Publish and Subscribe with Reply

Jonathan C. Hill, John C. Knight, Aaron M. Crickenberger, Richard Honhart
Department of Computer Science, University of Virginia
{jch8f, jck, amc6q, rph5d}@virginia.edu
TECH REPORT: CS-2002-32

Reply for Publish and Subscribe allows receivers of a publication to reply to the publisher. We demonstrate that Reply is a natural, efficient, and useful component of Publish/Subscribe. It is natural because it maintains the weakest possible coupling between senders and receivers. It is efficient because it stores computations discarded in publication forwarding, later applying them to channel replies. Most importantly, it is useful because it increases the domain of applications for which Publish/Subscribe is suited. This paper includes discussion of Reply's utility, introduction of two algorithms with differing state storage and capabilities, their analysis for worst-case conditions, modeling of required resources, and presentation of a modular implementation of Reply for distributed Publish/Subscribe systems.

1. Introduction

Distributed Publish/Subscribe offers powerful and flexible message-delivery to distributed systems. It is particularly useful for very large distributed systems where "one-to-many" messaging is beneficial and the "book-keeping" of point-to-point component interconnections is cumbersome.

Publish/Subscribe obtains its power and flexibility by supporting independence of messages senders and receivers. A Publish/Subscribe service registers receivers by the types of messages they will receive. (Either by category, formal type, or constraints over content.) When a publisher pushes a message into the system, all relevant subscribers receive the message. This allows publishers and subscribers to operate independently, supporting asynchronous event-level architectures that scale well with system size. As the scale of application systems continues to increase, their inclusion of Publish/Subscribe-like functionality is likely to increase.

Many application systems require Request/Response messaging. Request/Response messaging couples messages in message/response pairs. A message is sent to a receiver and a response is awaited. The receiver responds by sending a reply to the sender. Request/Response is typically used in client/server architectures. A client generates a request and sends it to a server. Then a server acts on the request and responds to the client. This component interaction is intuitive and fits well with commonly used programming paradigms. Despite its origins in point-to-point communications, Request/Response will continue to play an important role in large distributed systems.

A significant class of distributed systems would benefit from Publish/Subscribe, but requires features of Request/Response. Applications in this category include command-and-control infrastructures [7] [8], gnutella-like distributed information retrieval systems [2], and content dissemination services. In general, these systems are 'data-push'-oriented, but require client/server interactions. They represent a 'new' kind of distributed architecture that is both Request/Response-like and Publish/Subscribe-like. More generally, as systems that once relied on point-to-point Request/Response attempt to achieve increasingly distributed and scalable architectures, they will benefit from Request/Response-like capabilities in a Publish/Subscribe-like framework.

Publish/Subscribe can support Request/Response semantics if it supports a reply capability. A reply capability allows each receiver of a message to respond to the message's publisher. A reply might consist of an acknowledgment of receipt, a return of requested data value, or the results of a computation. Traditionally, Publish/Subscribe has avoided internal support for a reply feature because of concerns about its effects on the paradigm. Researchers recognize Publish/Subscribe as a decoupling of message senders from message receivers in both space and time [6]. A reply feature might alter this decoupling at the expense of the paradigm's flexibility. It is also argued that Publish/Subscribe is an information 'pushing' technology with filtering at receivers. A reply capability might complicate a user's understanding of the correct application of the paradigm. Furthermore, Publish/Subscribe is efficient for large distributed networks as it does not store information about forwarded publications. Worst-case scenarios for a reply algorithm, one that must maintain such state, might limit efficiency or application of Publish/Subscribe.

We argue that a reply feature can be a useful, efficient, and natural element of Publish/Subscribe. We refer to the

augmented paradigm as *Publish and Subscribe with Reply.* Our arguments are based on the definition and analysis of two reply algorithms, as well as the implementation and application of one of them.

Section 2 introduces Reply, a reply function for Publish/Subscribe. It considers the mention of reply functions in existing Publish/Subscribe architectures. Section 3 introduces our algorithms for Reply. Section 4 is an analysis of the worst-case performance and resource requirements for our Reply algorithms. Section 5 discusses the effectiveness and utility of Siena Harvest-an implementation of Reply in the Siena Publish/Subscribe architecture.

2. Publish and Subscribe with Reply

An effective Reply mechanism must not impede the power and flexibility of Publish/Subscribe. This is complicated by its decoupling [6] and 'data-push' properties. We introduce these properties of Publish/Subscribe and consider their implications for Reply as follows:

- **Spatial Decoupling**: A publisher may be unaware of a network's receivers. Therefore, it may not have explicit control over how many receivers obtain its message. This means that a publisher cannot assume the size of a receiver set for a given message. To be a natural and effective component, Reply must handle messaging with similar temporal efficiency to Publish/Subscribe so that publishers are not required to consider the time-cost of Reply independent of the time-cost of publication. That is, in the event that all receivers reply to a message simultaneously, the time to return the replies to the publisher should be similar to the time to send the publication to all receivers.
- **Temporal Decoupling**: Receivers are not obligated to process a message at any time. Hence, a publisher cannot assume a receiver will process a message and send a reply. Reply must handle temporally decoupled reply events while providing useful information to the publisher and reasonable behavior of services at publishers and repliers.
- **Information 'Pushing' Technology:** Publishers push information and subscribers filter it by content. Hence, Publish/Subscribe is an information 'push' technology [10]. A Reply feature must not alter the efficiency of Publish/Subscribe with respect to information pushing.

It is easy to construct a reply system that would violate these constraints. Consider the action of a naive, point-to-point reply mechanism when a publication from a single source is sent to millions of receivers: A response from each receiver would result in millions of individual reply messages sent towards the publisher. Many of these replies would traverse the network simultaneously. This would generate a massive sequence of message events at the publisher over a short interval of time. The effect would be an overloading of the publisher with message events. Second, some replies would proceed to the publisher at dispersed times in the future, perhaps well after the scope of their effective consideration has been abandoned by the publisher. The publisher application would still be required to accept these untimely replies. Finally, the effective routing of millions of replies to a single point would require reactive network-level resource management to avoid affecting performance of additional publications.

2.1 Existing Reply Functions and Related Work

The utility of a reply feature has resulted in its appearance in varied form in at least two Publish/Subscribe architectures. JEDI, a distributed Publish/Subscribe architecture, introduces a reply function to support functionality required by a wide-area distributed workflow system [4]. Reply was introduced after they investigated the value of Publication/Response synchronization and the undesirable overhead of an application-level point-to-point response mechanism. They note the ability of an integrated reply function to maintain the anonymity of publishers and subscribers. They also discuss the role of a reply service in presenting a publisher with a single reply event as opposed to independent replies from each receiver.

Java Messaging Service [9] introduces a reply function in two contexts: it supports a built in message acknowledgment feature, and otherwise suggests ad hoc application-level reply. The algorithm of the acknowledgment feature is not discussed. Ad hoc application-level reply exposes location information to receivers, or otherwise requires rebinding of receivers to publishers.

To the best of our knowledge, the algorithms and scalability of existing reply mechanisms have not been discussed in the literature. For example, a reply mechanism's worst-case scenarios and their respective costs have not been demonstrated.

