Fault-Tolerant Reliable Delivery of Messages in Distributed Publish/Subscribe Systems

Conference Paper · July 2007

DOI: 10.1109/ICAC.2007.18 · Source: IEEE Xplore

3 authors, including:

Hasan Bulut
Ege University
49 PUBLICATIONS **1,964** CITATIONS

SEE PROFILE

Geoffrey Charles Fox
University of Virginia
1,507 PUBLICATIONS **28,817** CITATIONS

SEE PROFILE

Fault-Tolerant Reliable Delivery of Messages in Distributed Publish/Subscribe Systems

Shrideep Pallickara, Hasan Bulut and Geoffrey Fox
(spallick, hbulut, gcf)@indiana.edu
Community Grids Lab, Indiana University

Abstract

Reliable delivery of messages is an important problem that needs to be addressed in distributed systems. In this paper we present our strategy to enable reliable delivery of messages in the presence of link and node failures. This is facilitated by a specialized repository node. We then present our strategy to make this scheme even more failure resilient, by incorporating support for repository redundancy. Each repository functions autonomously. The scheme enables updates to the redundancy scheme depending on the failure resiliency requirements. If there are ***N*** *available repositories, reliable delivery guarantees will be met even if* ***N-1*** *repositories fail.*

Keywords: Publish/subscribe middleware, reliable delivery, fault tolerance, redundancy.

1. Introduction

Increasingly interactions that services, resources and components have with each other are network-bound. In several cases these interactions can be encapsulated in messages. These messages can encapsulate, among other things, information pertaining to transactions, data interchange, system conditions and finally the search, discovery and subsequent sharing of resources. Messaging is a critical component in distributed systems. Here, the messaging infrastructure can be based on several different paradigms viz. publish/subscribe systems, queuing systems, and peer-to-peer systems among others. Our work focuses on systems based on the publish/subscribe paradigm.

In the publish/subscribe paradigm clients have a clear decoupling of the message publisher and subscriber roles. The routing of messages from the publisher to the subscriber is within the purview of the message oriented middleware (MOM), which is responsible for routing the right content from the publisher to the right subscribers. A subscriber registers its interest in messages by subscribing to *topics*. In its simplest form these topics are typically "/" separated Strings; complex forms can be based on XML. When a publisher issues messages on a specific topic the middleware substrate – comprising a set of distributed broker nodes – routes these messages to the subscribers that have registered an interest in this topic.

In this paper we present a scheme for the reliable delivery of messages issued over a topic in publish/subscribe systems. Topics over which authorized publishers and subscribers can have reliable communications are referred to as reliable-topics. The scheme outlined in this paper facilitates the reliable delivery of messages from the publishers to the subscribers in the presence of node and link failures. The communication links within the system could also be unpredictable, with messages being lost, duplicated or re-ordered in transit over them, en route to the final destinations. Finally, subscribers are able to retrieve messages issued over the reliable-topic during the subscriber's absence (either due to failures or intentional disconnects). We also extend the basic reliable delivery scheme for greater redundancy and fault-tolerance.

Preliminary ideas of the basic reliable delivery scheme were displayed in the poster [1] session of the 2004 IEEE Autonomic Computing Conference. This basic scheme has been extended for efficient processing of acknowledgements and error corrections to better cope with missed messages. We have also incorporated support for multiple repositories, which provides greater redundancy and fault tolerance.

2. NaradaBrokering Overview

We have implemented the scheme described in this paper in the context of the NaradaBrokering substrate [2], which is based on the publish/subscribe paradigm. In NaradaBrokering this MOM is itself a distributed infrastructure, comprising a set of cooperating router nodes known as *broker*s. A broker performs the routing function by routing content along to other brokers within the broker network. Entities are connected to one of the brokers within the broker network, an entity uses this broker, which it is connected to, to funnel messages to the broker network and from thereon to other registered consumers of that message.

2.1 The Topic Discovery Scheme

Interactions between entities in publish/subscribe systems are predicated on the knowledge of the topic that will be used for communications; the publisher will publish over this topic while the subscriber registers a subscription to this topic. The topic discovery and creation scheme (details can be found in Ref [3]) in NaradaBrokering facilitates the creation, advertisement and authorized discovery of topics by entities within the system. When an entity creates a topic, that entity is also the topic owner. This ownership can be cryptographically, and deterministically, verified based on the contents of the topic advertisement that is generated as part of the topic creation process. The discovery process is a distributed process and is resilient to failures that might take place within the system. Topic creators can advertise their

topics and can also enforce constraints related to the discovery of these topics. Specifically, a topic creator may require the presentation of appropriate credentials (a X.501 security certificate) prior to being able to discover a topic. This scheme provides a solution for issues such as

1. Provenance – The system can verify easily the owner of a certain topic.
2. Secure discovery –- A topic owner can restrict the discovery of a topic only to authorized entities or those that possess the valid credentials.

These capabilities are provided by specialized nodes – Topic Discovery Nodes (TDNs) – within the system. Since a given topic advertisement will be stored at multiple TDN nodes, this scheme sustains the loss of TDN nodes due to failures or downtimes.

