# Fault Tolerant Reliable Delivery of Events in Distributed Middleware Systems

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

## Abstract

*Reliable delivery of events (or messages, where the terms messages and events are used interchangeably) is an important problem that needs to be addressed in distributed systems. Increasingly, interactions between entities within a distributed system are encapsulated in events. In this paper we present our strategy to enable reliable delivery of events in the presence of link and node failures. We then present our strategy to make this scheme even more failure resilient, by incorporating support for replicated reliable delivery nodes. Each replica functions autonomously and makes decisions independently. The strategy we present incorporates schemes to add or remove replicas dynamically depending on the failure resiliency requirements. Finally, in this scheme if there are* ***N*** *available replicas, reliable delivery guarantees will be met even if* ***N-1*** *replicas fail.*

**Keywords:** publish/subscribe, middleware, reliable delivery, fault tolerance and robustness.

## 1 Introduction

Increasingly interactions that services and entities have with each other, and among themselves, are network bound. In several cases these interactions can be encapsulated in events. These events can encapsulate, among other things, information pertaining to transactions, data interchange, system conditions and finally the search, discovery and subsequent sharing of resources. The routing of these events is managed by a middleware, which as the scale and scope of the system increases, needs to be based on a distributed messaging infrastructure.

In this paper we describe our scheme for the reliable delivery of events within NaradaBrokering [1-5], which is a distributed messaging infrastructure supporting a wide variety of event driven interactions – from P2P interactions to audio-video conferencing applications. The scheme outlined in this paper facilitates delivery of events to interested entities in the presence of node and link failures. Furthermore, entities are able to retrieve any events that were issued during an entity's absence (either due to failures or an intentional disconnect). The scheme outlined in this paper can be easily extended to ensure guaranteed exactly-once ordered delivery.

This reliable delivery guarantee must hold true in the presence of four distinct conditions.

1. Broker and Link Failures: The delivery guarantees need to be satisfied in the presence of individual or multiple broker and link failures. It is conceivable that the entire broker network may fail. In this case, once the broker network recovers (even if the new broker network comprises of a single broker node) the delivery guarantees are met.
2. Prolonged Entity disconnects: Entities interested in a certain event may not be online at the time the event is published. This entity may reconnect after disconnects and the delivery guarantee will be met with the entity receiving all the events missed in the interim.
3. Stable Storage Failures: It is possible that stable storages present in the system may fail. The delivery guarantees must be satisfied once the storage recovers.
4. Unpredictable Links: The events can be lost, duplicated or re-ordered in transit over individual links, en route to the final destinations.

The remainder of this paper is organized as follows. Section 2 provides an overview of related work. Section 3 provides an overview of NaradaBrokering, while section 4 includes details regarding the main components of the reliable delivery scheme. Sections 5 and 6 outline our scheme for ensuring reliable delivery and recovering from failures. In section 7 we describe how this scheme was deployed to make GridFTP mode robust. We include experimental results in section 8. In section 9 we extend our scheme to include support for replicated reliable delivery nodes, which provide much greater resilience to failures. In section 10 we include a discussion and results from our implementation of the WS-ReliableMessaging specification. Finally in section 11 we outline our conclusions.

## 2 Related Work

The problem of reliable delivery [6, 7] and ordering [8, 9] in traditional group based systems with process crashes has been extensively studied. The approaches normally have employed the primary partition model [10], which allows the system to partition under the assumption that there would be a unique partition which could make decisions on behalf of the system as a whole, without risk of contradictions arising in the other partitions and also during partition mergers. However the delivery requirements are met only within the primary partition. Recipients that are slow or temporarily disconnected may be treated as if they had left the group.

This virtual synchrony model, adopted in Isis [11], 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 [12] and Transis [13] manage minority partitions (by having variants of the virtual synchrony

model) and can handle concurrent views in different partitions. The overheads to guarantee consistency are however too strong for our case.

Spinglass [14] employs "gossip" style algorithms, where recipients periodically compare their disseminations with other members of the group. Each recipient compares it dissemination sequence (a message digest of the message sequences received so far) 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 for our scheme where memberships are fluid and hence a recipient is unaware of other recipients that should have received the same message sequences. Approaches to building fault-tolerant services using the state machine approach have been suggested in Ref [15].

DACE [16] introduces a failure model, for the strongly decoupled nature of pub/sub systems. This model tolerates crash failures and partitioning, while not relying on consistent views being shared by the members. DACE achieves its goal through a self-stabilizing exchange of views through the Topic Membership protocol. This however may prove to be very expensive if the number and rate at which the members change their membership is high. The Gryphon [17] system uses knowledge and curiosity streams to determine gaps in intended delivery sequences. This scheme requires persistent storage at every publishing site and meets the delivery guarantees as long as the intended recipient stays connected in the presence of intermediate broker and link failures. It is not clear how this scheme will perform when most entities within the system are both publisher and subscribers, thus entailing stable storage at every node in the broker network. Furthermore it is conceivable that the entity itself may fail, the approach does not clearly outline how it handles these cases. Systems such as Sienna [18] and Elvin [19] focus on efficiently disseminating events, and do not sufficiently address the reliable delivery problem.

Message queuing products (MQSeries) [20] are statically pre-configured to forward messages from one queue to another. This leads to the situation where they generally do not handle changes to the network (node/link failures) very well. Furthermore these systems incur high latency since they use the store-and-forward approach, where a message is stored at every stage before being propagated to the next one. Queues need to also recover within a finite amount of time to resume operations.

The Fault Tolerant CORBA (FT-CORBA) [21] specification from the OMG defines interfaces, policies and services that increase reliability and dependability in CORBA applications. The fault tolerance scheme used in FT-CORBA is based on entity redundancy [22], specifically the replication of CORBA objects. In CORBA objects are uniquely identified by their interoperable object reference (IOR). The FT-CORBA specification introduces interoperable group object references (IGOR). When a remote object, the client can access a replica simply by iterating through the references contained in the IGOR until the invocation is successfully handled by the replicated object. The specification introduces several schemes to manage different replication schemes.

The DOORS (Distributed Object-Oriented Reliable Service) system [23] incorporates strategies to augment implementations of FT-CORBA with real time characteristics. Among the issues that the DOORS system tries to address are avoiding expensive replication strategies and dealing with partial failure scenarios. DOORS provides fault tolerance for CORBA ORBs based on the service approach. Approaches such as Eternal [24] and Aqua [25], provide fault tolerance by modifying the ORB. OS level interceptions of have also been used to tolerate faults in applications.

