n".

Similarly, we represent the set S_b of a broker as a vector V_b, with one element per leaf broker. Assume a V_b vector $(SB_1 : v'_1, \cdots, SB_n : v'_n)$, the set S_b of the broker is " subscriptions at broker SB_1 with *vst* no later than v'_1 and *vet* later than v'_1, $\cdots$, plus subscriptions at broker SB_n with *vst* no later than v'_n and *vet* greater than v'_n".

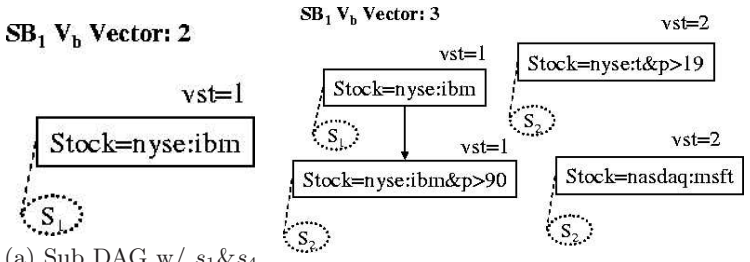
Because virtual time is generated from a leaf broker's virtual clock, $v'_i \leq VT_i$ $(i = 1..n)$. If $v'_i \geq v_i$ $(i = 1..n)$, $S_b \supseteq S_m$. Thus, the sufficiency test can be efficiently implemented as comparison of V_b and V_m on the relevant elements. That is, for a downstream where certain subscriber-connecting/leaf brokers exist, the sufficiency test succeeds when V_b is element-wise no less than V_m for those brokers. In Figure 1, when broker b_1 or b_2 routes a message to tree node N_{11}, the sufficiency test succeeds when its V_b is no less than V_m for the element of SB_1. When broker PB_1 in node N_{31} routes a message toward N_{21}, the sufficiency test succeeds when its V_b is no less than V_m for the elements of both SB_1 and SB_2.

In the remainder of this section, we describe in detail how subscriptions are assigned virtual start and end times, how subscription information is propagated, how messages are routed and how the system decides a message delivery starting point for a subscription. We discuss in Section 5.1 the optimization techniques to avoid sending full V_m vectors in messages and to cache sufficiency test results.

3.3 Assigning Subscription Virtual Start/End Times

Clients submit subscriptions to a pub/sub system through their connecting brokers. A broker assigns a virtual start time to a subscription during its subscribing process. As mentioned in Section 2, these brokers are in the leaf nodes of the redundant routing tree. A leaf broker usually aggregates subscriptions and only propagates to upstream the aggregated results.

A client subscription/unsubscription may or may not change the aggregated subscriptions of its connecting broker, depending on existing subscriptions. Whenever a leaf broker's subscriptions change and the broker decides to propagate the change to upstream, the broker advances its virtual time clock by 1. The virtual clock is maintained in an integer counter and is also used to assign virtual start/end times to subscriptions.



(a) Sub DAG w/ s_1&s_4 (b) Sub DAG w/ s_1&s_4&s_2

Fig. 2. Broker SB_1 Subscription DAGs

The leaf broker ensures the monotonicity of its virtual clock, despite crash recovery. There are many possible techniques to achieve this, such as using a monotonic system clock, or persisting an upper bound on the highest clock value. We assume that the clock time never overflows, which is reasonable for a value that is 64 bits.

We assume that a client subscription is in the form of a set of Boolean conjunctions. Any other form of Boolean expression can be transformed into disjunctive normal form (DNF). We define the following "covering" relationship of conjunctions similar to that of [5].

Definition 1. *Conjunction c_1 covers c_2, denoted as $c_1 \succeq_c c_2$, if and only if all messages matching c_2 also match c_1. That is*

$c_1 \succeq_c c_2 \Leftrightarrow \{M(c_1) \supseteq M(c_2)\}$

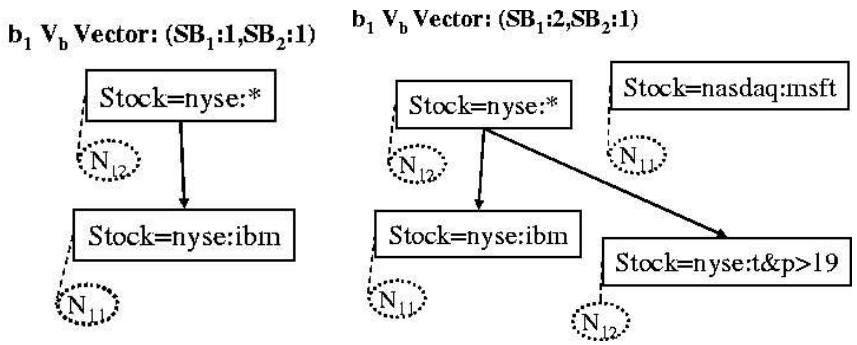
where $M(c)$ is the set of all messages matching conjunction c.