3. Reliable Delivery of Messages

The scheme for reliable delivery of messages, issued over a reliable-topic, needs to facilitate error corrections, retransmissions and recovery from failures. In our system, a specialized *repository* node which *manages* this reliable-topic plays a crucial role in facilitating this. The repository facilitates reliable delivery from multiple publishers to multiple subscribers over its set of managed reliable-topics. The only requirement for the basic reliable delivery scheme is that if a repository fails, it should recover within a finite amount of time. There can be multiple repositories within the system and a given repository may manage multiple reliable-topics, however (for the purposes of discussion in this section) a given reliable-topic can only be managed by exactly one repository. Section 4 describes a scheme where there can be multiple repositories for a given reliable-topic; this scheme can sustain repository failures.

Management of reliable-topics involves two key components. First, the repository should facilitate the registration (and de-registration) of authorized entities for reliable communications over the reliable-topic.

Second, to support error-corrections, retransmissions, and recovery from failures (including those of the repository itself) a repository also needs to provision a *persistent storage* (this function is typically provided by a database. We have also implemented support for flat-files) so that messages and other information pertinent to the reliable delivery algorithm can be stored. For a reliable-topic managed by a repository, this repository stores messages issued over this reliable-topic by any of the authorized publishers. This persistent storage of messages facilitates subsequent retrievals should the need arise.

Reliable delivery of messages involves two key components. The first one involves ensuring that messages published by the publisher, over a reliable-topic, are stored exactly-once, without gaps and in-order at the repository managing this reliable-topic. Second, for every such stored message, the repository also has to compute the intended destinations and ensure the reliable delivery of the stored message to the computed destinations.

3.1 Control-Events

The reliable delivery algorithm involves communications between various entities through the exchange of *control- events*, (summarized in Figure 1 on page 4) where the term events is used to distinguish it clearly from messages published over reliable-topics. The control-events (simply events, for brevity, hereafter) relate to intermediate steps to facilitate reliable delivery, acknowledgements, error-corrections, retransmissions and recovery related operations. Our notation for events identifies the source, the destination(s) and the type of the control-event: `Source2Destination-ControlType`. For purposes of brevity, we use only the starting alphabets of the entities involved in the exchange. Thus, an acknowledgement issued by the repository to the publisher is represented as `R2P-ACK`. The destination part is in **`bold-face`** if there are multiple destinations.

3.2 Communications between entities

Entities (repository and the clients) communicate with each other through exchanges issued over a topic. Entities may restrict the discovery [2] of the *communication-topic* to a set of entities or to those that present valid credentials: this allows an entity to restrict the list of entities that can communicate with it.

3.3 Registering a reliable-topic

As mentioned earlier, there can be multiple repositories within the system, and a given repository may manage more than one reliable-topic at a time. The first step to facilitating reliable communications over a reliable-topic is the registration of the reliable topic at a repository. To do this the owner of the reliable-topic locates a repository willing to manage the reliable-topic. Once a repository has been located, the owner then proceeds to register clients that are authorized for reliable delivery over the reliable-topic. Publishers and subscribers that are not explicitly authorized (and registered) by the reliable-topic owner cannot avail of reliable communications over that topic.

Prior to reliable communications over a reliable-topic a client needs to locate the corresponding reliable-topic repository by discovering the communication-topic associated with the repository.

3.4 Publishing Messages

One of the prerequisites for reliable delivery is that messages published by a publisher be stored reliably, and in-order, at the corresponding reliable-topic repository. To ensure this the publisher maintains a local buffer for

temporarily storing published messages. The publisher and the repository also exchange events to ensure that published messages are received correctly at the repository.

For a given reliable-topic a repository will specify the weakest subscription constraint for that topic. This ensures that most, if not all, messages issued by publishers to the topic will be also received at the repositories. To ensure that a repository can know about, and retrieve, missed messages for every published message the publisher also issues a P2R-Order event to the reliable-topic repository.

A publisher stores every message that it publishes, over a reliable-topic, in its local buffer (maintained in memory), which serves as a temporary storage. The P2R-Order event issued in tandem with the published message contains a monotonically increasing catenation number and the message-identifier of the published message that it correlates to. The catenation numbers contained within the P2R-Order event allows the reliable-topic repository to determine the order in which these messages were generated and to determine if messages were lost in transit.

3.5 Repository processing of published message

Upon receipt of a message (issued over one of its managed reliable-topics) the repository queues the message in a temporary buffer, this message is not acted upon until the corresponding P2R-Order event is received.

When a P2R-Order event is received a check is made to see if the corresponding published message is received: this is done by checking the temporary repository buffer to see if a message with the identifier contained in the event has been received. If the published message has been received an acknowledgement – R2P-ACK event – is issued back to the publisher. The R2P-ACK event encapsulates two types of acknowledgements: first type covers specific catenation numbers; the second type simply includes one catenation number signifying that all messages up until that catenation number have been received. If the published message corresponding to the P2R-Order event is missing, the repository issues a negative acknowledgement – R2P-NAK event –- to the publisher to retrieve the missing message.

The repository then checks to see if this event has been previously received at the repository: this is done by checking the publisher/catenation number pair within the P2R-Order event. If a message correlated with this duplicate event is in the repository's temporary buffer, that message is also discarded as a duplicate.

Finally, the repository also checks to see if there are any gaps in the P2R-Order events received from the publisher. Since every P2R-Order event contains a monotonically increasing catenation number this is easy to do.

Once a message has is identified as not being a duplicate message, the repository is ready to remove this message from its temporary buffer, and store it onto the underlying persistent storage. The repository assigns a monotonically increasing *sequence number* for every message that it stores to persistent storage. When a repository receives an event identifying missing messages the repository sets aside sequence numbers for these missing messages and issues a R2P-NAK event to retrieve them. Messages other than these missed messages are stored onto the persistent storage based on the advanced sequence number.