The WS-ReliableMessaging [26] specification provides a scheme to ensure reliable delivery of messages between the source and the sink for a given message. The specification provides an acknowledgement based scheme to ensure that data is transferred reliably between the communicating entities. The specification, though it is for point-to-point communications, supports composition and interoperates with specifications pertaining to policies, transactions, coordination and metadata exchanges..

## 3 NaradaBrokering Overview

NaradaBrokering [1-5] (www.naradabrokering.org) is distributed messaging infrastructure designed to run on a large network of cooperating broker nodes. Communication within NaradaBrokering is asynchronous and the system efficiently routes any given event between the originators and the registered consumers of that event. NaradaBrokering places no constraints either on the number of entities or on the size/rate of the events.

The smallest unit of the messaging infrastructure that provides a back bone for routing these events needs to be able to intelligently process and route events while working with multiple underlying network communication protocols. We refer to this unit as a *broker* where we avoid the use of the term *server*s to distinguish it clearly from the application servers that would be among the sources/sinks to events processed within the system. Entities within the system utilize the broker network to effectively communicate and exchange data with each other. These interacting entities could be any combination of users, resources, services and proxies thereto. These are also sometimes referred to as clients.

In NaradaBrokering we impose a hierarchical structure on the broker network, where a broker is part of a cluster that is part of a super-cluster, which in turn is part of a super-super-cluster and so on. Clusters comprise strongly connected brokers with multiple links to brokers in other clusters, ensuring alternate communication routes during failures. This organization scheme results in average communication pathlengths between brokers increasing logarithmically with geometric increases in network size, as opposed to exponential increases in uncontrolled settings.

### 3.1 Entities, Profiles and Event Templates

An event comprises of headers, content descriptors and the payload encapsulating the content. An event's headers provide information pertaining to the type, unique identification, timestamps, dissemination traces and other QoS related information pertaining to the event. The content descriptors for an event describe information pertaining to the encapsulated content. The content descriptors and the values these content descriptors take collectively comprise the event's *content synopsis*.

The set of headers and content descriptors constitute the *template* of an event. Events containing identical sets of headers and content descriptors are said to be conforming to the same template. It should be noted that the values which the content descriptors and payloads take might be entirely different for events conforming to the same template. When we say template events, we mean events conforming to the same template.

Entities have multiple *profiles* each of which signifies an interest in events conforming to a certain template. This interest is typically specified in the form of a constraint that events need to satisfy, before being considered for routing to the entity in question. This constraint is also sometimes referred to as a subscription. Entities specify constraints on the content descriptors and the values some or all of these descriptors might take. Individual profiles can also include information pertaining to security and device types for special processing of events. When an event traverses through the system these constraints are evaluated against the event's synopsis to determine the eventual recipients.

An event's synopsis thus determines the entities that an event needs to be routed to. Two synopses are said to be equal if the content descriptors and the values these descriptors take are identical. It is possible for events with the same synopsis to encapsulate different content in its payload. It is however possible for events with different synopses to be routed to the same set of destinations.

The type of constraints specified by the entities varies depending on the complexity and type of the content descriptors. In NaradaBrokering the specified constraints could be a simple character string based equality test, an XPath query on the XML document and an SQL like query on the properties and the values these properties take, Integers or (tag, value) equality tests.

There is a crucial difference between constraints specified on simple and complex event templates. In the former case, all entities that have specified the constraint are valid destinations. In the latter case it is possible that none, some or all the entities that have specified constraints on the same complex template are valid destinations for the event.

Every entity within the system has a unique identifier (EID) associated with it. Every entity within the system subscribes to its EID to ensure that interactions targeted to it are routed and delivered by the broker network.

## 4 The reliable delivery scheme

In this section we describe in detail the key elements of our reliable delivery scheme. To ensure the reliable delivery of events conforming to a specific template to registered entities there are 3 distinct issues that need to be addressed. First, there should be exactly one RDS node that is responsible for providing reliable delivery for a specific event template. In a subsequent section we discuss the presence of replicators within the system to provide additional robustness. Second, entities need to make sure that their subscriptions are registered with RDS. Finally, a publisher needs to ensure that any given event that it issues is archived at the relevant RDS. In our scheme we make use of positive acknowledgements (abbr. ACK) and negative acknowledgements (abbr. NAK).

### 4.1 Objectives of this scheme

There are several objectives that we seek to achieve in our scheme. We may enumerate these below

- Not tied to a specific storage type: We need the ability to maintain different storage types (flat files, relational databases or native XML databases) for different event templates.
- Unconstrained RDS instances: There could be multiple RDS instances within the system. A given RDS instance could manage reliable delivery to one or more templates.
- Autonomy: It should be possible for individual entities to manage their own event templates. This would involve provision of stable storage and authorizing entity constraints on the managed template.
- Location independence: A RDS node can be present anywhere within the system.
- Flexible template management: It should be possible to handoff template managements easily within the system.
- Fast Recovery schemes: The recovery scheme needs to efficiently route missed events to entities.

### 4.2 The Reliable Delivery Service (RDS)

RDS can be looked upon as providing a service to facilitate reliable delivery for events conforming to any one of its managed templates. To accomplish this RDS provides four very important functions. First, RDS archives all published events that conform to any one of its managed templates. This archival operation is the precursor to any error corrections stemming from events being lost in transit to their targeted destinations and also for entities recovering either from a disconnect or a failure.

Second, for every managed template, RDS also maintains a list of entities (and the corresponding EIDs) for which it facilitates reliable delivery. RDS may also maintain information regarding access controls, authorizations and credentials of entities that generate or consume events targeted to this managed template. Entity registrations could either be user controlled or automated.

Third, RDS also facilitates calculation of valid destinations for a given template event. This is necessary

since it is possible that for two events conforming to the same template, the set of valid destinations may be different. To ensure that system resources are not expended in ensuring reliable delivery of an event to uninterested entities the service maintains a list of the profiles and the encapsulated constraints specified by each of the registered entities. For each managed template the service also hosts the relevant matching engines, which computes entity destinations from a template event's synopsis. It is conceivable that two or more of the managed templates share the same matching engine.

Finally, RDS keeps track not only of the entities that are supposed to receive a given template event, but also those entities that have not explicitly acknowledged receipt of these events. The information maintained by RDS forms the basis for servicing retransmission and recovery requests initiated by registered entities.

Every event template within the system has a unique identifier – templateID. RDS advertises its archival capabilities for a specific event template by subscribing to: RDS/EventType/Template-ID. For example RDS/XML/98765213 could be the subscription from a RDS node managing reliable delivery functions for an XML template with templateID 98765213.