Directed diffusion [12] [13] is the reduction of data messages through data fusion of messages on route. Its application is to sensor networks where queries are met by distributed filtering of sensor events. Its application is to effectively reduce the event cost of messaging [12] and provide intelligent, content-based data fusion [13] within the network service. This saves power and reduces application complexity. Our goal is to reduce message traffic so that reply to a publication will scale with the efficiency of distributed Publish/Subscribe. We apply data fusion at the forwarding level provided by Publish/Subscribe, rather than the routing level, in order to achieve this scalability.

2.2 A Reply Function

Publish and Subscribe with Reply is an extension of the existing Publish and Subscribe model. An example of its event cycle is depicted in Figure 1. The steps of Publish and Subscribe with Reply shown in the figure are as follows:

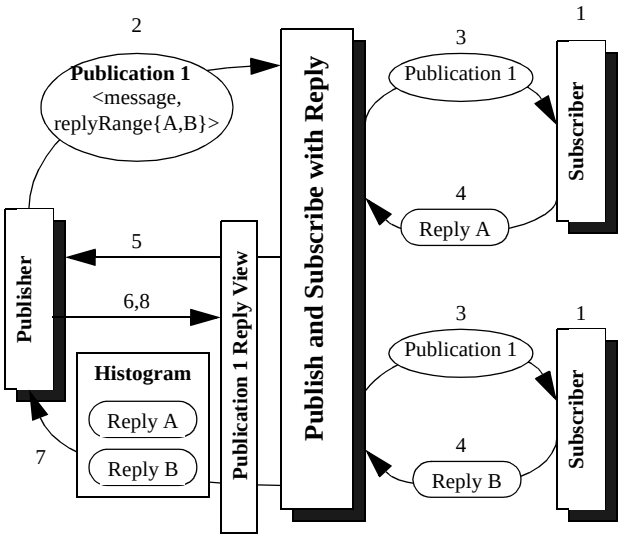

FIGURE 1. The Publish and Subscribe with Reply event structure with an example publication.

1. Subscribers are elements that subscribe to messages based on message type or content. They will be receivers of any messages matching their subscriptions.
2. A sender publishes a message known as a publication. In addition to its message content, the publication specifies that it will accept a reply. It specifies the reply data type defining legal reply values. In step 2 of Figure 1, a "Publisher" has published "Publication 1."
3. After a message is published, it arrives at all receivers subscribed to the message's content (type.)
4. The receivers observe that the message requests a response, and *if they choose*, they may respond. They respond to the Publish/Subscribe system by issuing an instance of the specified reply data type along with the publication's unique identifier. A receiver may only reply to a given publication once. In Figure 1, the topmost subscriber responds with "Reply A" and the bottom subscriber responds with "Reply B."
5. The publisher of a message observes replies to the publication through a *reply view.* A *reply view* is represented by a token given to the publisher to observe reply data. A publisher uses a *reply view* token to observe reply data that has returned to the publisher.
6. The publisher may request currently accumulated replies, or request replies after blocking for a specified time interval. It makes such requests to the reply view token for a given publication.
7. Replies to a publication are viewed by the publisher as a histogram sent to the publisher. A histogram indexes over distinct replies from the range of the reply data type specified in the initial publication (see step 2). The count of each index is the number of receivers replying with the associated reply data value.
8. A reply view terminates when a publisher returns the view token to the system.
9. Replies sent after the reply view duration of a publication are discarded by the system.

We refer to the set of replies returned to the publisher as a *reply harvest*. We call the process of gathering replies *reply harvesting*.

There are three key elements of Publish/Subscribe with Reply that distinguish it from Publish/Subscribe. First, Reply tells a publisher how many subscribers have replied with each instance value of the reply data type. Second, Reply creates an intrinsic temporal ordering between a publication and its replies. Third, Reply allows information to flow from receivers of a message to the message's publisher.

Publish/Subscribe with Reply maintains decoupled space and time. Publishers and receivers are never implicitly made aware of each other's locations, nor is a publication or its replies addressed by location. Likewise, there is no temporal coupling between a publisher and a subscriber. A publisher listens for replies for a time interval independent of subscribers. A replying subscriber may reply at any time (or not at all), before or after the interval during which a publisher is listening.

3. Reply Algorithms

Space and time decoupling imply that a publisher cannot assume the size of a reply harvest or the response time of replies. In Section 2 we discussed the requirements created by space and time decoupling. Of remaining concern are the efficiency requirements for an algorithm implementing the stated Reply function. To re-iterate, a reply algorithm must be of similar efficiency to Publish/Subscribe, and it must not significantly alter the efficiency of existing distributed Publish/Subscribe algorithms.

3.1 Observations

Much of the computation required for an efficient Reply algorithm is provided by the forwarding algorithms of dis-

tributed Publish/Subscribe. Forwarding computations-though discarded by Publish/Subscribe-have intrinsic value that we now elucidate. A key observation is this:

- A publisher is independent of receivers, but during the forwarding of a message, the dynamic association of a publisher with all subscribing receivers is computed.

This observation suggests that the core computation of Publish/Subscribe is a temporary association of a set of receivers with a publisher.

Consider what happens in a distributed, subnet-based Publish/Subscribe system. An example of the forwarding of a message in such a system is depicted in Figure 2. The diagonally-hatched circle represents the publisher of a message. Darkly shaded circles represent receivers that obtain the message. The message is passed from dispatcher to dispatcher in a process called forwarding. We represent forwarding steps as the edges in the graph. At each dispatcher, the message is forwarded to zero or more receivers subscribed to the message, and to zero or more other dispatchers.

In subnet-based forwarding the path of a message's dissemination forms a tree. This tree is a data structure associating receivers with a publisher for a given message. It has the following properties for a message:

1. The root of the tree is a dispatcher local to the message's publisher.
2. The intermediate nodes of the tree are dispatchers forwarding the publication.
3. The leaf nodes of the tree are the receivers of the message.

A tree is created for each message published, and its structure depends only on the forwarding of the message. In turn, the message's forwarding depends only on the subscriptions of potential receivers and the connection topology of the Publish/Subscribe network.

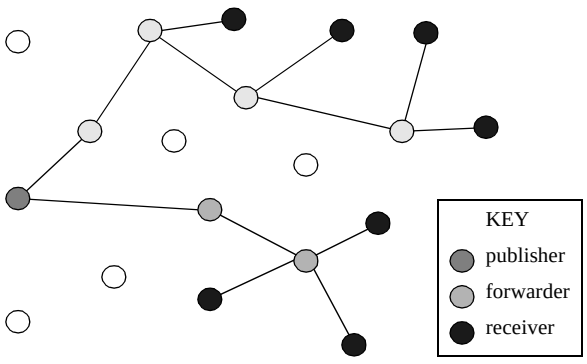

FIGURE 2. An example of a forwarding tree generated by forwarding of a publication.

A forwarding tree presents two important levels of information. First, as we have earlier stated, it associates a publisher with a set of receivers. The root of the tree represents the publisher and the leaves of the tree represent the receivers. More important is the form of this association in the tree. Each receiver is associated with the publisher by a path of nodes and edges from the receiver's leaf node to the root of the tree. One or more receivers might share any given intermediate node of the tree as a common node on their paths to the root. Additionally, a common node shared between two paths to root indicates that both paths share a common path postfix to the root node.