3.6 Processing repository acknowledgements

A positive acknowledgement R2P-ACK event signifies successful receipt of the message and the corresponding P2R-Order event at the repository. The local buffer entry corresponding to this message can then be removed. A negative acknowledgement R2P-NAK event signifies that the message corresponding to a specific catenation number was lost in transit to the repository. This lost message should be retransmitted from the repository.

Upon receipt of a positive acknowledgement R2P-ACK event from the repository the publisher processes the event to delete appropriate entries from its local buffer. If the acknowledgement is a stand-alone acknowledgement, entries corresponding to the acknowledgement catenation numbers are removed from the local buffer. If the R2P-ACK event encapsulates an encompassing acknowledgement, all entries up until that catenation number included in the event are erased from the local buffer.

Upon receipt of the negative acknowledgement R2P-NAK event the message(s) corresponding to the specified catenation number(s) are retrieved and prepared for retransmission. The retransmission occurs in the P2R-Retransmit event which contains both the original published message along with the catenation number for the message. Similar, to the P2R-Order event, the P2R-Retransmit event is received only by the repository.

3.7 Persistence notifications

Upon successful receipt of a published message at the repository, in addition to the operations (which includes storing the message to persistent storage) outlined in section 3.5 the repository performs three additional functions. First, depending on the topic type contained in the original published message the repository loads the appropriate matching engine to compute destinations for the published message based on the registered subscriptions.

Second, the repository adds an entry to the dissemination table that it maintains. For a given sequence number, the dissemination table enables a repository to keep track of destinations that have not explicitly acknowledged the receipt of the corresponding

published message. The dissemination table is continually updated to reflect the successful delivery of the published message to the intended destinations. The dissemination table thus allows us to determine holes in sequences for messages that should have been delivered to a client.

Finally, the repository issues an event signifying the persistence of the published message. If **S** is the set of registered subscribers to a given reliable-topic, and if **S*** is the subset of subscribers whose subscription constraints are satisfied by the published message, then the R2**S***-Persistent event signifies that it would be received only by that subset of subscribers. To ensure this, the repository ensures that the topic information for the R2**S***-Persistent event is the same as that of the original published message.

The R2**S***-Persistent event contains the sequence number assigned to the published message and also the identifier associated with the published message. A subscriber can then correlate a published message and its persistence event.

3.8 Processing persistence events at the subscriber

A subscriber to a reliable-topic receives published messages from the publishers, and events from the repository. Upon receipt of a message from a publisher, a subscriber stores this message in its temporary local buffer. In our reliable delivery scheme we rely on the *all-or-none* model: here, a message is either delivered to all the subscribers or it is not delivered to any of the subscribers. To enforce this, a subscriber releases a message only after it has been confirmed to be *stable.* A message is considered stable, only if both the message and the corresponding R2**S***-Persistent event have been received.

If the subscriber has received both the message and the corresponding R2**S***-Persistent event, this subscriber proceeds to issue an acknowledgement to the repository. This acknowledgment is issued in a S2R-ACK event which can contain either one, an array or a range of acknowledgements: an S2R-ACK event is issued once every few (configurable) seconds or after a certain number of messages have been received.

If the subscriber encounters a R2**S***-Persistent event without the corresponding published message it concludes that the message was lost in transit. The subscriber waits for a pre-configured duration of time before it issues a S2R-NAK event with the missing sequence number(s) for a given reliable-topic to retrieve the corresponding messages.

3.9 Processing subscriber acknowledgements

Upon receipt of an acknowledgement from the subscriber, the repository checks the dissemination table to see if there are any un-acknowledged messages within the range of sequence numbers contained in the S2R-ACK event.

On receipt of the S2R-ACK event from a subscriber, the repository updates the dissemination table entries corresponding to the sequence(s) contained in the event to reflect the fact that the subscriber received messages corresponding to those persistence sequences.

The repository maintains a *sync* for every subscriber to the reliable-topics that it manages. The subscriber sync corresponds to the sequence number up until which the repository is sure that this subscriber has received all preceding messages. A subscriber maintains a local copy of this sync. The sync at a subscriber is advanced by the corresponding reliable-topic repository through the R2S-Sync event. To account for the fact that the S2R-ACK event may be lost in transit to the repository, the subscriber should continue to maintain information about the persistence sequences till such time that it receives a R2S-Sync event.

If the subscriber has received all the messages that it was supposed to receive, and if there were no missed messages between the subscriber's current sync and the highest sequence number contained in the S2R-ACK event, the repository advances the sync point associated with this subscriber and issues a R2S-Sync event which notifies the subscriber about this sync advancement. *Only* upon receipt of this event is the subscriber allowed to advance its sync.

It is possible that the repository, based on the S2R-ACK event, detects that there are some persistence sequences (between the subscriber's sync and highest sequence number in the S2R-ACK event) which were not explicitly acknowledged by the subscriber. The repository assumes that these un-acknowledged messages were lost in transit to the subscriber. The repository also checks to see if, based on the sequences acknowledged, the subscriber's sync can be advanced up until the point at which the sequencing information contained in the S2R-ACK acknowledgement is lower than that of the detected "missed" message.

After the detection of missed sequences the repository issues an R2S-Rectify event, which contains information pertaining to the client's sync advancement (if it is possible) and also the sequencing information and message-identifiers of the missed messages. The repository does not pro-actively retransmit the messages based on the detection of missed messages. This is because it is possible that these missed message(s) are in transit or that just the R2**S***-Persistent event was lost.