RDS also archives entity profile related operations initiated by registered entities. These operations include the addition, removal and update of constraints specified on any of the managed templates. For every archived event or other entity profile related operations, RDS assigns monotonically increasing sequence numbers. These sequence numbers play a crucial role in error detection and correction, while also serving to provide audit trails. Templates managed by a RDS are also referred to as *reliable templates*.

### 4.3 Publishing template events

In this sub-section we discuss details pertaining to publishing events to a reliable template. A publisher can of course generate events that conform to different templates. The system imposes no constraints on the number and type of template events that a publisher is allowed to generate.

When an entity is ready to start publishing events on a given template (either for the first time or after a prolonged disconnect) it issues a discovery request to determine the availability of RDS that provides archival for the generated template events. The publisher will not publish template events till such time that it receives a confirmation that a managing RDS is available. The discovery message is issued with a synopsis of the form RDS/EventType/TemplateID.

Since the RDS that would perform archival operations for these template events had previously subscribed to this topic, it can respond to this discovery request. The request and responses to discover this availability of RDS can of course be lost. The publisher issues this discovery request at increasingly larger intervals till such time that it receives the discovery response. The discovery operation can timeout after a certain number of failed attempts or a specified elapsed time. A publisher is ready once it confirms the existence of RDS for a templateID.

For every template event that it generates, the publisher is required to ensure that these events are archived by the relevant RDS. *Archival negotiations* occurring between a publishing entity and RDS is a precursor to ensuring reliable delivery of that event to all interested entities. Archival negotiations pertain to the acknowledgement of received template events and also requests for retransmissions of template events lost en route to the RDS. The negotiations comprising acknowledgements and retransmission requests are initiated by RDS.

To ensure archival, the publisher generates a *companion event* for every template event that it generates. The companion event has only one destination – the relevant RDS – and contains enough information to determine the right ordering and also to detect losses that might have taken place during the template events' transit to RDS. A given template event and its companion event share the same EventID and entity identifier EID.

A publisher assigns monotonically increasing *catenation numbers* to every template event that it publishes. These catenation numbers allow us to determine the order in which the template events were generated. Since it is conceivable that a publisher might publish events conforming to multiple templates, for a given template we also need information pertaining to the catenation number associated with the last published event that conformed to this template. Catenation information is maintained in a template event's companion event. Figure 1.(a) and depicts a template event, while Figure 1.(b) depicts the companion event.

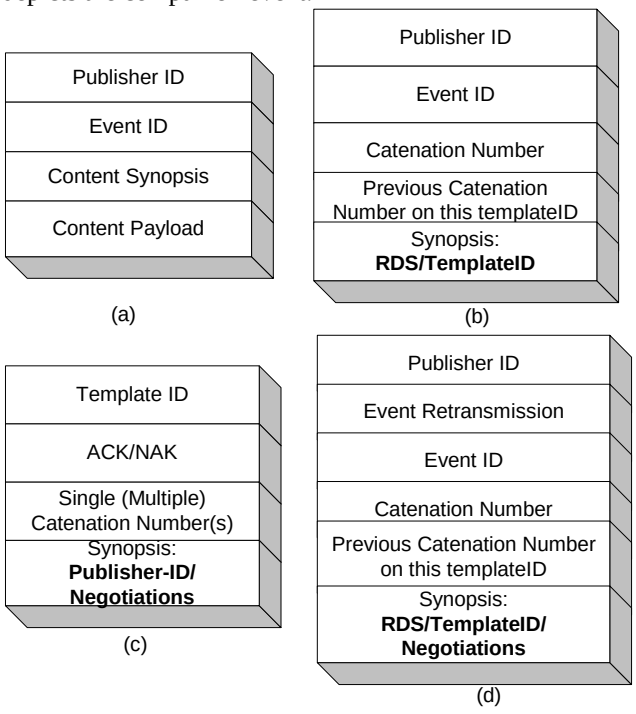


**Figure 1: Publishing template events**

A RDS generally has the weakest constraints on a template event's synopsis. This ensures that most (if not all) templates events are routed to RDS. RDS also maintains catenation information pertaining to each registered publisher for every managed templateID.

Upon receiving an event conforming to one of its managed templates, RDS does not act on this template event till such time that it receives the corresponding companion event. Based on the catenation information in the companion events RDS has the ability to determine the order (publisher) and to detect any missed template events. RDS can easily retrieve the precise catenation information that should be used to retrieve a missed template event.

Based on the catenation, successful receipt can be confirmed, if there were no prior losses and if the template event is in the right order. Upon successful receipt the event is archived and a negotiation ACK is issued (with synopsis EID/Negotiations) to the publisher EID. Otherwise, a negotiation NAK with the appropriate catenation is issued to the publisher EID. The format of the archival negotiation request is depicted in Figure 1.(c). Receipt of the archival negotiation ACK signifies that all template events issued by the publisher up until that point have been received and successfully archived. A publisher is expected to hold an event in its buffer till such time that it receives a negotiation ACK confirming the archival of the template event.

Upon receipt of the negotiation ACK the publisher releases the template event corresponding to the catenation information included in the negotiation ACK. If on the other hand, the publisher receives a negotiation NAK from RDS, the publisher creates an event, as depicted in Figure 1.(d). This republished event is a fusion of the information contained in both the template event and its companion event. This republished event is routed by the system to the requesting RDS (with synopsis RDS/TemplateID/ Negotiations).

Finally, it is possible that companion event for a given template event might have been lost in transit. In this case RDS issues an archival negotiation NAK with event's identifier EventID to retrieve the template event. If both template and companion events for a catenation are lost, subsequent events (template or companion) will trigger a request to retrieve this lost template event.

### 4.4 Archiving template events at RDS

Upon confirming successful receipt of a template event at RDS the relevant matching engine is used to compute destinations associated with the template event. The template event and its intended destinations now need to be archived.

At RDS we maintain two sets of information. First, we create the inventory event which includes the template event in its entirety minus the dissemination traces associated with it. Also associated with every inventory event is a monotonically increasing *sequence number*, which plays a crucial role in recoveries. We also store the templateID and the eventID associated with the original template event. Including the templateID in the inventory event allows for easier migrations of one or more managed templates to other entities, locations or underlying storage implementations. The eventID information is useful in dealing with certain types of retransmission requests. Figure 2.(a) depicts the structure of the *inventory event*.