Conversely, c_2 is said to be "covered by" c_1.

Subscribing Process To assign a virtual start time to a new subscription, we analyze the covering relationship of its conjunctions with the existing subscription conjunctions. We use a directed acyclic graph (DAG) by modeling conjunctions as nodes and drawing a directed edge from a covering conjunction to a covered conjunction. As the covering relationship is transitive, we omit the transitive edges. Initially, broker conjunction DAGs are empty. Examples of conjunction DAGs are shown in Figure 2 and 3. Each conjunction is represented as a rectangle, with a virtual start time and an oval representing the list (called *subscriber list*) of subscribers or downstream routing tree nodes whose subscriptions contain the conjunction.

For each conjunction c of the new subscription, the broker finds existing conjunctions that cover c and set $c.vst$ to the minimum of these conjunctions. If no covering conjunction exists, $c.vst$ is set to the broker's current virtual time.

To ensure reliable delivery, message delivery for a new subscription cannot start until matching messages for all its conjunctions start to arrive, hence *vst* of the new subscription is set to the maximum of its conjunctions'. The new conjunction nodes are added into the conjunction DAG.



(a) Sub DAG w/ s_1&s_4 (b) Sub DAG w/ s_1&s_4&s_2

Fig. 3. Broker b_1 Subscription DAGs

Example 1. Suppose clients in Figure 1 submit the following subscriptions:
1. s_1:$stock = nyse : ibm$ (submitted at SB_1)
2. s_4:$stock = nyse : *$ (submitted at SB_2)
3. s_2:$stock = nyse : ibm\&p > 90$ *or* $stock = nyse : t\&p > 19$ *or* $stock = nasdaq : msft$ (submitted at SB_1)
The conjunction DAGs and V_b vectors of broker SB_1 and b_1 after subscription s_1 and s_4 are shown in Figure 2(a) and 3(a). When s_2 is subscribed, its first conjunction is covered by the only node in the conjunction DAG (Figure 2(a)) and thus has a *vst* of 1. The other two conjunctions are not covered and thus have *vst* of 2. Hence $s_2.vst = 2$. SB_1 increments its VT to 3 and changes the conjunction DAG to Figure 2(b). □

It is interesting to observe that the nodes in the conjunction DAG are assigned non-increasing *vst*'s as one travels from a root node to the covered nodes. Thus computing *vst* of a new conjunction only takes into consideration the immediate covering nodes. As a reminder, transitive arcs are omitted from the conjunction DAG.

Unsubscribing Process During client's unsubscription of its conjunctions, the client is removed from the subscriber lists of the nodes in the conjunction DAG. Conjunctions with empty lists are removed from the DAG. The subscription is assigned a virtual end time using the broker's current virtual time. Note that since a subscription is removed, there is no need to store information for it. The *vet* is not really assigned and is never maintained in the system. We only use it for convenience in describing the subscription sets S_m and S_b.

At the end of the client subscribing/unsubscribing process, if any roots are added or removed, the leaf node broker advances its virtual clock by one. In addition, an incremental update is generated and propagated to upstream brokers in the redundant routing tree.

We should note that although we have discussed the subscribing and unsubscribing process as if there is only one request at a time, the algorithm actually

processes requests in batches. The broker only advances its virtual clock at the end of the batch process if the set of root nodes is changed.

3.4 Propagating Subscription Changes

As described previously, if client subscription/unsubscription results in changes in the set of root nodes of a conjunction DAG, the leaf broker generates and propagates incremental changes. An incremental change contains the name of the originating leaf broker SB_1, the virtual time v_1 of SB_1 when the change occurred, a list of additive/subtractive conjunctions $\{+c_1, +c_2, \cdots, -c_i, -c_{i+1}, \cdots\}$, the subscribing tree node, and a constraint vector on receiving broker's V_b vector. c_1 and c_2 are the new root DAG nodes. c_i and c_{i+1} are the old roots that were just removed from the DAG. The constraint vector is initially $(SB_1, v_1 - 1)$. That is, we require a broker have information of all subscriptions connected at leaf broker SB_1 with $vst < v_1$.

Example 2. Continuing with Example 1, the incremental change SB_1 generated as a result of s_2 subscription is:
SB_1,2, $\{+(stock = nyse : t\&p > 19), +(stock = nasdaq : msft)\}$, N_{11}, $cons = (SB_1 : 1)$. □

The leaf broker sends the incremental change to a broker in its parent node, e.g., broker b_1 in N_{21} in Figure 1. As brokers in the same tree node are usually fully connected, b_1 forwards the incremental change to other brokers (b_2) in N_{21}. How this incremental change is propagated to all brokers in the parent node is not fixed and a specific algorithm may adapt if the topology assumptions are different.