The tree data structure formed during message forwarding provides the basis for efficient Reply algorithms. The distribution of replies over the paths of the forwarding tree distributes the cost of reply message traffic over the links of the tree. Meanwhile, all reply paths eventually coincide, and this combination of paths is also distributed in the tree. A function to merge replies can be applied wherever reply paths merge. A merging function potentially reduces the overall message traffic sent back to the root of the tree.

3.2 Algorithms

Our algorithms re-use the forwarding tree of a publication. The algorithms send replies from the leaves of their publication's forwarding trees to their roots. As stated in Section 3.1, re-using the paths of a publication's forwarding tree distributes the return-paths of reply messages and supports merging of replies. Resource distribution in message passing and data merging is a primary element of the scalability of reply algorithms. In very large networks, where individual dispatchers have limited computational and bandwidth resources, distribution allows arbitrary publications to distribute the cost of reply collection just as Publish/Subscribe distributes the cost of publication dissemination.

Publish/Subscribe does not require the storage of forwarding tree information. Therefore, its use in Reply requires that we store the information as a cost of Reply. How this information is stored effects Reply's costs and capabilities. We introduce two Reply algorithms representing distinct options for the storage of forwarding tree information.

Dispatcher-Stateful-Reply is a Reply algorithm that stores forwarding tree state in the dispatchers of the forwarding tree. Its disadvantage is that it requires dispatchers store state regarding publications. Its advantages include reduced message traffic and bounding of the quantity of reply messages received by any dispatcher. Additionally, the reply view can calculate a lower bound on the number of outstanding replies without requiring additional message traffic.

Message-Stateful-Reply is a Reply algorithm that stores forwarding tree state in publications and their replies rather than in dispatchers. Its advantage is that dispatchers remain stateless for publications. Its primary disadvantages are that it cannot calculate the number of outstanding replies without additional messaging, and cannot merge replies to reduce reply traffic. Additionally, the length of a publication message increases with each forwarding in the dispatcher network.

In general, these algorithms work as follows:

1. A receiver generates a reply as an instance of the reply data type specified by the causing publication.
2. Receivers send their replies as ordered pairs, **(publication identifier, reply data instance)**, to the Reply service. The service is contacted at the dispatcher from which the receiver obtained the publication.
3. The Reply service sends replies back along a path in the causing publication's forwarding tree. The path is from a leaf of the tree to its root.
4. At each intermediate node of the forwarding tree, Dispatcher-Stateful-Reply performs a merging function with other replies for the publication. Message-Stateful-Reply does not.
5. The reply results are merged into a histogram at the root of the tree and presented in histogram form to the publisher.

We now discuss each algorithm in more detail.

3.2.1 Dispatcher-Stateful-Reply

Dispatcher-Stateful-Reply stores publication and reply information at all dispatchers within a publication's forwarding tree. For a given publication, each dispatcher that is part of the publication's forwarding tree stores (1) the local edges of the tree (an edge to the dispatcher from which this dispatcher received a publication, and edges to the dispatchers to which it forwarded the publication) and (2) the reply values it receives over the edges to which it forwarded the publication.

Dispatcher-Stateful-Reply functions as a distributed algorithm over all dispatchers in a Publish/Subscribe with Reply system. The algorithm is initially active at each dispatcher within a publication's forwarding tree when that dispatcher receives the publication. The algorithm operates in response to events as follows:

1. On receipt of a new publication requesting replies, a dispatcher creates an empty histogram of reply results for that publication. It records the dispatcher from which it received the publication.
2. When the dispatcher forwards the publication to dispatchers and receivers, the addresses of those elements are stored in a local table. Hence, a dispatcher stores the address of each of its child-nodes in the publication's forwarding tree. These are the elements from which the dispatcher might observe replies.
3. When a reply is received by a dispatcher, the dispatcher records that the child node in question has replied. Then the reply is stored in the local histogram created in step 1. This is done by merging the reply into the histogram. Any reply received by the dispatcher will already be a histogram. (In the case of a reply sent from a receiver this histogram contains only a single entry and a count of one on that entry.) Hence, the reply merging operation is always between the locally stored accumulated histogram and a received reply histogram.
4. When a dispatcher has received a reply from each node to which it forwarded the publication (the last outstanding child node sends a reply as handled by step 3), the dispatcher sends the resulting merged reply histogram to its parent in the forwarding tree. It indicates to the parent that this is its final report of reply data. It then deletes is local state regarding the publication and its associated replies. Alternatively, it performs these actions if it receives a terminate message from its parent in the forwarding tree. It does so after sending the terminate message further along the tree and waiting a specified duration for final reply results from child nodes in the forwarding tree.
5. A dispatcher may receive a request for its collected histogram before it has received replies from all dispatchers that are child nodes from which it awaits replies. In this case it forwards the request to its children, waits some time duration specified in the request, and then forwards its current accumulated histogram state to its parent node. Additionally each responding child node sends the dispatcher a lower bound on the number of receivers from its sub-tree that have not yet replied to the publication. The dispatcher adds to this number the number of its own child nodes that did not respond to the request for reply information. It forwards this information to its parent node along with the merged reply histogram. The dispatcher does not delete its local state.
6. The publisher may query reply harvest results. This results in the propagation of query messages as discussed in step 5. This precipitates reporting of intermediate results to the publisher's histogram view.
7. The publisher may terminate a reply view. This results in the propagation of terminate messages and the resulting final merging of reply results as discussed in step 4.

In the above algorithm there are three methods by which a publisher's reply view is updated. If a dispatcher receives

all possible reply information from its children dispatchers in a forwarding tree, then it automatically sends final reply information to its parent in the forwarding tree. A dispatcher can also be asked to accumulate current or final reply results. Therefore if all receivers reply to a publication, and all reply messages are received, then reply storage reduces to storage at the dispatcher local to the publisher. It is unlikely that this will occur in a large network, regardless of application policy. The current and final view request capabilities accommodate this general case.

Dispatcher-Stateful-Reply stores information at dispatchers throughout forwarding trees. It obtains potential optimizations in message traffic in return for this storage overhead. Additionally, it can calculate a lower bound on outstanding replies at no extra message cost. We will consider the costs of this algorithm in Section 4.

3.2.2 Message-Stateful-Reply

Message-Stateful-Reply operates without storing publication and reply information at all dispatchers in a forwarding tree. Instead, it only stores state in the dispatchers at the root and leaves of a forwarding tree. All other state is stored in publication and reply messages. Message-Stateful-Reply works as follows:

1. When a publisher emits a publication, the publisher's local dispatcher creates and stores an empty histogram of replies.
2. When a dispatcher forwards a publication to other dispatchers, its appends its address to a forwarding path stored in the internal representation of the message.
3. When a leaf node (receiver) obtains a publication, the receiver's local dispatcher (the receiver's parent in the forwarding tree) stores the reply path in a local table indexed by publication identifier.
4. When a reply is issued by a receiver application, it must be issued to the dispatcher from which it received the publication. The reply is issued in association with the publication's identifier. This identifier is used by the dispatcher to look up the reply path in its local state table. The path is attached to the reply message. The address of the parent forwarding node is removed from the path and the reply is forwarded to that dispatcher.
5. When a dispatcher at an intermediate position in a forwarding tree (as indicated by remaining path elements in the message) receives a reply message, it finds the parent node in the path data of the reply, removes it from the reply data, and returns the reply to that node.
6. When a reply is received by the root of a forwarding tree (indicated by a null remaining path element), it has been received by the dispatcher local to the publisher (the root of the forwarding tree.) The reply message is merged into the publication's histogram. If no such histogram exists, the message is discarded.
7. The publisher may request an updated view of the collected histogram at any time. The publisher may also terminate the reply view at any time.