3.10 Processing errors & syncs advances

Upon receipt of the R2S-Rectify event a subscriber performs three steps. First, the subscriber checks to see if any of the messages that it maintains in its temporary buffer has the identifier(s) corresponding to those listed in the R2S-Rectify; this accounts for the case where the

R2**S***-Persistent event was lost in transit to the subscriber, but the original published message was not. If the message exists in the temporary buffer, the message is *delivered*.

Second, the subscriber then proceeds to issue a S2R-NAK negative-acknowledgement event to the repository while excluding messages that were reliably delivered in the previous step. The S2R-NAK issued by the subscriber corresponds to the case where messages corresponding to the listed sequence numbers were lost in transit.

Finally, the subscriber advances it sync based on the advancement contained in the R2S-Rectify event. Note that this is also done in response to the R2S-Sync event.

The events between the repository and the subscriber might themselves be lost in transit due to process failures at various components or at the intermediate links. The only way a subscriber will not be routed a published message that it was supposed to receive is if the sync is advanced incorrectly at the repository. However, syncs corresponding to a subscriber (for a specific managed reliable-topic) are never advanced at the repository until the subscriber has explicitly acknowledged every prior message.

3.11 Subscriber and Publisher Recovery

When a subscriber reconnects to the broker network after failures or a prolonged disconnect it needs to retrieve the missed messages published over the reliable-topic. The recovering entity issues a recovery request S2R-Recovery for every reliable-topic that it had previously subscribed to.

Upon receipt of the recovery request, the repository scans the dissemination table starting at the sync associated with the client. The repository then generates an R2S-Rectify event, which is processed by the subscriber to advance its local sync and also to initiate retransmissions as described earlier in section 3.10. Subscription constraints for this subscriber are also retrieved from the repository.

In the case of publisher recovery, the repository's recovery response includes the last known catenation number for a given reliable-topic to which the publisher publishes. Subsequent messages from this publisher results in monotonically increments to this catenation number in the corresponding P2R-Order events.

In our scheme the subscriber is not required to maintain any information pertaining to its sync or the persistence sequences that it had previously received on a given reliable-topic. Similarly, a publisher is not expected to store its catenation number. Publishers and subscribers are automatically notified of their last catenation and sync-advances on the specified reliable-topic. Failures can take place even during this recovery process and the scheme can sustain the loss of both the recovery requests/responses. Figure 1 summarizes interactions outlined in section 3.0.

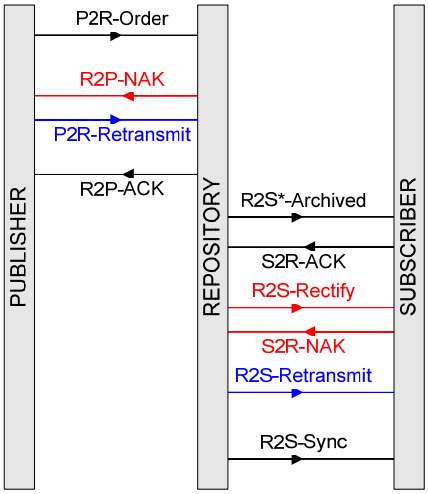

Figure 1: Summary of interactions between entities

4. Repository redundancy & Fault-tolerance

In the previous sections we outlined our strategy to ensure reliable delivery. In this scheme if there is a failure at the repository, the clients interested in reliable communications, over any of the managed reliable-topics, need to wait for this repository to recover prior to the reliable delivery guarantees being met. We now extend this scheme to ensure that reliable delivery guarantees are satisfied in the presence of repository failures. To achieve this we include support for multiple repositories –- constituting a *repository-bundle* –-for a given reliable-topic; it is not necessary that the topics managed by these repositories be identical. A repository may thus be part of multiple repository-bundles at the same time.

We support a flexible redundancy scheme with easy addition and removal of repositories that manage a given reliable-topic. There are no limits on the number of repositories for a given reliable-topic. This scheme can sustain the loss of multiple repositories: in a system with **N** repositories for a given reliable-topic **N-1** of these repositories can fail, and reliable delivery guarantees are met so long as at least one repository is available.

The repositories that constitute the repository-bundle for a given reliable-topic function autonomously. At any given time, for a given reliable-topic, a client communicates with exactly one repository within the corresponding repository-bundle. This entity is also allowed to replace this repository with any other repository within the bundle at anytime.

Besides additional redundancy, and the accompanying fault-tolerance, a highly-available, distributed repository scheme enables clients to exploit geographical and network proximities. Using repositories that are "*closer*" ensures reduced latencies in the receipt of events from the repository. Packet loss-rates typically increase with the number of intermediate hops (for UDP and Multicast).

Besides the selection of the repository from a repository-bundle, as part of the bootstrap, operations at the clients are identical to those in place for a single repository.

4.1 Steering repository

A publisher or subscriber to a reliable-topic can interact with exactly one repository within the repository-bundle for that reliable-topic; this repository is referred to as the *steering repository* for that publisher/subscriber. At any time a client is allowed to replace its steering repository with any other repository from the repository bundle.

Every repository within the bundle keeps track of a client's delivery sequences passively and actively. For a given entity, at any given time, there will be one steering repository operating in the *active* mode by initiating error-corrections and retransmissions. Other repositories operating in *passive* mode do not initiate these actions.