Upon receiving an incremental change, a broker checks whether its V_b vector satisfies the constraint. If so, it updates its V_b vector and applies the additive/subtractive conjunctions by adding/removing the tree node (e.g., N_{11}) to/from the subscriber lists of conjunction nodes and inserting new conjunctions or removing conjunctions with empty subscriber lists from the DAG. A non-leaf node broker's conjunction DAG (e.g., Figure 3(a) and (b)) is similar to that of a leaf node broker's, except that vst's are not recorded for conjunctions. Same as in a leaf broker, the additive/subtractive conjunctions in the new incremental change are computed as a result of the root node changes in the DAG. If the incremental change is a pure additive change and no aggregation happened in the current broker, the constraint vector of the new change is unchanged. Otherwise, the constraint vector is set to the old V_b vector of the broker. The subscribing tree node of the new incremental change is set to the tree node where the current broker resides. The original receiving broker of the incremental change then forwards the new incremental change to a broker in its parent tree node.

If the constraint is not satisfied, the broker cannot apply the incremental change. Furthermore, if some elements of the broker's V_b vector are smaller than that of the constraint vector, the broker detects its subscription information is lagging behind, and initiates liveness mechanism to get up-to-date (described in Section 4).

Example 3. In our example, broker b_1 satisfies the constraint vector of the incremental change. As conjunction $stock = nyse : t\&p > 19$ is covered by an existing conjunction $stock = nyse : *$, it is aggregated away. Hence the new incremental change is:
SB_1, 2, $\{+(stock = nasdaq : msft)\}$, N_{21}, $cons = (SB_1 : 1, SB_2 : 1)$
b_1's V_b vector is set to $(SB_1 : 2, SB_2 : 1)$. □

In our examples below, we use a vector such as $(1, 1, 1)$ to represent V_m or V_b vectors without mentioning the leaf broker names SB_1, SB_2 and SB_3.

Similar things happen at the root broker in the routing tree. Continuing with our example, broker PB_1 updates its V_b vector from $(1, 1, 1)$ to $(2, 1, 1)$.

3.5 Data Message Routing

Data messages are published through the root broker (e.g., PB_1) of the routing tree. Before sending a newly published message, PB_1 assigns to it a V_m vector. How this V_m vector is assigned is a dimension in which the algorithm could vary and is further discussed in Section 3.7. For now, we assume PB_1 keeps a persistent record of the highest V_m vector it has ever assigned and ensures non-decreasing V_m's for new messages. When $V_b \geq$ highest V_m, V_b is used.

For an incoming message, a broker performs matching to decide which downstream routing tree nodes it should send the message to. Many efficient matching algorithms exist such as [2] [8]. Our algorithm works with any of them.

Furthermore, the broker compares its V_b vector with the V_m vector of the message for each child node. It does so by slicing both vectors with only the elements for the leaf brokers in the subtree rooted at the child node. If the V_b vector is no less than the V_m for the slicing, the broker sends the message if and only if the matching results show a match for the downstream. Otherwise, the broker conservatively sends the message down, regardless of the matching result. This is the *sufficiency test* we have mentioned in Section 3.1.

As we can see, it is possible for the broker to send down messages that do not match any subscription. This only happens at non-leaf brokers. In the leaf broker, as it always has the latest subscription information, its V_b vector (containing only one element for itself) always satisfies the sufficiency test and only the matching messages will be delivered to subscribers.

Example 4. Suppose PB_1 assigns V_m vector $(1, 1, 1)$ to message $m_1(nyse : ibm, 95)$ and $m_2(nyse : t, 20)$. m_1 is sent to b_1 and m_2 to b_2. Suppose V_b vectors of b_1 and b_2 are $(2, 1, 0)$ and $(1, 1, 0)$. Both messages will be sent to SB_1 as the sufficiency tests are satisfied for SB_1 and there is a match from node N_{11}.

Suppose now PB_1 processes an incremental change and advances its V_b vector to $(2, 1, 1)$. On the other hand, b_2 does not receive the incremental change and thus its V_b vector stays at $(1, 1, 0)$. PB_1 assigns V_m vector $(2, 1, 1)$ to $m_3(nyse : ibm, 98)$ and $m_4(nyse : t, 22)$ and sends m_3 to b_1 and m_4 to b_2. Both broker b_1 and b_2 send the messages to SB_1 because b_1 has a match and b_2, even though without a match for N_{11}, detects its sufficiency test fails. □

Justification of Correctness The sufficiency test is satisfied when a broker's V_b vector is equal to or greater than V_m with regard to the relevant leaf brokers. When it is greater, the broker can have *wider* (matching more messages) conjunctions as new subscriptions may have happened. It can also have *narrower* conjunctions as unsubscriptions may have happened. When the conjunctions are wider, the broker obviously passes all messages matching the subscriptions required by the V_m vectors plus more that match the new subscriptions. In the narrower case, a broker drops messages matching only unsubscribed subscriptions at a leaf broker sb. Those subscriptions have $vet \leq V_b(sb) \leq VT_{sb}$ and hence are not in the S_m set.