Message-Stateful-Reply applies path-storage in publications and replies to reduce the state storage required in dispatchers (as compared to Dispatcher-Stateful-Reply.) A dispatcher uses a message's stored path information to reverse-forward replies along a path in a forwarding tree. Thus it maintains the distribution of replies over the paths of the publication's forwarding tree. Note that it cannot apply merging algorithms at intermediate nodes because the coordination of reply events at common intermediate points on the return path is not determined or enforced. We will consider the costs of this algorithm in the next section.

3.2.3 Immediately Implied Algorithm Variations

Message-Stateful-Reply and Dispatcher-Stateful-Reply are two points in the potential algorithm design space representing variations in state-storage. It is possible to produce a Reply algorithm with combinations of properties from these two algorithms. A third algorithm could be produced with intermediate storage and reply capabilities by altering dispatcher and message storage as well as by introducing additional messaging. Additionally, other techniques for reply harvesting can be employed such as periodic reply-view updates.

Furthermore, the heterogeneous application of Message-Stateful-Reply and Dispatcher-Stateful-Reply dispatchers in the same dispatcher network is possible with minimal adjustments to the above algorithms. A system might maintain 'stateful' dispatchers at dedicated Publish/Subscribe service providers and 'stateless' dispatchers at client hosts or low-level routers in a Publish/Subscribe system.

4. Algorithm Analysis

The costs and performance of our Reply algorithms depend on their storage and communication costs. Message-Stateful-Reply incurs storage costs in publications, replies, and the root and leaf nodes of forwarding trees. It incurs a potential message event cost of one reply message traveling from each leaf of a forwarding tree to its root. Dispatcher-Stateful-Reply requires storage at all dispatchers in a forwarding tree. Its reply message event cost is significantly less than that of Message-Stateful-Reply. Its total message data cost is the same as Message-Stateful-Reply's under some circumstances but is significantly better under bounding type-enforced constraints on reply values.

Our analysis of Message-State and Dispatcher-State Reply depends on the characterization of a worst-case publication. We apply the same characterization to the analysis of both algorithms. For our analysis, worst-case is the worst-case cost in storage and message traffic for a publication. In this section we will not be considering worst-case cost for multiple publications, such as extended and overlapping reply view lifetimes.

Characteristics effecting a worst-case publication are a large forwarding tree in which all receivers are at great depths in the tree, and where all receivers respond with a wide range of replies in a short time interval.

For the purpose of clarity in our analysis, worst-case forwarding trees will be Reply-optimal. By this we mean that although the above characteristics describe our worst-case tree, it will have features, such as constant branching factor, that distribute the paths of reply messages well. We will consider in a later section those trees that are non-Reply-optimal.

Let the forwarding tree consist of a set of dispatchers, **D**, and a set of receivers, **R**. Let the set of dispatchers and receivers form a full and complete tree with constant branching factor **b**. Require that the leaves of the tree are only elements from set **R**, and the non-leaves of the tree are only elements from set **D**. Hence, a worst-case, Reply-optimal forwarding tree is a tree of constant branching factor where all receivers are leaves of the tree at a depth of the tree $\mathbf{\log_b(|R|)}$. Given these elements, we parameterize a worst-case, Reply-optimal, publication with:

1. the branching factor, **b**, of the forwarding tree for a published message,
2. the range of valid replies, **P**, in the reply data type for a message,
3. the set of receiver nodes available in the Publish/Subscribe network, **R**, and
4. the depth, **d**, of a forwarding node in a worst-case forwarding tree.

The resulting costs of Message-State and Dispatcher-State Reply are presented in Table 1. We derive these results below.

4.1 Message-Stateful-Reply Cost

Message-Stateful-Reply stores state at the root and leaf nodes of a forwarding tree and within publication and reply messages. The state stored per publication at a receiver node (leaf-node of a forwarding tree) is the publication's forwarding path from the root (publisher) to the leaf (receiver) in the tree. This is of length **O(log|R|)** for a worst case-publication. The state stored at the publisher is a histogram of received replies. A histogram will never store more than one entry for each unique data value from set **P**. Hence the size of the histogram is **O(min(|P|, |R|)),** where **min** is a function choosing the minimum value of its arguments. No state is stored at intermediate nodes of a forwarding tree.

The length of publications and replies varies in Message-Stateful-Reply with the length of the forwarding path attached to publication and reply messages. It is of order **O(d)**.

Each node of the forwarding tree will receive all replies that must pass through it to reach the root of the tree. Hence, the number of message events received by a forwarding node is $\mathbf{O(|R|b^{-d})}$. A leaf node will send one reply event, namely, the reply passed from the receiver application element to its local dispatcher. The publisher's local dispatcher (the root of the forwarding tree) will observe **O(|R|)** replies if all recipients reply.

The total reply traffic, in bytes, sent by a dispatcher to its parent in the a worst-case forwarding tree is of the order of $\mathbf{O(|R|(d)b^{-d})}$. Recall that it receives $\mathbf{O(|R|b^{-d})}$ events and each is of length **O(d)**. Note that at the publisher this cost is more precisely bound as **O(|R|)**.

In summary, the storage cost of stateless Reply is nil for the intermediate forwarding network while the message traffic cost scales linearly with receiver set size. In the worst case of publication to the entire network periphery, reply traffic scales linearly with network size.

4.2 Dispatcher-Stateful-Reply Cost

Dispatcher-Stateful-Reply stores publication and reply state at all dispatchers in a publication's forwarding tree. The purpose behind this state storage is to apply a merging function over reply values at each dispatcher. This merging function reduces generated reply traffic.

Dispatcher-Stateful-Reply does not store path information in publications or replies. Hence publication length is **O(1)** with respect to our parameters. Each reply message sent in Dispatcher-Stateful-Reply is a histogram of reply values. The reply sent by receivers is a single reply data value. The replies received by the publisher (root node) are reply histograms for a **b**-way partitioning of the replies from receiver set **R**. As discussed in Section 4.1, the size of a histogram cannot exceed the size of the data value set from which entries consist. Hence the size of a reply message is of order $\mathbf{O(min(|P|, |R|b^{-d}))}$. The size of histogram data stored at any given forwarding node is also of the order $\mathbf{O(min(|P|, |R|b^{-d}))}$.

Dispatcher-Stateful-Reply receives **O(b)** reply messages at each internal forwarding node of a forwarding tree. Therefore the total bytes received in reply message at a dispatcher at depth **d** is $\mathbf{O(min(|P|b, |R|b^{-d}))}$.