At every repository, within the repository-bundle for a given reliable-topic, the list of registered clients is divided into two sets –- those that the repository steers and those that it does not. The repository operates in the *active* mode for *steered* clients and in the *passive* mode for clients that it does not steer. In the active mode, a repository performs all functions outlined in section 3. In the passive mode, a repository listens to all events initiated by the publishers and subscriber; however, the repository will not issue events – related to reliable communications – to clients that it does not steer. Operating in the passive mode, allows a repository to take over as the steering repository for clients that it does not presently steer.

When a client is ready to initiate reliable communications, it has to designate a steering repository from the set of repositories within the repository-bundle associated with the reliable-topic. Selection of the steering repository is done based on network proximity using probes to compute network round-trip delays to the repositories. The client then issues a event over the repository's communications-topic designating it as the steering repository. Upon receipt of this event, the repository adds that client to its list of steered clients.

4.2 Ordered storage of published messages

For every published message, the publisher issues a P2**R**-Order event (where **R** is the repository-bundle), which is received by all repositories within the repository-bundle. This allows all repositories within the repository-bundle to keep track of published messages. However, only the steering repository (operating in active mode) for this publisher is allowed to issue the R2P-ACK and R2P-NAK events to acknowledge receipt of messages and to initiate retransmissions respectively.

Retransmissions issued in response to the R2P-NAK event are sent to all repositories using the P2**R**-Retransmit event. The rationale for this is that if a message was lost in transit to the publisher's steering-repository, there is a good chance that the message (or the corresponding P2**R**-Order) event was also lost in transit to the other repositories.

4.3 Generation of Persistence Notification

Once a published message is ready for persistent storage at the repository, the message is assigned a sequence number and is stored onto persistent storage along with the published message. In this scheme each repository is autonomous, and thus maintains its own sequencing information. This implies that a message published by a publisher, MAY have different sequence numbers at different repositories. It follows naturally that the sync associated with a given subscriber can be different at different repositories. However, the catenation number associated with a publisher is identical at every repository within the repository-bundle.

A repository computes destinations associated with every published message. These destinations are computed based on the subscriptions registered by subscribers to this reliable-topic irrespective of whether they are steered by the repository or not. The repository then proceeds to issue a persistence notification. The topic associated with the R2***S****-Persistent event is such that it is routed only to the subset ***S**** of its steered subscribers with subscriptions that are satisfied by the topic contained in the original message.

4.4 Acknowledgements, Errors and Syncs

Upon receipt of R2***S****-Persistent events from its steering repository, a subscriber proceeds to issue acknowledgements. This acknowledgement, the S2**R**-ACK is issued over the repository-bundle communications topic. Since, the message is received by the repository-bundle, all repositories are aware of delivery sequences at different subscribers. The S2**R**-ACK event contains sequence numbers corresponding to its steering repository and also includes the identifier associated with the steering repository.

Error correction, and sync advancements, for a given subscriber is initiated by its steering repository through the R2S-Rectify event. Retransmission requests by a subscriber are targeted to its steering repository in the S2R-NAK event.

4.5 Gossips between repositories

Repositories within a repository-bundle gossip with each other. Repositories within a repository-bundle need to exchange about the registration/de-registration of clients to the managed reliable-topic. Additional, and removal, of subscription to this reliable-topic are also exchanged between all repositories within the bundle. A given repository stores each of these actions and assigns each action the next available sequence number.

4.5.1 Processing stored messages

A repository assigns monotonically increasing sequence numbers to each message that it stores. At regular intervals or after the persistent storage of a certain number of messages, the repository issues a Gossip-ACK event, which contains an array of entries corresponding to its persistent storage of published messages. Each entry contains the publisher identifier, the catenation number assigned by the publisher, the message identifier and the sequence number assigned by the repository. This gossip message plays a big role in allowing repositories to be aware of the sequence numbers associated with a specific message at different repositories.

Every repository also maintains a repository-table. In this table for every publisher-catenation number pair, the repository maintains the sequence number assigned to it by every repository (including itself) within the bundle. For every message that it stores, a repository adds an entry in the repository-table. Upon receipt of the Gossip-ACK event from a repository, this entry (corresponding to the publisher/catenation pair) is modified to reflect the sequence numbers assigned at different repositories.

The repository table thus allows a repository to correlate sequence numbers assigned to a given message at every other repository. The table also tracks messages not received by other repository. This allows a repository to recover from any other repository after a failure. This repository table also plays an important role in processing acknowledgements from subscribers that it does not steer.

4.6 Processing subscriber acknowledgements

When a repository receives a S2**R**-ACK event from a subscriber, it checks to see if it steers the subscriber. If it does, the repository simply proceeds to update its dissemination table to reflect receipt of the message at the repository. If the repository does not steer the subscriber that issued the acknowledgement, the repository retrieves the sequence number corresponding to the original message from the repository-table. It then proceeds to update the dissemination-table for that sequence number to confirm receipt from the subscriber in question. This scheme allows all repositories are aware of delivery sequences at the various subscribers irrespective of whether they are steered or not. Furthermore, based on the S2**R**-ACK acknowledgements from a subscriber, every repository computes its sync for that subscriber. Additionally, at regular intervals, a repository gossips about sync advancements for its steered subscribers.

Since all repositories process acknowledgements from all the subscribers any one of these repositories can take over as the steering repository for a given subscriber. Furthermore, since all messages issued by all publishers are stored at all repositories, and since the catenation numbers are identical on all repositories, a repository can take over as the steering repository for a given publisher at any time.