3.6 Detecting Subscription Delivery Starting Point

For a new subscription, its connecting leaf broker must decide a safe point from which the system can deliver a gapless, in-order stream of published messages. The actual protocol for reliable delivery is a complicated scheme and has been discussed in our work in [3] [4]. We do not deal with that problem here, rather we deal with how subscription propagation will not produce subscription information that is wrong for reliable delivery. Our solution can work with any reliable delivery protocol.

Detecting delivery starting point is accomplished by comparing a message's V_m vector element with the vst of the subscription s. As soon as the leaf broker sb sees a message with $V_m(sb) \geq s.vst$, it starts to deliver matching messages for s.

Example 5. Broker SB_1 receives m_1. Even if m_1 matches subscription s_2, this is not a delivery starting point for s_2 as m_1's $V_m(SB_1) == 1$ and is less than $s_2.vst == 2$. This is correct because there is no guarantee a later message $(nyse : t, 20)$ will not be filtered out if it is routed through broker b_2. Broker SB_1, instead, starts delivery for s_2 from message m_3. □

3.7 Algorithm Variants

We have extended our basic algorithm with two variants. These extensions continue to support reliable delivery. Due to space restriction, we only provide brief description of these extensions:

- Non-monotonic assignment of V_m vectors to messages in a published stream;
- Non-fixed V_m vector for a message m.

Our basic algorithm stores persistently the highest V_m assigned and assigns monotonically non-decreasing V_m vectors to messages at the publisher connecting brokers. Our extension does not require persistent storage and allows non-monotonic V_m vector assignments to messages. This could happen due to a publisher connecting broker crash/recovery.

In the basic algorithm, a message's V_m vector is fixed once assigned. In some cases, the aggregation of subscriptions may result in incremental updates with

empty lists of additive/subtractive conjunctions. The basic algorithm requires such *empty* updates to propagate to the root broker as a means of conveying the latest V_m the broker could use. Our extension does not require propagation of this kind of update, rather we record the fact that a V_m vector could be automatically changed to V'_m due to the empty incremental update. This is recorded at the last broker where such an empty update occurred.

In both extensions, not only are published messages assigned V_m vectors, so are the silence periods between them. In addition, the leaf brokers hosting subscribers perform monotonicity checking on message/silence V_m vectors and initiate negative-acknowledgments if monotonicity is violated.

4 Liveness and Failure Handling

Our solution intrinsically supports pub/sub message delivery with high availability and light failover. The solution itself has to deal with failures as incremental subscription updates can be lost and arrive out-of-order. This section describes the failure handling with regard to the subscription propagation protocol. This is an important part of our work, however, due to space restriction, we only describe it briefly.

Broker Crash Recovery Upon recovery from a crash, a leaf broker sets its virtual clock time VT to be greater than all previous values it has assigned. The leaf broker initializes its conjunction DAG to contain conjunctions for the durable subscriptions ([1][4]) it maintains. All conjunctions are then propagated by sending a full state update and the current VT. The broker then advances its VT by one.

Non-leaf brokers recover from a crash by initializing an empty conjunction DAG and resetting its V_b vector to all 0's.

Leaf Broker Driven Liveness As the sources of subscription changes and virtual times, leaf brokers ensure all publisher connecting brokers receive up-to-date subscription information and assign latest V_m vectors to messages. For subscription/unsubscription received during virtual time vt and propagated, a leaf broker SB maintains an expected starting time from which data messages should have V_m vector with an element vt' for SB such that $vt' \geq vt$. This expected time can be dynamically adjusted through similar techniques that estimates TCP round trip times. Messages received after the elapsed time with $vt' < vt$ trigger a full subscription state update with SB's latest propagated VT. Alternatively, SB can repeat the incremental updates sent from vt' to vt. This requires a cache for the latest incremental updates at SB.

Non-Leaf Broker Driven Liveness A non-leaf broker (including publisher connecting brokers) b detects that its subscription information is lagging behind in two ways: (1) receives a data message with V_m vector that is greater than V_b vector for the elements of downstream leaf brokers; (2) receives incremental updates

with constraint vectors $\not\leq V_b$. Broker b initiates a negative acknowledgment message toward the leaf broker SB for whom b's V_b vector is lagging behind. Such negative acknowledgment may be satisfied by SB or a broker on the route from b to SB with the required subscription information.

5 Implementation and Experimental Results

5.1 Implementation

Our implementation performs monotonic and fixed V_m vector assignment at the publisher connecting brokers. These brokers persist the highest V_m vector assigned. The liveness and failure handling utilizes full-update messages.

Subscription aggregation is based on covering relations (Section 3). Our current covering relation is restricted to identical conjunctions. Since the covering test is a black-box component in our implementation, a sophisticated covering relation can be plugged in to replace this simple one.

The number of leaf brokers affects system performance and scalability, since it directly impacts the length of V_m and V_b vectors and computations involving them. In our implementation, we utilize the property that links are usually FIFO. This is true of links implemented as TCP connections such as in Gryphon.