	Publication Length	Reply Length in bytes	Reply Event Count	Total Reply Sending Cost in bytes	Histogram Storage Cost (in entries)
Message-Stateful-Reply	O(d)	O(d)	$O(\|R\|b^{-d})$	$O(\|R\|(d)b^{-d})$ $O(\|R\|)$ at publisher	$O(\log\|R\|)$ at receivers $O(\min(\|P\|,\|R\|))$ at publisher O(0) elsewhere
Dispatcher-Stateful-Reply |P|>|R|	O(1)	$O(\|R\|b^{-d})$	O(b)	$O(\|R\|b^{-d})$	$O(\|R\|b^{-d})$
Dispatcher-Stateful-Reply |P|<=|R|	O(1)	$O(\min(\|P\|, \|R\|b^{-d}))$	O(b)	$O(\min(\|P\|b, \|R\|b^{-d}))$	$O(\min(\|P\|,\|R\|b^{-d}))$

TABLE 1. Cost of Reply at a node at depth d in a worst-case publication's forwarding tree.

4.3 Analysis of Costs vs. Capabilities

Table 1 summarizes the costs of Dispatcher-Stateful and Message-Stateful Reply for worst-case publications. In particular, reply message sending cost scales linearly with network size (**|R|** scales with network size) for both algorithms. Additionally, the cost grows exponentially greater towards the root of a forwarding tree. At the leaves of the tree, message sending cost is constant. At the publisher, it is **O(|R|)**. The only distinction in total message cost between the two algorithms is the factor of **d** in Message-Stateful Reply due to storage of state within messages.

The cost in asynchronous message events sent or received by a dispatcher is significantly different for the two algorithms. It is constrained to **b** (the branching factor of the forwarding tree) for Dispatcher-Stateful Reply. It scales linearly with network size and decays exponentially with depth in the forwarding tree for Message-Stateful Reply. Branching factor is largely constrained in practice and hence is relatively constant (we discuss this further in Section 4.5.) As a result, the asynchronous event cost of Dispatcher-Stateful Reply is significantly reduced in a forwarding tree's interior as compared to application of Message-Stateful Reply. Furthermore, when **P** contains fewer elements than **R** the total message cost for Dispatcher-Stateful-Reply can be significantly reduced as compared to that of Message-Stateful-Reply.

Finally, the storage cost of Message-Stateful-Reply is significantly less than storage cost in Dispatcher-Stateful-Reply at dispatchers acting as internal nodes in a forwarding tree. Message-Stateful-Reply requires no such storage. Dispatcher-Stateful-Reply's storage is linear with network size and decays exponentially with depth in the forwarding tree. However, if the size of set **P** is less than the size of **R**, then the storage cost is constrained linearly by **|P|** rather than by **|R|**.

To summarize, examination of the two algorithms indicates that the relative merits of the two approaches depends on (1) the acceptability of storage at arbitrary forwarding nodes and (2) the relative size of the reply data type range, **|P|** as compared to the size of the network's receiver set, **|R|**, and (3) the benefit of obtaining a lower bound on the number of outstanding replies. In application-level systems, where dispatcher state is affordable and acceptable and where reply value ranges are small (**|P|** is small), Dispatcher-Stateful reply can offer significant performance improvements. Where Publish/Subscribe with Reply is applied at the system-level or where additional dispatcher state is not acceptable, Message-Stateful-Reply might be more appropriate. Further, we have previously mentioned that combinations and variations of these approaches can be employed.

4.4 Average Storage and Bandwidth Cost for Dispatcher-Stateful-Reply

Resource modeling benefits from knowing the average of algorithm cost over forwarding trees. In particular, this is important for calculation from steady-state mean expected resource usage. The following average calculation holds wherever a cost at a node of a forwarding tree is $\mathbf{O(|R|b^{-d})}$: Approximate a worst-case forwarding tree and note its height as **h** ($\mathbf{|R|=b^h}$.) If the cost at a node is $\mathbf{O(|R|b^{-d})}$ then with the definition of **h** it is equivalently $\mathbf{O(b^{h-d})}$.

Each layer of the tree has $\mathbf{b^d}$ nodes. Therefore we calculate the average cost at nodes of the tree by

$$\frac{\sum_{d=1}^{d=h} b^d b^{h-d}}{\sum_{d=1}^{d=h} b^d} = \frac{hb^h}{\frac{b(b^h-1)}{b-1}} = \frac{h(b^h-b^{h-1})}{(b^h-1)} \approx O(h)$$

from which we have derived that average cost scales with the height of the tree. This is **O(log|R|)**. We apply this result in our resource model. By similar calculation, if a cost at a node is $\mathbf{O(|R|db^{-d})}$ then cost averaged over all nodes of a forwarding tree is $\mathbf{O(h^2)}$ or $\mathbf{O((\log(|R|)^2)}$. Hence, the average total reply messaging cost over nodes of a forwarding tree is **O(log(|R|))** for Dispatcher-Stateful-Reply with **|P|>|R|** and is $\mathbf{O((\log(|R|))^2)}$ for and the average messaging cost of Message-Stateful-Reply.

4.5 Resource Allocation Model for Reply

If Reply is to be effective its messaging costs must be considered, in particular relative to those of Publish/Subscribe. Additionally, Dispatcher-Stateful-Reply must consider that storage requirements of forwarding nodes within a Publish/Subscribe with Reply network.

Current Publish/Subscribe systems are generally constructed at the application level. These systems transmit messages using the network-level protocols provided by operating systems. TCP/IP messaging is a typical example.

Our previous cost analyses allow us to specify the bandwidth requirements of links between distributed dispatchers in a deployed Publish/Subscribe with Reply system. We can specify these resource requirements to a network-level service as anticipated mean and maximum resource usage. Additionally, we can reserve the required memory and computational resources required for histogram storage and processing. Hence, static resource reservation should be sufficient to fulfill the resource availability requirements of a Publish/Subscribe with Reply network experiencing expected traffic under optimal operating and network conditions.

Let our network expect to support **n** simultaneous publications expecting replies. Let each publisher have at most **m** of the **n** simultaneous publications. Let a reply data type with value range **P**. Let **R** be the set of receivers in the network. Assert relatively equal distribution of receivers throughout the dispatcher network. Then the message traffic of Dispatcher-Stateful-Reply can be modeled as

- average worst-case traffic
  O(n min(log(|P|), log(|R|))), and
- peak worst-case traffic
  O(m min(|P|, |R|)).

In addition, the histogram storage cost of Dispatcher-Stateful-Reply can be modeled as

- average worst-case storage
  O(n min(log(|P|), log(|R|))), and
- peak worst-case storage
  O(m min(|P|, |R|)).

For the same parameterized conditions, the message traffic of Message-Stateful-Reply is

- average worst-case traffic
  $\mathbf{O(n\ min(\log(|P|)^2, \log(|R|)^2))}$, and
- peak worst-case traffic
  O(m min(|P|, |R|)).

and its storage model is

- average worst-case storage
  O(n log(|R|)), and
- peak worst-case storage
  O(m min(|P|,|R|)).