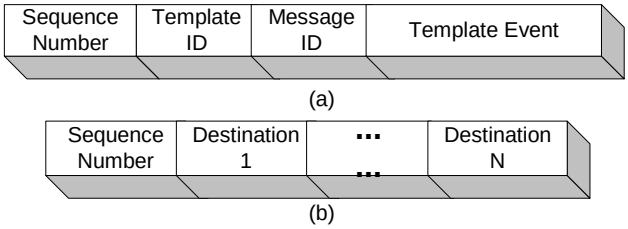


**Figure 2: The stored elements**

Second, we separately maintain a dissemination table. For a given sequence number, the dissemination table maintains information regarding the list of entities to which the original template event is yet to be routed to. The dissemination table is continually updated to reflect the successful delivery of the template event to the intended destinations. The dissemination table thus allows us to determine holes in sequences for events that should have been delivered to an entity. Figure 2.(b) depicts the structure of the record format in the dissemination table.

### 4.5 Subcriptions

RDS stores entity interactions corresponding to registration and change of profiles (including constraint additions, removals and updates) too. Just like template events, these entity registrations and profile updates also have a sequence number associated with it.

The first time an entity registers with RDS, the sequence number associated with the archival of this interaction is its *epoch*. The epoch signifies the point from which the registered entity is authorized to receive events conforming to the template for which it registers.

The subscribing entity also needs to make sure that its profile and encapsulated constraint are registered at RDS managing the template in which it is interested in. Prior to this the entity needs to discover RDS that manages the template in question. This discovery process is similar to what we described for the publishing entity in an earlier section. We impose no limit on the number of constraints a subscribing entity specifies on a given event template.

To register its constraint on an event template, the entity proceeds to issue a subscription message comprising its identifier and the constraint. This subscribing event is issued with synopsis RDS/TemplateID/ProfileChange. The entity will issue this event at regular intervals until it receives a response from the relevant RDS confirming the addition of this profile to the list of profiles that RDS maintains. This response contains the sequence number corresponding to the archival performed by RDS. If this is the first profile specified by the subscribing entity on a given template this is that entity's epoch on that template. The response indicates that the entity's change has been

registered and that the entity will have reliable delivery of template events from this point on if any template event satisfies the constraints specified in the entity's profile. A newly specified entity profile on a templateID is valid only after an express notification from the relevant RDS signifying receipt of the entity profile change.

It is conceivable that there could be multiple profile change requests on a given template and the corresponding responses may be lost. The detection and correction of these are errors and losses are governed by the same principles that correspond to ensuring storage of template events issued by a publisher.

It should be noted that for a given template and an entity consuming those template events, there is a *sync* at both the entity and RDS. The sync (for a specific templateID) associated with an entity corresponds to the sequence number, up until which, RDS is sure that the subscribing entity has received all prior events up until its epoch. There is a sync associated with *every* reliable template to which an entity is registered to. However, the sync (for a specific templateID) at an entity cannot be advanced until it has been advanced by RDS and this advancement is notified to the entity. The sync advancement at an entity is an indication that the subscriber has received all the template events that it was supposed to receive up until the sequence number contained in the sync advancement.

## 5 Ensuring Reliable Delivery

Once a template event has been archived, RDS issues an *archival notification*. Archival notifications allow a subscribing entity to keep track of the template events it has received while facilitating error detection and correction. This archival notification, depicted in Figure 3.(a), contains among other things the sequence number associated with the archival and also the sequence number associated with the last archival of an event which conformed to this template. We need to make sure that the archival notifications reach the entities interested in the corresponding template event. To do this we make sure that the synopsis for this archival notification is the same as that of the original template event.

*Invoice events* encapsulate exchanges, between the entity and RDS, corresponding to the set of template events received and also requests to retransmit missed template events. The archival notification for a template event includes the eventID for that template event. Upon receipt of an archival notification the subscribing entity checks to see if it has received the corresponding template event. If it has indeed received the template event the subscribing entity issues an ACK invoice event, which outlines the archival sequence(s) that it has received after its sync was last advanced by RDS. An entity must await a response to this ACK invoice to advance the sync associated with the template. Figure 3.(b) depicts the structure of the ACK invoice event.

To account for the fact that ACK invoice events may be lost in transit to RDS, the entity should continue to maintain information about the archival sequences it has received. If this information is lost, RDS will route those events which were not explicitly acknowledged using invoice events.

Upon receipt of the ACK invoice event from the entity, RDS updates records in the dissemination table associated with the sequence(s) outlined in the ACK invoice event to reflect the fact that the entity received template events corresponding to those archival sequences. If the entity has received all the template events it was supposed to receive and there were no missed events between the entity's current sync and the highest sequence number contained in the ACK invoice event, RDS advances the sync point associated with this entity and issues the ACK-Response invoice event which notifies the entity about this sync advancement. *Only* upon receipt of this event is the entity allowed to advance its sync.

It is possible that RDS, based on the ACK invoice event, detects that there are some archival sequences (between the sync and highest sequence number in the ACK invoice event) which were not explicitly acknowledged by the entity using ACK invoice events. RDS then assumes that these events were lost in transit to the entity. RDS also checks to see if, based on the current invoice event, the sync associated with the entity can indeed be advanced. The sync associated with an entity is advanced up until the point at which the sequencing information contained in the ACK invoice is lower than that of the detected "missed" event.

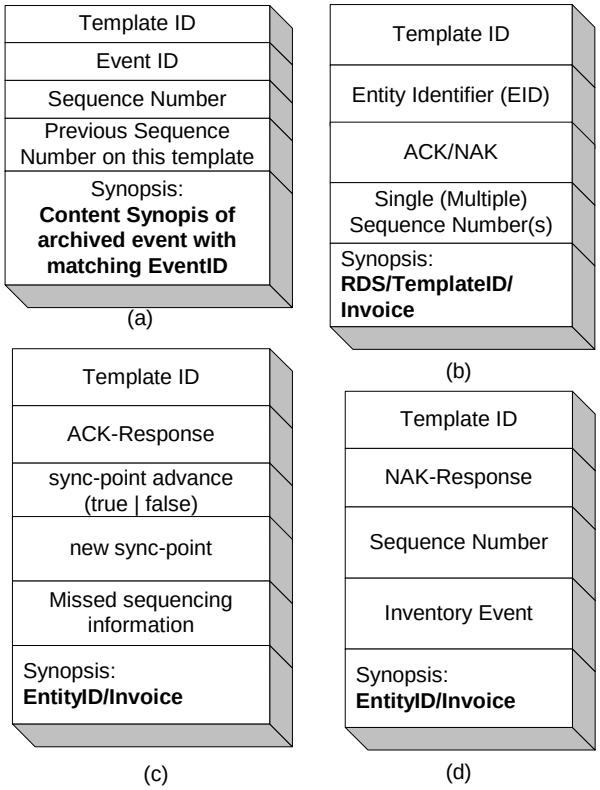


**Figure 3: Archival notifications and such ...**

After the detection of missed sequences RDS issues an ACK-Response invoice (Figure 3.(c)), which contains

information pertaining to the entity's sync advancement (if it is indeed possible) and also the sequencing information corresponding to the "missed" template events. It is entirely possible that the ACK invoice events may have been lost in transit and that the entity may actually have indeed received these events.