Instead of full V_m vectors, we use fixed-length vector digests. When a new V_m is first assigned to a message, the publisher connecting broker assigns a digest to V_m by taking a snapshot of a monotonically increasing value. This can be implemented in many ways, such as using the system clock. As publisher connecting broker assigns monotonically non-decreasing V_m vectors and digests are monotonic, the system satisfies the following monotonicity - $d < d'$ if $V_m < V'_m$ with regard to the same assigning broker. A broker does not persist the digests, therefore it can assign two different digests to one V_m at different times. Thus $d < d' \Rightarrow V_m \leq V'_m$.

In this scheme, the first message with a new V_m vector going down each link carries the original V_m and its digest. Later messages only need to carry the digest until the V_m changes or the link fails and recovers.

In addition, each broker maintains a cache of the sufficiency test results. This cache is indexed by the digest and the assigning broker. It only maintains an entry for the highest digest it has seen from a publisher connecting broker. From the digest monotonicity, any message originated from the same broker with a digest no greater than the cache entry can reuse the cache result. Otherwise, the broker conservatively computes $false$ as the sufficiency test result for the message. The cache entry is updated when the broker advances its V_b vector or when it receives a message with higher V_m vector and digest. We omit the details of a negative-acknowledgement scheme that is used to retrieve the original V_m vector for a digest if the message carrying the mapping was lost.

5.2 Experiment Results

The testbed for our experiments is a set of RS6000 F80 machines with six 500MHz processors and 1G RAM. Each machine has dual network interfaces

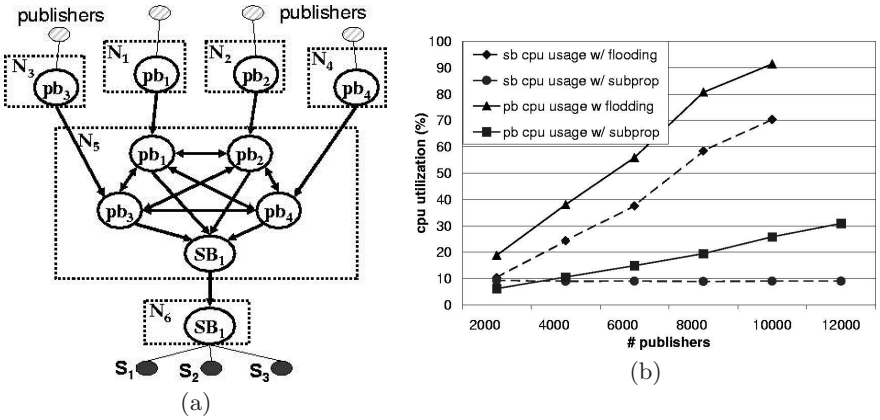


Fig. 4. System Load(CPU) Comparisons

and is connected through a 100Mbps Ethernet network and a gigabit switch to other machines.

We focus primarily on metrics that are impacted by the specifics of our solution. Metrics such as routing table (conjunction DAG) size and subscription incremental update message size are common to subscription propagation algorithms and have been investigated in previous work such as [12, 5].

System Load Comparison in Selective Subscription Tests This test compares the system overhead of using subscription propagation with that of using flooding when client subscriptions are selective. The workload is motivated by sensor networks where there are many publishers collecting and publishing various kinds of data and relatively few subscribers that selectively subscribe to data of interest.

The test is set up as Figure 4(a) with 4 publisher connecting brokers pb_{1-4} and 1 subscriber connecting broker sb, each in its own tree node. These brokers also reside in an intermediate tree node N_5 and thus each implements two virtual brokers. Four redundant routing trees can be defined by taking each of pb_{1-4} as root node.

We fix the number of subscriptions at 2000 and vary the number of publishers from 2000 to 12000. The message rate per publisher is fixed at 2 messages/second. Each subscription is distinct and to the messages published by 1 publisher. Hence, the total receiving message rate of subscribers is 4000 messages/second throughout the test. The publishers are evenly distributed among pb_{1-4}. Figure 4(b) shows the CPU utilization of sb and pb_{1-4} (averaged) in both schemes.

With 2000 publishers, all published messages are subscribed. This case is not favorable to subscription propagation. However, CPU utilization difference at sb in flooding and subscription propagation schemes are negligible due to the use of efficient matching algorithm [2]. In addition, when the number of publishers

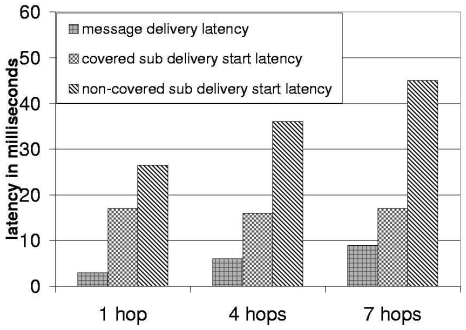


Fig. 5. Subscription Delivery Start Latency & Message Delivery Latency

increases, *sb* CPU utilization stays constant in the subscription propagation scheme but increases linearly in flooding scheme even though the number of *useful* messages does not change.