4.7 Dealing with repository failures

A publisher detects a failure in its steering repository, if it does not receive R2P-ACK events for published messages within a certain duration. A subscriber detects a steering-repository failure if it receives published messages to reliable-topics, but no corresponding persistence notifications from its steering repository. These clients then proceed to discover a new steering repository. The publisher then exchanges information about its catenation number with the replacement steering repository. If there is a mismatch wherein the steering repository's catenation is lower than that at the publisher, the repository proceeds to retrieve this message from a repository within the bundle.

4.8 Recovery of a repository

Upon recovery from a failure, it needs to discover an assisting-repository: this is a repository within the repository bundle that is willing to assist the repository in the recovery process. The recovering replica first checks to see if the list of registered clients and subscriptions have changed, and proceeds to retrieve updates to this list.

Next, the repository proceeds to retrieve the list of catenation numbers associated with the publishers. Based on these catenation numbers, the repository computes the number of missed messages and proceeds to set aside the corresponding number of sequences. For messages (missed and real-time) that it stores, a recovering-repository issues Gossip-ACK acknowledgements at regular intervals.

The recovering-repository proceeds to do two things in parallel. First, it proceeds to retrieve missed messages from the assisting repository. For every missed message the recovering-repository also retrieves the dissemination list associated with it. This allows a repository to keep track of the subscribers that have not acknowledged these messages. Additionally, the repository-table entries corresponding to each message are also retrieved. A repository cannot be the steering repository for any entity till such time that all the missed messages have been retrieved.

Second, it subscribes to the various communications topics so that it can start receiving messages published in real-time. The first time a repository receives a message from a publisher, it checks to see if the catenation number associated with the message indicates missed messages. This could happen because the missed message(s) would have been in transit to the assisting repository. Thus, during recovery if the assisting repository reported a catenation number of `2000`, and if the catenation number associated with the first real-time message received from the publisher is `2010` it implies that thee were `9`

additional missed messages from this publisher. The repository sets aide 9 sequence numbers, and issues a request to retrieve these messages. The repository also proceeds to store the published message based on the newly advanced sequence number.

4.9 Addition of a repository

When a repository is added to the repository-bundle associated with a reliable-topic, the newly added repository takes the following steps. First, it needs to discover an assisting-repository: this is a repository which is present in the repository bundle and one which is willing to assist the repository in the addition process.

Second, the repository retrieves the list of registered clients, and the subscriptions registered by the registered subscribers. As described in section 4.8 the repository then proceeds to retrieve missed messages along with the corresponding dissemination lists and repository-table entries in addition to processing real-time messages.

4.10 Graceful removal of a repository

When a repository is ready to leave a repository bundle, it proceeds to issue an event to its active steered clients, requesting them to migrate to another repository. The departing-repository then operates in silent mode as far as the clients are concerned. The departing-repository also gossips with other repositories within the repository-bundle to check if the catenation numbers associated with previously steered publishers is greater than or equal to its last known value at the departing-repository.

Once a repository has confirmed that all messages published by its previously steered publisher have been received at one of the repositories within the bundle, it is ready to leave the repository-bundle. The departing repository then simply issues a Gossip-LEAVE event. Repository table entries corresponding to this repository will no longer be maintained at other repositories.

5. Experimental Results

We have measured several aspects of the reliable delivery framework (implemented within NaradaBrokering) to determine costs involved in reliable communications. All processes executed within Sun's Hotspot™ JVM 1.4.2 with Linux (2.4.22) as the OS, and were hosted on a 100 Mbps LAN. In our benchmarks we have used the topologies depicted in Figure 2, the repositories, brokers and sets of clients are all hosted on different machines. In all test cases, to obviate the need for clock synchronizations, the publisher and the *measuring* subscriber (which reports the results) were hosted on the same machine. Note that the publisher and measuring subscriber are connected to different brokers in topologies B and C. Machines involved in the benchmark have the following profile: 4 CPU (Xeon, 2.4GHz), 2GB RAM. The persistent storage used by the repositories is MySQL 5.0.

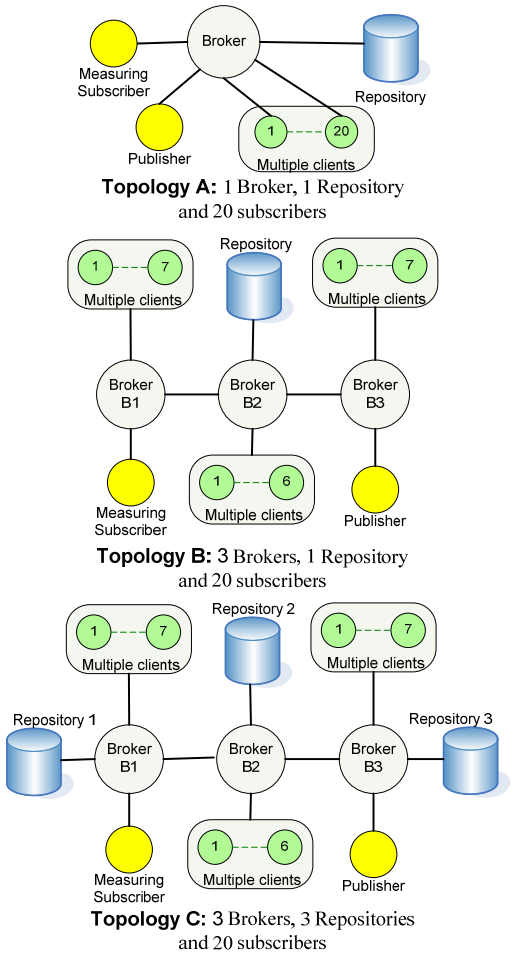

Figure 2: Benchmark Topologies

For each topology we also measured the costs involved in best effort delivery. This allows us to see the overheads introduced by the reliable delivery scheme. Each reported delay value (for a given payload size) in the graphs is the mean of repeating the test 50 times. We also report the standard deviation (SD) for the delay samples in the test.