The resource model above assumes steady-state behavior of the system in which mean and maximum usage statistics are sufficient to provide static reservation or allocation of required resources.

In practice, forwarding trees are not optimal for Reply. That is, they are not full and complete and do not have a constant branching factor. Our analysis and the resulting resource model assumed a relatively constant branching factor in forwarding trees. A forwarding tree without a constant branching factor might worse-than-expected local and regional cost and performance.

Assume that the bandwidth and computational resources of a Publish/Subscribe with Reply network have been allocated with the assumption of a Reply-optimal forwarding tree **T**. That is, assume that the forwarding tree is full and complete with a constant branching factor. Given resource allocation assuming **T**, there are two classes of non-Reply-optimal forwarding trees:

- ***Reply-performance Preserving trees***: A forwarding tree is Performance Preserving if and only if it consists of a subset of the nodes and edges of a Reply-optimal tree of equivalent dimensions to **T**.
- ***Reply-performance Degrading trees***: A forwarding tree is Performance Degrading if it does not consist of a subset of the nodes and edges of a Reply-optimal forwarding tree of equivalent dimensions to **T**.

The first case preserves the anticipated performance of Reply because it utilizes less bandwidth and computational resources than allocated with the assumption of Reply-optimal tree **T**. An example of a Reply-performance Preserving tree is depicted in Figure 3. A tree with constant branching factor 3 is shown as a set of nodes connected by edges. The Reply-optimal tree assumed in resource alloca-

tion was of branching factor 3. Reply-performance Preserving trees utilize a subset of Reply-optimal links between forwarding nodes. They perform fewer merging operations at any given nodes than full use of a tree **T**, and additionally send histograms of smaller than maximum possible size observed on **T**. Figure 3 shows a Reply-performance preserving tree as a set of shaded nodes and thicker edges, covered by the Reply-optimal tree.

The class of Performance Degrading trees use more bandwidth than assumed by tree **T** under which resource allocation is modeled. The forwarding tree contains at least one forwarding node **N** at depth **d** such that **N** has more descendants than a node in **T** at depth **d**. In such a tree, the bandwidth requirements from **N** to the root of the tree will be greater than anticipated by a Reply-optimal tree. An example of a Reply-performance Degrading tree is depicted in Figure 4. In the example, a Reply-optimal tree with branching factor 3 is assumed for **T**. The graph of Figure 4 cannot be covered by a tree with branching factor 3. Consider the implications for very large trees. Let a node **N**, at depth **d**, have 1,000,000 descendants where **T** assumed 5,000 descendants at depth **d**. The resulting bandwidth requirements from **N** to the root of the tree-to transport the harvested 1,000,000 replies-will considerably overload the allocated bandwidth from **N** to the root node.

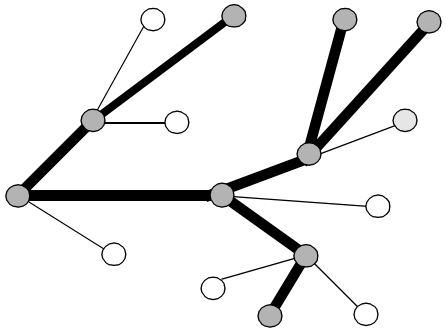

FIGURE 3. A Reply-performance preserving tree.

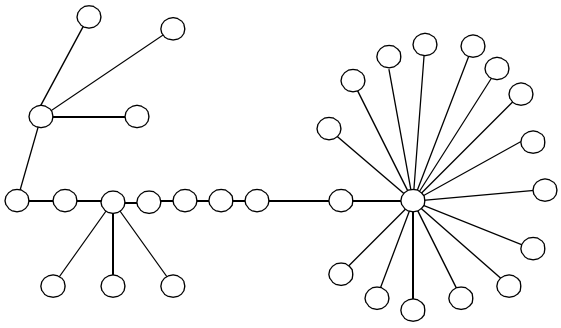

FIGURE 4. A Reply-performance Degrading tree where branching factor was assumed to be at most 3 during network resource allocation.

Allocating bandwidth to account for non-optimal forwarding trees is an important consideration in performance in real application networks. While performance degradation near the leaves of a tree will not greatly impact performance, variance from anticipated branching factor deep within the interior of a tree may have a significant effect. Handling performance degradation through avoidance of irregular forwarding trees or compensation for their bandwidth requirements is necessary. A solution strategy will largely depend on the type of distributed Publish/Subscribe application-level network architecture in use, and the resource-reservation capabilities of the network-level architecture on which it resides.

4.5.1 Hierarchical Dispatch Architectures

Hierarchical Publish/Subscribe Dispatch Architectures easily support Publish/Subscribe with Reply. They can guarantee that all non-Reply-optimal trees are Performance Preserving.

Subnet-based distributed Publish/Subscribe architectures consist of a graph of dispatchers. Each node of the graph is a dispatcher that may forward published messages to other dispatchers and receivers to which it is connected.

The connectivity of a *Hierarchical* Dispatch Architecture always forms a tree of dispatcher nodes. Each dispatcher node of the tree has a fixed set of connections to identifiable child nodes. All nodes except the root have a single, identifiable parent node.

Any forwarding tree generated within a hierarchical dispatcher network is constrained to a subset of the dispatcher network graph's nodes and edges. If we allocate bandwidth resources assuming a tree **T**, then conformance of the hierarchical dispatcher tree to dimensions equivalent to **T** guarantees that all forwarding trees will be covered by a tree of dimension **T**. Hence, all possible forwarding trees will be Performance Preserving.

4.5.2 Peer-to-Peer Dispatch Architectures

Peer-to-peer ('mesh') Publish/Subscribe can create Reply-Performance Degrading forwarding trees. In a *peer-to-peer* network, there is no explicit parent of any given dispatcher node. There are two methods that can be employed to support resource allocation for Reply:

- *Homogenize topology while holding resource requirements constant.*
  'Mesh'-like peer-to-peer connection topologies, with constraints on peer connectivity work well with Reply. Let any dispatcher node of the Publish/Subscribe dispatcher network have between **x** and **y** connections to other dispatcher nodes. Regardless of receiver-set size,

this topology enforces a local branching factor between **x** and **y** for any node of a generated forwarding tree. If we assume bandwidth reservations with a Reply-optimal tree **T** with branching factor **y**, then any generated forwarding tree will be covered by a tree of dimensions equivalent to those of **T**. Hence, all generated forwarding trees will be Performance Preserving.

- *Vary resource requirements with connective topology.* If topology does not constrain connectivity, then bandwidth must be reserved according to worst-case forwarding trees. The required graph analysis is beyond the scope of our work.

5. Dispatcher-Stateful-Reply Implementation

We have implemented Dispatcher-Stateful-Reply for Publish/Subscribe. Our implementation, called Siena Harvest, is an independent co-application attached to the Siena Publish/Subscribe architecture. Our architecture tests the capabilities of Dispatcher-Stateful-Reply as an independent co-application. It also demonstrates the feasibility and utility of Reply.

5.1 Architecture

We have constructed Dispatcher-Stateful-Reply as a co-application of a Publish/Subscribe system. By co-application, we mean that the architectural elements of Reply work in conjunction with unmodified elements of a Publish/Subscribe system. In particular, this co-application works in conjunction with the forwarding nodes, called Dispatchers, of Publish/Subscribe. It works by creating 'hooks' to observe the message traffic of the Dispatcher elements.