In both schemes, CPU utilization at pb_{1-4} increases linearly with the number of publishers. However, the flooding scheme shows a much steeper slope because each of pb_{1-4} not only accepts messages from publishers but also sends these messages to other *pb*'s and receives all messages published through other *pb*'s. As a result, CPU utilization at pb_{1-4} reaches $> 90\%$ with only 10000 publishers in the flooding scheme compared to 31% with 12000 publishers with subscription propagation.

Latency Measurements This test examines two latency metrics. Message delivery start latency ($DSLat$) measures the time elapsed from a subscription is submitted to the first message is delivered to it. Message delivery latency ($DLat$) measures the time it takes the system to deliver a message to an existing subscription. We use a linear topology consisting of broker *pb* for publishers and *sb* for subscribers each in its own routing tree node. *pb* and *sb* are connected through n hops of intermediate tree nodes, each with one broker.

$DLat$ is measured by a latency sampler(LS) that publishes messages through *pb* and subscribes to its own messages through *sb*.

$DSLat$ for a new subscription is measured at the subscriber by taking the difference between time of subscription and time of first message delivery. We further distinguish $DSLat$ for subscriptions that are covered locally in *sb* and subscriptions that are not covered hence must propagate to *pb*. We call the first $DSLat$ *local* and the latter *global*. Global $DSLat$ is the sum of the following times: (1) time taken to send subscription to *sb*; (2) processing time at *sb*; (3) processing time at other brokers; (4) network delays(bi-directional) at each hop; (5) expected interval till next message published that matches the subscription. If messages that match the subscription are published at a steady rate every t milliseconds (ms), this time is $t/2$ on average; (6) time taken to send the message from *sb* to the subscriber. Similarly, local $DSLat$ is the sum of (1),(2),(5),(6).

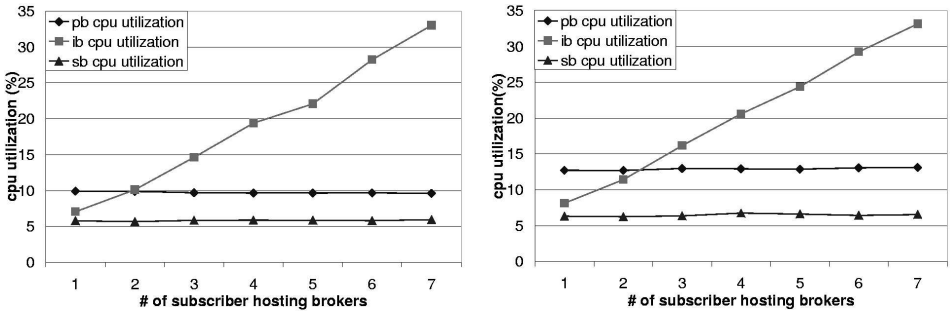
We make time (5) negligible by using a high publishing rate (200 messages/second/topic) but on few (16) topics. The aggregated publishing rate is 3200 messages/second. In addition, since we are mainly interested in measuring processing overhead (2) and (3), we do not inject additional latency on the links. Since we are running in a LAN environment, (1), (4) and (6) are small. We impose load on *sb* by connecting 60 steady subscribers to 8 topics. Local $DSLat$ is measured by subscribers that dynamically subscribe and unsubscribe to the same 8 topics as the steady subscribers and global $DSLat$ is measured by subscribers that dynamically subscribe to the remaining 8 topics. We take the median of all measurements. Figure 5.2 shows $DLat$, local and global $DSLat$ when there are 1,4 and 7 hops from *pb* to *sb*.

Message delivery latency $DLat$ increases linearly from 3 to 6 to 9 ms. Local $DSLat$ - delivery start latency for covered subscriptions - stays roughly constant at $17ms$. Global $DSLat$ increases linearly from $26.5ms$ to $36ms$ to $45ms$. The differences of global and local $DSLat$, which shows the network latency and broker (other than *sb*) processing overhead also increases from $9.5ms$ to $20ms$ to $28ms$, which shows that subscription processing overhead is small.

Scalability Measurements This test examines the system scalability with regard to the V_m and V_b vector sizes, i.e., number of subscriber connecting brokers. The test is set up with a broker *pb* for publishers and a broker *ib* connecting *pb* to a number of brokers sb_{1-n} for subscribers. Each of these brokers is in a separate routing tree node. We vary the number n of *sb*'s.