RDS does not pro-actively retransmit the inventory event based on the missing sequences. There are two main reasons for this. First, it is possible that the template event(s) are in transit or that just the ACK invoice event was lost. Second, the retransmission costs may be prohibitive with increases in payload sizes.

Upon receiving an ACK-Response invoice event, the entity gains information regarding the archival sequences that it missed. To retrieve events corresponding to these archival sequences, entity has to issue a NAK invoice event requesting the missed event(s). The NAK invoice event contains sequencing information pertaining to the "missing" template events. Upon receipt of this NAK invoice at a RDS, the service retrieves the inventory event corresponding to this sequence number and proceeds to create the recovery event depicted in Figure 3.(d). A recovery event includes information contained in both the template event and the correlated archival notification that was issued in the wake of its archival.

A subscribing entity can detect that it has missed either the template event or the archival notification detailing sequencing information for a given template event or both. An entity can issue a NAK invoice to retrieve sequencing information regarding a template event (with id eventID) that it has previously received. If both the event and the archival event are lost, upon receipt of an ACK-Response invoice event an entity knows the sequences that it has missed. To retrieve events the entity has to issue a NAK invoice event requesting the missed event(s).

The invoice events might themselves be lost in transit due to failures either at the client or en route to RDS or vice versa. The only way an entity will not be routed a template event that it was supposed to receive is if the sync is advanced incorrectly at RDS. However, syncs corresponding to an entity (for a specific managed template) are never advanced (at RDS) until it is confirmed that the entity has indeed explicitly acknowledged receipt of events up until that advancement point. As is clear from our discussions these sync advancements can sustain losses of invoice events.

## 6 Entity Recovery

When a entity reconnects to the broker network after failures or a prolonged disconnect. It needs to retrieve the template events that were issued in the interim and those that were in transit prior to the entity leaving. The recovering entity issues a recovery request for every reliable template that it is interested in. The structure of the recovery request is depicted in Figure 4.(a), these requests are targeted to RDS managing one or more of the templates in question.

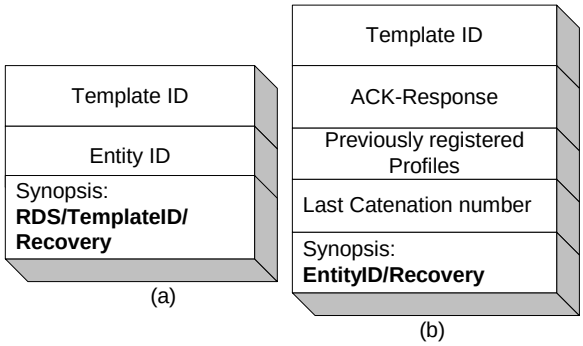


**Figure 4: Recovery requests**

Upon receipt of the recovery request, RDS scans the dissemination table starting at the sync associated with the entity (based on the EID contained in the request). RDS then generates an ACK-Response invoice event outlining the archival sequences, which the entity did not previously receive. In this scheme the entity is not required to maintain any information pertaining to its sync or the archival sequences that it had previously received on a given template. Subscribing entities are also automatically registered to all profiles that they were previously registered to. Publishing and subscribing entities are automatically notified of their last catenation and sync-advances on the specified templateID. The recovery response is depicted in Figure 4.(b).

The ACK-Response contained in the recovery response is processed to advance sync points and to initiate retransmissions as outlined earlier. Failures can take place even during this recovery process and the scheme can sustain the loss of both the recovery requests/responses.

## 7 Advantages & Applications

In this scheme since we do not maintain any state in the individual brokers, recovery does not involve state reconstructions. In fact brokers can fail and remain failed forever. The system will work even if there is just a single broker within the broker network. The scheme does not make any assumption regarding the underlying storage type. The storage structure makes it easier to migrate management of individual templates to different RDS instances. The failure of an RDS affects only those entities whose template(s) are being managed by the failed RDS in question. We do not place any restrictions regarding the placement or number of RDS' available within the system.

The scheme outlined in this paper can be easily extended to exactly-once ordered delivery by ensuring that delivery is allowed only upon receipt of a sync-advance and only if this sync-advance is greater than the sync currently at the entity.

Several NaradaBrokering applications utilize the reliable delivery service provided within system. Additionally we have augmented GridFTP to exploit this feature. Here, we had a proxy collocated with the GridFTP client and GridFTP server. This proxy, a NaradaBrokering entity, utilizes NaradaBrokering's fragmentation service to fragment large payloads (> 1 GB) into smaller fragments

and publish fragmented events. Upon reliable delivery at the server-proxy, NaradaBrokering reconstructs original payload from the fragments and delivers to the GridFTP server. Details of this application, demonstrated at SuperComputing'04 can be found in [27].

## 8 Experimental Results

In this section we include results from our performance measurements. We performed two sets of experiments involving a single broker and three brokers. In each set we compared the performance of NaradaBrokering's reliable delivery algorithms with the best effort approach in NaradaBrokering. Furthermore, for best effort all entities/ brokers within the system communicate using TCP, while in the reliable delivery approach we had all entities/brokers within the system communicate using UDP.

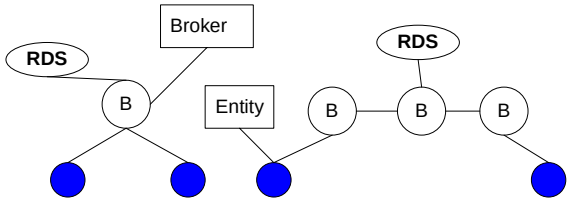


**Figure 5: Experimental Setups**

The experimental setups are depicted Figure 5. The lines connecting entities signify the communication paths that exist between the entities; this could be a connection oriented protocol such as TCP or a connection less one such as UDP.The publishing/subscribing entities (hosted on the same machine to accounting for clock synchronizations and drifts), brokers and RDS are all hosted on separate machines (1GHz, 256MB RAM) with each processes running in a JRE-1.4 Sun VM. Currently, in the RDS we support flat-file and SQL based archival. The results reported here are for scheme where the RDS utilizes MySQL 4.0 for storage operations. We found that the archival overheads were between 4-6 milliseconds for payloads varying from 100 bytes to 10 KB.