Reply consists of a distributed collection of processes called Harvesters. There is a Harvester associated with each Dispatcher in a Publish/Subscribe network. A Harvester is co-located with its Dispatcher in a local environment, so that a computer hosts a Harvester if and only if it hosts a Dispatcher.

The role of a Dispatcher and Harvester pair are shown in Figure 5. A Dispatcher receives publications and forwards them to other sites. A Harvester receives replies to publications, merges them, and returns the result to a site. As discussed in Section 3, the sites from which messages come and go are determined entirely by the forwarding performed in Dispatchers.

In order to perform its task, a Harvester observes its assigned Dispatcher as a 'black box' from which the input and output are observed. The term 'snooping' refers to this observation. Collecting the local forwarding information of a Dispatcher allows the Harvester to know from where replies will come, and to where the merged histogram must be sent.

When a publication message is local to a Harvester/Dispatcher pair the Harvester and Dispatcher interface directly with the publisher. Likewise, receivers also directly interact with Harvesters. These interactions (boundary conditions of the algorithm) are shown in Figure 6. First, a publication is sent to a publisher's local Dispatcher. At some later time, the publisher requests response information from its local Harvester. This can either be a blocking or non-blocking call. At the same time, various receivers in the network obtain the publication from a Dispatcher. As a result, they may reply to their local Harvesters. Reply information is collected and returned to the publisher's view of the reply harvest.

5.2 Siena Harvest Implementation

We have implemented Siena Harvest, a Dispatcher-Stateful-Reply feature for the Siena [1] Publish/Subscribe architecture. The implementation is a refinement of the architecture discussed in Section 5.1. It implements Reply

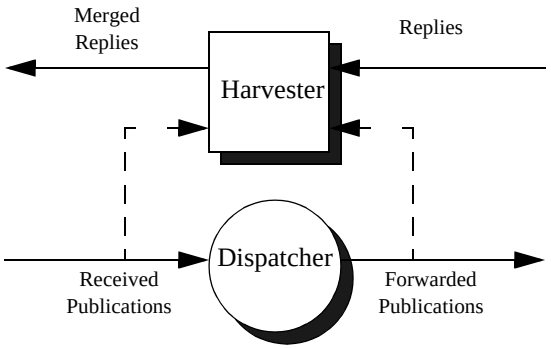

FIGURE 5. The architecture of Reply.

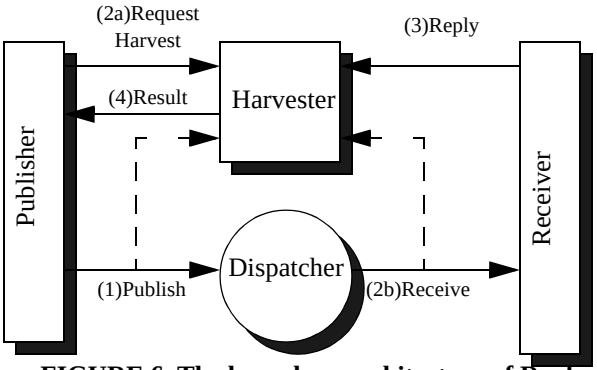

FIGURE 6. The boundary architecture of Reply.

as a set of distributed Harvesters that snoop on publications travelling to and from Dispatchers.

A Harvester's internal function is to respond to reply and publication messages by modifying data structures and emitting merged replies. A Harvester maintains forwarding-tree and reply histogram information in tables. It only deletes entries when publishers terminate a reply histogram view or when all possible information has been reported to a parent node in a forwarding tree.

Siena supports a plug-in message protocol layer. We take advantage of this it to replace existing transport components with identical components that also snoop publications for the Harvester. The resulting components are shown in Figure 7.

The harvester is augmented by a publication Packet_Receiver and Packet_Sender that are directly attached to a Hiearchical_Dispatcher. These components serve as the input and output mechanisms for publication traffic to both the Dispatcher and the Harvester.

5.3 Experience

Siena Harvest works well for distributed, loosely coupled applications. Our experience with Siena Harvest includes its application to a wide-area command-and-control infrastructure in which commands are published and results are sent as replies. We are able to emit commands to a distributed collection of computers at the same rate as in traditional Publish/Subscribe, and harvest replies to determine the effect of our commands.

Utilizing a combination of Reply and Site-Select Messaging [3], we have constructed a form of Selective Request/Response we call Site-Select Command. It is similar to one-to-many RPC with property-based selection of server sites. This system is utilized to direct orders to widely distributed, loosely coupled components based on described qualities of relevant components [8] [2] [3]. It allows us to issue orders by qualitative description of receiver-site conditions. Siena Harvest allows us to observe results from all sites receiving a command.

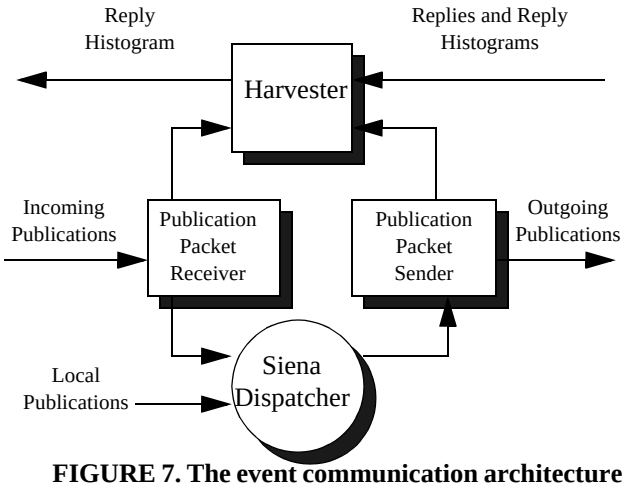

FIGURE 7. The event communication architecture of Siena Harvest within Siena.

The space and time decoupling of publishers and receivers has interesting implications for a publisher observing replies. First, it is desirable to support blocking and non-blocking calls to view reply results. In a typical client-server architecture, a blocking call is an implicit wait at the caller for a response from the callee. A non-blocking call does not wait for a response from a callee to return control flow to the caller.

Publish/Subscribe with Reply supports an initially unknown quantity of callees (receivers) for each caller (publication). Non-blocking 'calls' in Publish/Subscribe with Reply have clear semantics, as they express publication followed by asynchronous receipt of replies. This maintains the flow-independence property of Publish/Subscribe described in [6].

Blocking-call semantics for Publish/Subscribe with Reply are complicated by the attempt to couple the program flow of a publisher to receivers while maintaining the temporal and spatial decoupling between them. As a message is published, the forwarding tree for the message grows until all receivers currently subscribed to the content/type of the message have been forwarded the message and become leaves in the message's forwarding tree. During this time, the view of the reply results will indicate an increasing quantity of message receivers. A Publisher cannot determine a time at which all receivers will obtain the message. It does not know a quantity of potential receivers. Furthermore, it cannot determine when they will reply to the message they have received. Hence, "waiting for all receivers to get the message" or "waiting for all receivers to reply" are not appropriate blocking semantics. Space and time decoupling complicate parameterization of a "blocking wait."