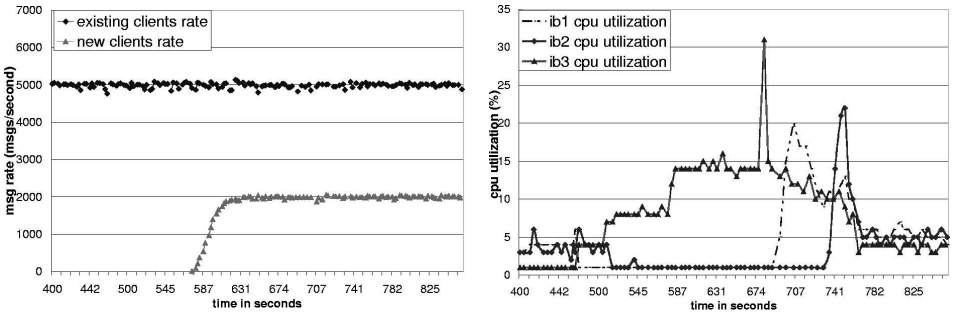
Messages are published through *pb* at a fixed rate of 2000 messages/second throughout the test. We demonstrate the result in a setup that is not favorable to subscription propagation to show the low overhead incurred by it compared with the flooding scheme. We set up each published message to be subscribed by some subscriber at each *sb*. Subscribers are evenly distributed onto *sb*'s. Each *sb* has two groups of subscribers with each group receiving 10000 messages/second. One group is steady and the other is dynamic with periodic unsubscriptions followed immediately by re-subscriptions. Every 2 seconds on average, an unsubscription/subscription occurs at a *sb* and causes the *sb* virtual time to advance by 2. Thus, this simulates the situation where the broker virtual time advances by one every second at each *sb*. In situations where subscription/unsubscription occurs more frequently, they could be batched to reduce the rate at which the *sb* virtual times advance. Figure 6 shows the CPU utilization at *pb*, *ib* and sb_{1-n}(averaged).

In both the flooding scheme and with subscription propagation (Figure 6(a) & (b)), CPU utilization at *pb* and sb_{1-n} stays constant with n changing from $1-7$. This is due to the Vector-Digest and caching scheme we described. The CPU differences at *ib* and *sb*'s between the two schemes are also very small. There are (3%) differences on the *pb* CPU utilization. This is due to the sophisticated message encoding used in Gryphon. As *pb* assigns V_m digest to a message, the message has to be re-encoded and this is not needed with flooding in Gryphon. Such difference can be eliminated by encoding optimizations.



(a) flooding algorithm (b) subscription propagation

Fig. 6. Scalability: Broker CPU Utilization Versus Number of Brokers



(a) subscriber message rate (b) CPU utilization at ib_{1-3}

Fig. 7. client message rate and intermediate broker CPU utilization with crash failure

Failure Test This test demonstrates the lightweight failover characteristics of our approach. Even in the absence of a majority of brokers in a cell, our algorithm is able to accept new subscriptions and deliver messages for them. When a path fails, the system switches to the remaining available paths and provides continuing service.

The test is set up as a redundant routing tree of 4 nodes and 6 brokers: broker *pb* for publishers and sb_{1-2} for subscribers, each in a separate node N_{pb}, N_{sb_1} and N_{sb_2}. Broker ib_{1-3} are all in one node N_{ib}. Node N_{pb} is connected to N_{ib}, which further connects to N_{sb_1} and N_{sb_2}.

In this test, traffic from *pb* to sb_{1-2} is shared among ib_{1-3}. Messages are published through *pb* at a rate of 2000 messages per second on 100 topics. Initially, there are 2 groups of clients connected to sb_1 and 1 group to sb_2, each group with 250 subscribers evenly distributed among the first 50 topics. Thus, the aggregated message rate per group is 5000. A fourth group of 100 clients to the remaining 50 topics connect at a later time. The aggregated message rate for this group is 2000. Figure 7(a) shows the message rates for one of the first 3 groups and the fourth group. Figure 7(b) shows the CPU utilization at ib_{1-3}.

At time 400, since only the first 50 topics are subscribed to, the messages are routed through ib_1 and ib_2. Broker ib_3 is not used because of the simple hashing scheme used for load balancing. At time 475, ib_1 crashes, the system fails over to ib_3, and CPU utilization at ib_3 increases to 4% to the same as ib_2. About 30 seconds later, at time 505, ib_2 crashes, and all messages on the first 50 topics are routed through ib_3. CPU utilization at ib_3 doubles to 8%. During these routing changes the client message rate is not affected. At time 565, a new group of 100 subscribers to the latter 50 topics starts to connect. Even though only ib_3 is available, our approach is able to make progress and starts to deliver messages for the new clients. When ib_1 and ib_2 recover about 130 and 160 seconds later at time 691 and 731, traffic is once again shared among the available paths. During this process, service to clients is not affected as their message rate stays constant.

6 Related Work

In this section we survey previous work on subscription propagation and aggregation in publish/subscribe systems. Techniques for subscription aggregation ([12]) are complimentary to our work.