We computed the delays associated with the delivery of best-effort and reliable delivery schemes. The results reported here for the reliable delivery case correspond to the strongest case where the event is not delivered unless the corresponding archival notification is received. Figure 6 and Figure 7 depict the transit delay and standard deviation associated with a single broker network, while Figure 8 and Figure 9 depict the same for the 3 broker network.

In the reliable delivery case there is an overhead of 4-6 milliseconds (depending on payload size) associated with the archival of the event, with an additional variable delay of 0-2 milliseconds due to *wait()-notify()* statements in the thread which triggers archival. These factors, in addition to retransmissions (NAKs) triggered by the subscribing entity due to lost packets, contributed to higher delays and higher standard deviations in the reliable delivery case.

It should be noted that we can easily have an optimistic delivery scheme which does not wait for archival notifications prior to delivery. This scheme would then produce overheads similar to the best effort case.

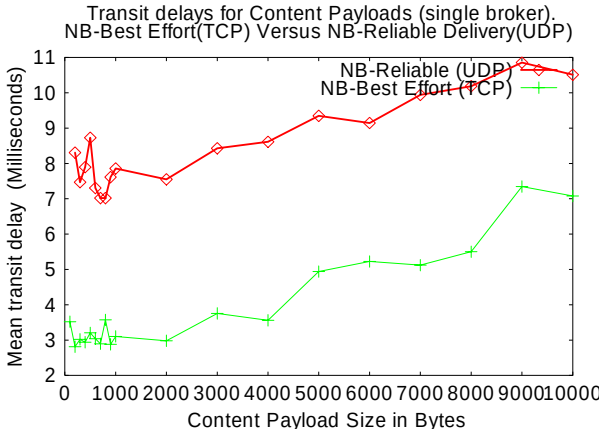


**Figure 6: Transit Delay comparison (single broker)**

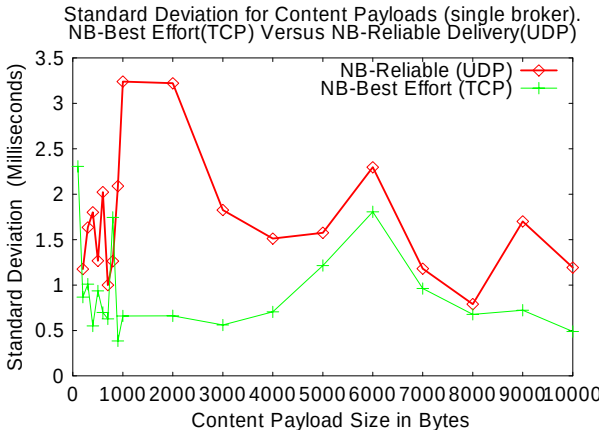


**Figure 7: Standard deviation comparison (1 broker)**

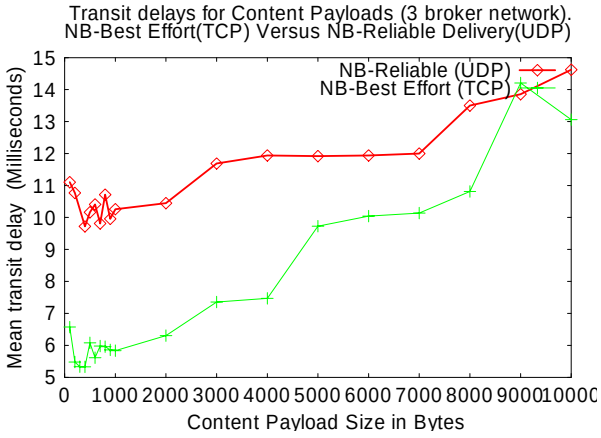


**Figure 8: Transit Delay comparison (3 brokers)**

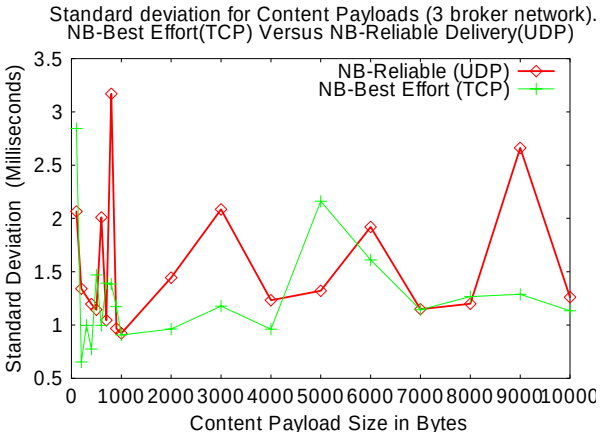


**Figure 9: Standard Deviation comparison (3 brokers)**

## 9 Replicated RDS nodes

In the previous sections we outlined our strategy to ensure reliable delivery. In this scheme if there is a failure at the RDS node the entities need to wait for this RDS node 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 RDS failures. To achieve this we include support for multiple replications of the RDS node. The scheme does not impose any restrictions on the number of replicas, and also includes strategies for the addition and removal of replicas. The scheme can sustain the loss of multiple replicas, so long as there is at least one RDS node available reliable delivery guarantees will be met.

In this scheme there would be multiple instances of the RDS node, with each RDS replica making decisions autonomously. Please note that this scheme does not have the concept of primary or secondary replicas: each replica is just as good as the other and can make decisions independently. Additional replicas can be added for greater fault-tolerance and redundancy. Similarly, replicas can be gracefully removed from the system. Entities, publishers and subscribers alike, can interact with any of these replicas through the broker network and the reliable delivery guarantees will be met. Furthermore, these replicas can fail during these interactions, and the entities will discover another valid replica and the reliable delivery guarantees will continue to be met. At any time (including after disconnects and failures) entities can change the replicas they interact with. Finally, it should be noted that every replica keeps track of events received (or missed) by individual entities or other replicas.

Every replica publishes and subscribes to a set of topics to ensure communications and consistency with each other. Please see Table 1 for a compendium of the topics and the primary purpose of the exchanges over the corresponding topics. These topics are in addition to the topics (outlined in sections 4, 5 and 6) that an RDS node publishes/subscribes to facilitate interactions with the entities.