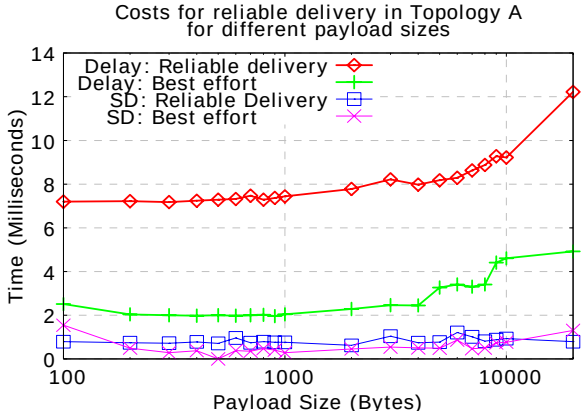

Figure 3: Costs for reliable delivery: Topology A

Figure 3 depicts the cost involved in reliable delivery for topology A, while Figure 4 depicts the costs for reliable delivery in Topology B and Figure 5 depicts the costs for reliable delivery in Topology C. In general the costs for reliable delivery are higher than the corresponding costs for best-effort delivery. The overheads for reliable delivery are caused by the costs for storing the events to persistent storage, and waiting for the message to be stable (i.e. waiting for a message to be persistent) prior to delivery. It should be noted that the best effort delivery will provide none of the guarantees provided by the fault-tolerant delivery scheme outlined in this paper.

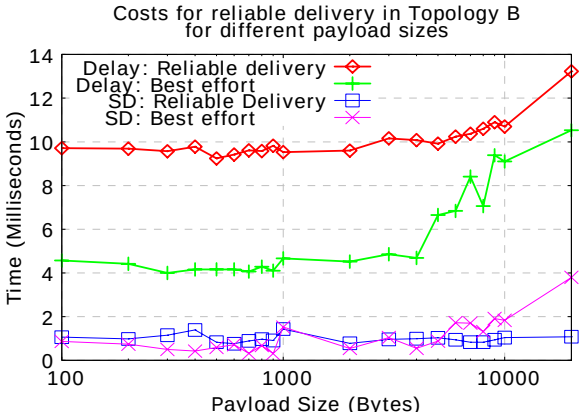

Figure 4: Costs for reliable delivery: Topology B

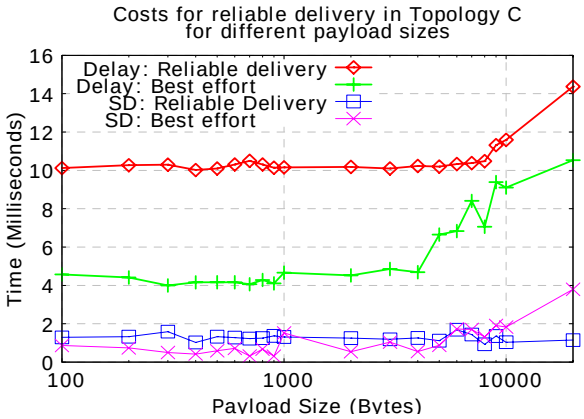

Figure 5: Costs for reliable delivery: Topology C

Figure 6 depicts the costs involved in reliable delivery in different topologies for different payload sizes. In general the costs increase as the number of brokers, and the number of repositories, for a given reliable-topic increase. The results also demonstrate that the costs for reliable delivery are acceptable. There is a slight (and acceptable) increase in the costs for reliable delivery when there are 3 repositories for the reliable-topic in Topology C compared to just one repository in Topology B.

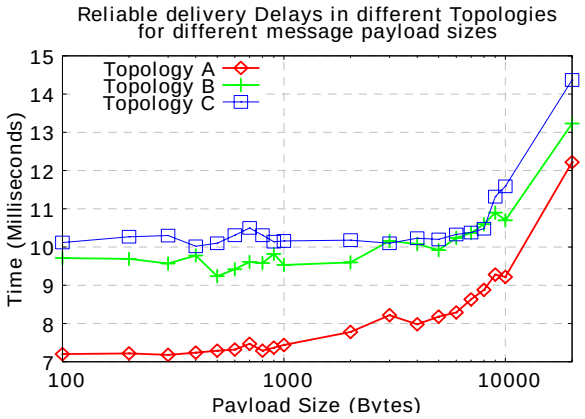

Figure 6: Overheads for reliable delivery in different Topologies

Costs associated with various aspects of the framework are summarized in Table 1. Some of these costs are reported in milliseconds (mS) and some in microseconds (µS). The values that we report are for messages with a payload size of 8KB.

Table 1: Reliable delivery costs within the framework. Values reported for a message size of 8KB.