Siena [5] and XNet [6] support subscription propagation and aggregation to achieve scalability. Their topology has redundancy, with multiple routes between servers. However, the subscriptions are only propagated along a single selected "best route" in a spanning tree. If a failure occurs on the selected path, the system must select another path and subscription information need to be set up for the new path before message routing can be resumed. As a result, recovery from a spanning tree link failure is slow. In addition, these works do not address the support of reliable delivery.

Elvin [17] is mainly designed around a single server that filters and forwards producer messages directly to consumers. It doesn't have a scalable solution for multiple servers.

Snoeren et. al [18] propose an approach for improving reliability and low latency by sending simultaneously over redundant links in a mesh-based overlay networks. The protocol does content-based routing and provides high level of availability. However, there is no guarantee of reliable delivery when subscriptions are dynamically added and removed.

REBECA [12, 13] supports subscription propagation and aggregation over a network constructed as a tree of brokers. Their subscription aggregation techniques, such as filter merging, are applicable to our work. The system has a self-stabilization component that uses time based leases to validate routing entries in brokers. This is a viable technique for best-effort delivery, but does not support reliable delivery since it is possible for a broker to filter a message that is relevant for a downstream subscriber.

JEDI [7] guarantees causal ordering of events. Their distributed version of event dispatcher constitutes of a set of dispatching servers interconnected into a tree structure. This distributed version, while addressing part of the need of

Internet-wide distributed applications engaging in an intense communication, does not accommodate and utilize redundant links between dispatching servers and hence is not highly available and easy for load sharing.

Tapestry [20] provides fault tolerant routing by dynamically switching traffic onto precomputed alternate routes. Messages in Tapestry can be duplicated and multicast "around" network congestion and failure hotspots with rapid reconvergence to drop duplicates. However, it does not support content routing.

Scribe [15] is a large-scale and fully decentralized event notification system built on top of Pastry - a peer-to-peer object location and routing substrate overlayed on the Internet. It leverages the scalability, locality, fault-resilience and self-organization properties of Pastry. However, Scribe does not support content-based routing and wild card topic subscriptions . The system builds separate multicast trees for individual topics using a scheme similar to reverse path forwarding and inverts the subscription message path for later event distribution. This makes it impossible to add a node to the multicast tree for load sharing. The system recovers from multicast node failures by building new trees. It does not support reliable delivery, and unsubscription has to be delayed until the first event is received.

Triantafillou et. al [19] proposed an approach to propagating subscription summaries and performing event matching and routing. Their subscription propagation algorithm, which affects the way events are routed, requires each broker to have global knowledge on the broker network topology. In addition, the approach does not support reliable delivery.

Since Lamport's work on logical time and clocks [11], significant work has been done using logical time such as virtual time [9], version vectors [14], vector times [16] and multipart timestamps [10]. Our work shares the property that logical time vectors are used as a concise form for representing large information. However, these works are focused on detecting state inconsistencies or causal relationships, which is only part of the problem subscription propagation is facing in pub/sub systems.

7 Conclusions

We have developed algorithms supporting subscription propagation with high availability, easy load-sharing and light failover in a content-based pub/sub system over a redundant overlay network. The algorithm does not require agreements or quorum among redundant brokers. We also presented the experiment results that show the high performance, scalability, low latency and availability. Future work includes investigation into support for event advertisement ([5]), adaptively applying subscription propagation according to different subscription profile and locality.

References

1. Java (tm) message service. In *http://java.sun.com/products/jms/*.

2. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Principles of Distributed Computing, 1999*, pages 53–61, May 1999.
3. S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, pages 7–16, 2002.
4. S. Bhola, Y. Zhao, and J. Auerbach. Scalably supporting durable subscriptions in a publish/subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2003)*, pages 57–66, 2003.
5. A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
6. R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'03), Cambridge, MA*, April 2003.
7. G. Cugola, E. D. Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.
8. F. Fabret and et. al. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):115–126, 2001.
9. D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.
10. R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *ACM Symposium on Principles of Distributed Computing*, 1990.
11. L. Lamport. Time, clock, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.
12. G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, September 2002.
13. G. Mühl, L. Fiege, and A. P. Buchmann. Filter similarities in content-based publish/subscribe systems. In *Proceedings of International Conference on Architecture of Computing Systems (ARCS'02)*, 2002.
14. D. Parker and et.al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.
15. A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of 3rd International Workshop on Networked Group Communication (NGC 2001), UCL, London, UK*, November 2001.
16. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, pages 149–174, 1994.
17. B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proceedings of AUUG2K, Canberra, Australia*, April 2000.
18. A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using xml. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.
19. P. Triantafillou and A. Economides. Subscription summarization: A new paradigm for efficient publish/subscribe systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, 2004.
20. B. Zhao, L. Huang, A. Joseph, and J. Kubiatowicz. Exploiting routing redundancy using a wide-area overlay. Technical Report UCB/CSD-02-1215, University of California, Berkeley, 2002.