**Table 1: Topics for exchanging information between replicas**

| Topic | Function |
|---|---|
| RDS/Replica/Template-ID/StoredEvents | Information regarding events issued by a publisher and stored at a specific replica are issued on this topic. |
| RDS/Replica/Template-ID/Invoice. | Acknowledgements (positive and negative) related to events stored at a replica are issued on this topic. |
| RDS/Replica/Template-ID/Info | Information regarding the addition and removal of replicas are exchanged on this topic. |
| RDS/Replica/Template-ID/EntityUpdates | Information regarding the addition and removal of entities are issued on this topic. |

Please note that in our scheme each replica is autonomous and thus maintains its own sequencing information. This implies that a given event, published by a publisher, MAY have different sequence numbers at different replicas. Furthermore, the sync-points associated with subscribing entities will also be different on different replicas. Our scheme copes with both these scenarios. As in our previous discussions it should be noted that a given RDS node may manage more than 1 templates, thus it is possible that multiple RDS replicas may managed a different set of templates in addition to the common template managed by them.

### 9.1 Storage and Order of Events at a Replica

An entity needs to first locate the RDS replica that it will interact with. Typically, this choice is predicated on the proximity of the RDS node and also on the load and computing power available at a given RDS node. The decision to use a specific RDS node is made after receiving responses to a discovery request, checking the usage metrics and subsequent pings to locate the nearest replicas from the set of least-loaded RDS replicas.

Upon receipt of an event from a publisher, the RDS replica performs actions outlined in section 4.4 and proceeds to store the event. It then proceeds to issue an event to the other replicas. The structure of this event is depicted in Figure 10. Among other things this event contains the original published event, the publisher of the original published event and the current catenation number associated with the publisher. This publisher information allows all the replicas to be up-to-date with catenation numbers associated with the publisher and also to retrieve any events that might have been lost in transit to a replica. The topic on which this event is published is RDS/Replica/Template-ID/StoredEvents

Since a replica that issues this event also includes the previous sequence number on a given template-ID any replica $\boldsymbol{R_X}$ can detect events that it was supposed to receive from any other replica $\boldsymbol{R_Y}$. In case a replica $\boldsymbol{R_Y}$ detects such a missed event that it was supposed to receive from $\boldsymbol{R_X}$, it issues a negative acknowledgement to the replica $\boldsymbol{R_X}$ to retrieve this missed event. Any given replica $\boldsymbol{R_Y}$ ensures that its preserves the order of events received from

any other replica $\boldsymbol{R_X}$,. Since a publisher's publishing order is preserved at a replica, it is preserved at all the replicas. It should however be noted that there is no consensus total ordering of events in this scheme. For events $e_a^1, e_a^2, e_a^3$ published by a publisher and $e_b^1, e_b^2, e_b^3$, the order in which they may be stored at a replica $\boldsymbol{R_Y}$ could be $e_a^1, e_a^2, e_b^1, e_a^3, e_b^2, e_b^3$ while the order in which they would be stored at replica could be $e_a^1, e_a^2, e_b^1, e_b^2, e_b^3, e_a^3$.

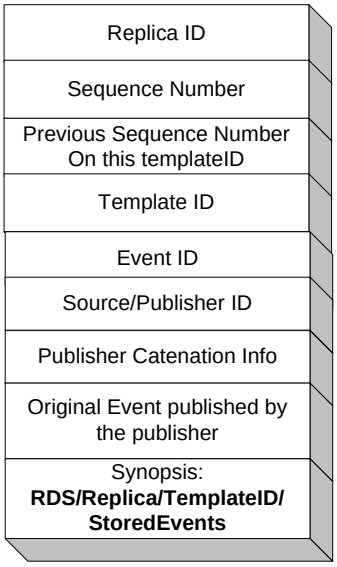


**Figure 10: Event propagated to other replicas**

## 9.2 Processing Replica Acknowledgements

Individual replicas then issue acknowledgements corresponding to the receipt of these events on the following topic: RDS/Replica/Template-ID/Invoice. The structure of this event is depicted in Figure 11.

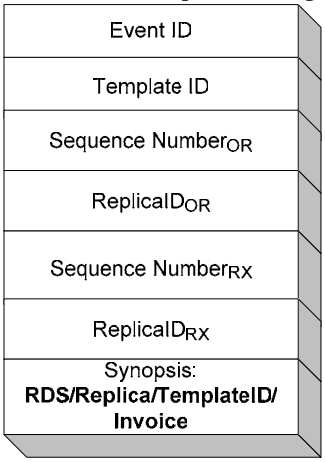


**Figure 11: Structure of invoice from other replicas**

Since all replicas are subscribed to this topic, they all receive this acknowledgement. Note that this acknowledgement contains the ReplicaID of the replica issuing the acknowledgement along with sequence number corresponding to the stored event at this replica. This acknowledgement thus allows all the replicas to be aware of the sequence numbers associated with a specific event (based on the event-ID) at different replicas.

At every replica there are also tables -- one for every other replica --which track the sequence numbers corresponding to a given event at every other replica. Figure 12 depicts a widget encapsulating the information stored at such a table, where a replica $\boldsymbol{R_X}$, maintains information regarding the sequence number associated with an event at another replica $\boldsymbol{R_Y}$. This information allows a replica to recover from any other replica after a failure, since each replica maintains information about events missed (and received) by a given replica.


**Figure 12: Widget with table maintained at replica Rx**

## 9.3 Dealing with subscribing entities

Next we discuss subscribing entities. Subscribers receive real-time events published by the publishers. Upon receipt of an event at a replica (either from a publisher or propagated by other replicas), the replica creates an archival notification and publishes it to the topic which replica-subscribers are listening to. Upon receipt of this archival notification, and subsequent acknowledgement the sync-point associated with this subscriber is advanced. Since every RDS node subscribes to the topic on which subscribing entities issue acknowledgements, each RDS replica independently maintains sync-points for a given subscribing entity. Even though the acknowledgements corresponds to sequence numbers at a certain replica $\boldsymbol{R_X}$, another replica $\boldsymbol{R_Y}$ can map this to the appropriate sequence number associated with the same event at $\boldsymbol{R_Y}$. Furthermore, if a subscribing entity is interacting with a replica $\boldsymbol{R_X}$, this replica also notifies other replicas of sync-advancements; this is useful to deal with any losses in the receipt of acknowledgements from the subscribing entity. This allows every replica to update the sync-point associated with the entity-ID in question. Note that the sync-points will be different at different replicas.

The interactions that entities – publishers and subscribers alike – have with individual replicas are identical to what we described in the earlier sections (4.3, 4.5 and 5.0). In fact, besides the selection of the RDS replica as part of the bootstrap, operations at these entities are identical to those in place for a single RDS node. We now proceed to outline strategies for dealing with the addition of replicas, and dealing with the failure of replicas.