Operation	Mean	Std Deviation	Std Error
End-To-End Delivery			
1 Broker, 1 RDS, 20 clients	8.88 mS	0.81 mS	0.11 mS
3 Brokers, 1 RDS, 20 clients	10.60 mS	0.83 mS	0.12 mS
3 Brokers, 3 RDS nodes and 20 clients	10.48 mS	0.93 mS	0.13 mS
Storage Overheads			
Message Storage	1408 µS	141.71 µS	31.7 µS
Message Retrieval	669 µS	77.93 µS	17.4 µS
RDS recovery in a single repository system			
Recovery time after a failure or scheduled downtime	85.7mS	4.3mS	958 µS
Client Recovery			
Time to generate Recovery response for a publisher	825 µS	215 µS	48 µS
Time to generate Recovery response for a subscriber	1613 µS	588 µS	131 µS
Repository Recovery (1000 missed messages and 20 clients)			
Recovery Response generation for Repository	59.28 mS	5.54 mS	2.77 mS
Recovery Time at repository	11172mS	1733 mS	867 mS
Repository Gossips			
Generation of gossip	243 µS	50 µS	11 µS
Processing a gossip	241 µS	16 µS	3 µS

6. Related Work

The virtual synchrony model, adopted in Isis [4], works well for problems such as propagating updates to replicated sites. This approach does not work well in situations where the client connectivity is intermittent, and where the clients can roam around the network. Systems such as Horus [5] and Transis [6] manage minority partitions and can handle concurrent views in different partitions. The overheads to guarantee consistency are however too strong for our case. Spinglass [7] employs gossip-style algorithms, where recipients periodically compare the message digest of the received message with one of the group members. Deviations in the digest result in solicitation requests (or unsolicited responses) for missing messages between these recipients. This approach is however unsuitable when memberships are very fluid.

DACE [8] introduces a failure model that tolerates crash failures and partitioning, while not relying on consistent views being shared by the members through a self-stabilizing exchange of views. This however may prove to be very expensive if the number and rate at which the members change their membership is high. The Gryphon [9] system uses knowledge and curiosity streams to determine gaps in intended delivery sequences. This scheme requires a persistent storage at every publishing site and meets the delivery guarantees as long as the intended recipient stays connected in the presence of failures.

Since message queuing products (MQSeries) [10] are statically pre-configured to forward messages from one queue to another they generally do not handle network changes (node/link failures) very well. The WS-ReliableMessaging [11] specification provides a scheme to ensure reliable delivery of messages between the source and the sink for a given message.

The Data Replication Service (DRS) [12], within the Globus Toolkit, leverages the Replica Location Service which is a distributed registry that keeps track of replicas on storage systems, and facilitates queries to locate replicated files. The Storage Resource Broker (SRB) [13] is a middleware that provides applications a uniform interface to access heterogeneous distributed storage systems. It utilizes a metadata catalog called MCAT which manages descriptive and system metadata associated with data collections and system resources. Both DRS and SRB transfer and replicate files and rely on a separate service to locate replicated files. In our system, we replicate messages. Messages are stored at repositories where the underlying persistent storage could be based on databases or flat-files with no need to maintain a separate registry or metadata service to manage the replications.

7. Conclusions & Future Work

In this paper we presented our scheme for fault-tolerant, reliable delivery of messages in publish/subscribe systems. The experimental results demonstrate the feasibility of this scheme and the acceptability of the costs introduced therein. As part of our future work we will research issues related to repository placement schemes. Specifically, the replica placement algorithm needs to ensure that average latencies for reliable communications are reduced as a result of its placement.

References

[1] S. Pallickara and G. Fox. A Scheme for Reliable Delivery of Events in Distributed Middleware Systems. POSTER. Proceedings of the IEEE International Conference on Autonomic Computing.NY. pp 328-329

[2] S. Pallickara and G. Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of the ACM/IFIP/USENIX Middleware Conference Middleware-2003. pp 41-61.

[3] S. Pallickara, G. Fox and H. Gadgil. On the Creation & Discovery of Topics in Distributed Publish/Subscribe systems. Proc of the IEEE/ACM GRID 2005, pp 25-32.

[4] Kenneth Birman. Replication and Fault tolerance in the ISIS system. In Proceedings of the10th ACM Symposium on Operating Systems Principles, pages 79–86, 1985.

[5] R Renesse, K Birman, and S Maffeis. Horus: A flexible group communication system. In Communications of the ACM, volume 39(4). April 1996.

[6] D Dolev and D Malki. The Transis approach to high-availability cluster communication. In Communications of the ACM, volume 39(4). April 1996.

[7] K. Birman, R van Renesse and W Vogels. Spinglass: Secure and Scalable Communications Tools for Mission-Critical Computing. International Survivability Conference and Exposition. DARPA DISCEX-2001, CA, June 2001.

[8] R. Boichat, et al. Effective Multicast programming in Large Scale Distributed Systems. CCPE, 2000.

[9] S. Bhola, et al: Exactly-once Delivery in a Content-based Publish-Subscribe System. DSN 2002: 7-16

[10] The IBM WebSphere MQ Family. http://www-3.ibm.com/software/integration/mqfamily/

[11] Web Services Reliable Messaging Protocol (WS-ReliableMessaging) March, 2006. From IBM, Microsoft

[12] Chervenak et al. Giggle: A Framework for Constructing Scalable Replica Location Services, Proceedings of ACM/IEEE Supercomputing 2002 (SC2002).

[13] C. Baru et al. The SDSC Storage Resource Broker. In:Procs. of CASCON'98,Toronto, Canada (1998)