We have not yet obtained a satisfactory solution for parameterizing a blocking wait on replies. Our current implementation applies a fixed time-interval to blocking wait. This is inadequate because it assumes time coupling where none is supported. Another possible parameterization is the rate of change in the number of received replies. Waiting for a rate of change is a useful heuristic, but not a precise or accurate measure of completeness. This is analogous to waiting for a bag of popcorn to pop in a microwave oven [11]. A popcorn bag invariably leaves some kernels un-popped, and can often mislead one to stop waiting too early if there is an anomalous decline in the number of pops in a given time-frame. In Siena Harvest, fluctuations in reply rates may cause us to miss the replies from some receivers while other receivers reply outside an effective

time-frame. In our work, an adequate parameterization for a blocking semantics remains an open question.

A Harvester implementation based on snooping forwarding information has the advantage of being independent of the subnet forwarding algorithm to which it is applied. In fact, the principle applied to harvesting is independent of the associated forwarding algorithm assuming that the algorithm guarantees that forwarding paths are trees. Our implementation snoops message traffic. This has two disadvantages. First, when a publisher requests intermediate harvest results (polls for currently harvested replies) it obtains only a lower bound on potential replies. This is because the Harvester is unable to determine when a Dispatcher is done forwarding a particular publication. Second, examining the messages of a Dispatcher, and reinterpreting their contents creates needless translation overhead. Direct communications between a Dispatcher and Harvester would allow collection of superior information as well as increased performance.

6. Future Work

Our future work includes experiments to determine the load capability of Publish/Subscribe with Reply versus Publish/Subscribe. We will examine the load capability on several hundred distributed components with varying application distribution graphs and usage patterns.

We will also continue to explore semantics for blocking calls on reply harvesting. Our goal will be to establish parameterization that is natural and useful. For example, a linear combination of temporal parameters might be sufficient for many applications. None-the-less it is possible that time decoupling does not permit reasonable blocking semantics for flow coupling.

Our application of the technology will continue in the command-and-control arena. Our work in this area involves control of wide-area software application configurations. Reply provides information about remote command execution to the publisher of a command. What is returned after the issuing of a command is a histogram of results, and this leads to interesting questions for the representation of success, failure, and exception conditions as an interpretation of histogram data.

7. Conclusions

Publish and Subscribe with Reply is an elegant extension of existing, state-of-the-art distributed Publish and Subscribe architectures. Intuitively, Reply works because it follows, in reverse, the forwarding path generated by distributed Publish/Subscribe forwarding algorithms. It is a natural extension of Publish/Subscribe because it does not impose coupling in time or space between components. Furthermore, it maintains the efficiency of data-push applications.

We considered algorithms that do and do not maintain state at intermediate forwarding nodes in a Publish/Subscribe system. A stateless algorithm may be required if intermediate storage throughout a network is not acceptable. However, Dispatcher-Stateful-Reply, an algorithm maintaining intermediate state, was shown to scale with a significant advantage over a stateless algorithm when the range of valid replies is small. The stateless algorithm, Message-Stateful-Reply, was shown to be equally effective when the range of valid replies is large.

Assuming bounded domains for reply-types, Reply scales with network size as well as Publish/Subscribe. For many applications this will be an important bound on reply applications. For applications in which the range of replies is effectively unbounded, extra cost must be assumed and the benefits of stateful reply are not significant.

Averaged over all nodes in a forwarding tree, worst-case cost for a single publication is of the order of the logarithm of network size. The worst case number of asynchronous replies received at any node is of the order of the branching factor of a forwarding tree, and can be constrained by dispatcher connectivity. This allows us to consider the average and peak resource requirements of stateful and stateless reply algorithms.

Our implementation of Dispatcher-Stateful-Reply for Siena demonstrates the effectiveness of the Reply algorithm and has already been useful in an experimental demonstration of distributed command-and-control. Acknowledging receipt of publications, distributed query, and selective command [3] are three examples of loosely coupled, event-driven distributed systems that can apply Publish/Subscribe with Reply. Its efficiency will allow such systems to achieve significant scale.

Publish and Subscribe with Reply provides an important framework for large-scale distributed applications with important roles. It supports a variant of Request/Response-like semantics in a Publish/Subscribe framework. Generally, it allows the establishment of a two-directional communication over the dynamic binding of a publisher to receivers for a given message. This capability greatly enhances the set of loosely-coupled, distributed applications that may benefit from Publish/Subscribe.

8. Acknowledgments

This work has been made possible through contributions to the field of Publish/Subscribe from Antonio Carzaniga, Alex Wolf, and Rosenblum. Interaction and collaboration with these researchers facilitated the applications and insights that led to research in a reply feature for

Publish/Subscribe. Thanks also go to Phil Varnar for introducing the ‘microwave popcorn’ analogy for reply harvesting.

This work has been funded, and thus made possible, by DARPA’s OASIS program.

9. References

[1] Carzaniga, A; Rosenblum, D; Wolf, A. ‘Achieving scalability and expressiveness in an Internet-scale event notification service.’ *Symposium on Principles of Distributed Computing*, 2000. pp 219-227.

[2] Heimbigner, D. ‘Adapting publish/subscribe middleware to achieve Gnutella-like functionality.’ *Selected Areas in Cryptography*, 2001. pp 176-181.

[3] Hill, J. C. ‘Site-Selective Messaging for Distributed Systems.’ University of Virginia Technical Report CS-2002-06.

[4] Cugola, G.; Di Nitto, E.; Fuggetta, A. ‘The JEDI event-based infrastructure and its application to the development of the OPSS WFMS.’ *IEEE Transactions on Software Engineering*, Volume: 27 Issue: 9, Sept. 2001. Page(s): 827 -850

[5] Rowstron, A; Kermarrwc, A; Castro, M; Druschel, P. ‘The Design of a Large-Scale Event Notification Infrastructure.’ *Networked Group Communication*, 2001. pp 30-43.

[6] Eugster, P; Felber, P; Guerraoui, R; Kermarrec, A. ‘The Many Faces of Publish/Subscribe.’ Technical Report DSC ID # unavailable

[7] Knight, John C., Dennis Heimbigner, Alexander Wolf, Antonio Carzaniga, Jonathan Hill, Premkumar Devanbu, Michael Gertz. ‘The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications.’ June 2002.

[8] Gerkey, Brian P. Mataric, Maja J. ‘Murdoch.’ *Proceedings of the Fourth International Conference on Automated Agents*. June 2000.

[9] P. Sjoberg. ‘The Java Message Service 1.0.2.’

[10] Cybenko, G. Brewington, B. ‘The foundations of information push and pull.’ *Mathematics of Information*. Springer--Verlag, 1998.

[11] Personal discussion with Phil Varner, Department of Computer Science, University of Virginia.

[12] B. Krishnamachari, D. Estrin, S. Wicker. ‘The Impact of Data Aggregation in Wireless Sensor Networks.’

[13] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, D.Ganesan. ‘Building Efficient Wireless Sensor Networks with Low-Level Naming.’ *Symposium on Operating System Principles*, 2001. pp 146-159