## 9.4 Addition of a replica

When an RDS replica $\boldsymbol{R_N}$ is added to the system, the replica needs to subscribe to topics related to Reliable delivery over a specific template-ID in addition to the topics which an RDS replica needs to subscribe to. The

newly-added RDS node then proceeds to notify its addition to the other replicas; it does this by also including its replica-ID in the notification which is sent over the topic RDS/Replica/Template-ID/Info. Next it issues a message to the other replicas to retrieve the number of events corresponding to a specific template-ID. All the RDS replicas respond with $T_N$ corresponding to the number of stored template events. Based on these responses the newly-added replica selects an assisting replica $\boldsymbol{R_A}$ to retrieve past events.

The replica $\boldsymbol{R_N}$ will not respond to discovery requests initiated by publishers and subscribers alike until such time that the recovery process is complete. The replica $\boldsymbol{R_N}$ sets aside $T_N$ successive sequence numbers to recover from replica $\boldsymbol{R_A}$. The replica $\boldsymbol{R_N}$ will however continue to store real-time events and proceed to process acknowledgements from other replicas to the real-time events. The replica also proceeds to retrieve information regarding replica-sequence tables for a given event-ID of a specific template-ID from the assisting replica $\boldsymbol{R_A}$.. Note that during the recovery process the replica $\boldsymbol{R_N}$ also issues acknowledgements corresponding to events that it is storing. This enables every replica to know which events are still missing; this is especially useful if the assisting replica $\boldsymbol{R_A}$. fails in which case one of the other replicas can take over.

### 9.5 Replica Recovery from failures

When an RDS replica $\boldsymbol{R_F}$ recovers from a failure, it issues a message to the other replicas. Based on the responses, the recovering replica chooses the best-available replica to recover from. Since each replica is aware of the events missed by a given replica, each replica responds with the number of "missed" events. The recovering replica chooses the replica which reports the highest number of missed events and proceeds to also store real-time events. If the assisting replica fails, other replicas can assist in the recovery from failures.

### 9.6 Dealing with replica failures

It is conceivable that an RDS replica $\boldsymbol{R_F}$ can fail. If no events are currently being published and at least one replica is available no further actions need to be taken. However, if an entity was utilizing this RDS replica $\boldsymbol{R_F}$ by either publishing events or recovering from it there are actions that need to be taken. In the case of publishers, once it detects that the replica is unresponsive it issues a discovery request to retrieve a new replacement replica $\boldsymbol{R_R}$. The publisher then proceeds to inform this replica $\boldsymbol{R_R}$.of its catenation number. It is possible that there is a mismatch. The only mismatch possible is that the catenation number $C_R$ maintained by the replacement replica $\boldsymbol{R_R}$ is lower than the one reported by the publisher $C_P$. If such a mismatch exists a request is issued to retrieve the events corresponding to the catenation numbers between $C_P$ and $C_R$. If this is available the replica which contains this information responds with the necessary information.

If a subscribing entity detects a failure in the replica that it is interacting with, it discovers a replacement replica $\boldsymbol{R_R}$ and supplies it with its latest sync-point corresponding to the failed replica $\boldsymbol{R_F}$. This replacement replica then proceeds to update its sync-point associated with this entity. This is done to account for 2 possible scenarios. First, it is possible that the sync-point advancement from replica $\boldsymbol{R_F}$. was lost in transit. Second, it is possible than replica failed before it could issue a sync-point advance, since replica $\boldsymbol{R_R}$ keeps track of acknowledgements from the entity it is also aware of what the true sync point of the entity should be. Once it has received the new sync-point corresponding to the replica $\boldsymbol{R_R}$, the entity proceeds with operations as outlined before.

A replica contains some state $\boldsymbol{S}$; for a replica to leave the system permanently the only requirement is that this shared state $\boldsymbol{S}$ be propagated to at least one other replica. This shared state corresponds to the event published by a publisher that is interacting with it and the catenation number associated with this publisher. Note that this information is propagated as outlined in Figure 10. What this means is that when an event published by a publisher is received at a replica, the stored event and the accompanying catenation numbers should be propagated to other replicas. This propagation if it is made to even one replica ensures guaranteed delivery.

### 9.7 Dealing with the entity additions/removals

If an entity-ID is registered with a replica it should be registered with all the replicas. Upon addition of an entity-ID at a given replica, the replica in question issues a message to the following topic RDS/Replica/Template-ID/EntityUpdates. The structure of the event carrying this information is depicted in Figure 13.

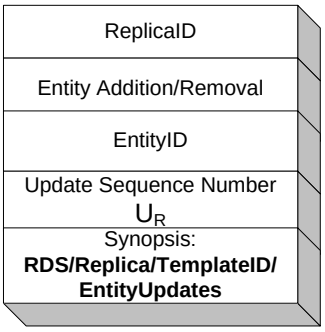


**Figure 13: Entity Update Event for Replicas**

The update-SequenceNumber allows one to detect if updates from a given replica were ever lost. A similar scheme is in place to deal with the de-registration of entities from a specific template-ID.

## 10 Support for WSRM in NaradaBrokering

Recently we have incorporated support for WSRM (WS-ReliableMessaging) within NaradaBrokering. WSRM

is a specification from IBM and Microsoft which facilitates reliable communications between two Web Service endpoints. In WSRM, service endpoints establish a sequence identifier and proceed to ensure reliable delivery of SOAP messages from the source to the sink over this identifier. We have deployed our WSRM implementation in two modes viz. as a handler/filter which provides incremental addition of capabilities at a Web Service and also as a proxy which provides reliable delivery capabilities. We now report some results from our implementation to give an idea of the costs involved. Since it is an XML-based specification, the most important cost is the cost related to performing various XML related operations such as creation of requests, addition of protocol elements to the SOAP message and generation of faults. Furthermore, like several WS-* specifications, WSRM leverages WS-Addressing (WSA). Incorporation of WSA mandates that any SOAP message that is under the purview of the WSRM specification needs to be parsed to retrieve all the WSA headers first. Among the WSA headers are the source and destination information – which is a precursor to any kind of error detection, message identifiers and information regarding where protocol errors need to be reported to. In our implementation (also available for download) we have leveraged XMLBeans as the schema compiler. EnvelopeDocument corresponds to the XMLBeans representation of the SOAP Envelope.

The experiments were performed on a 3.5 GHz Pentium IV machine with a JRE 1.4.2 virtual machine. Each test data was computed from an average of 100 runs. The times reported here are all in microseconds. These results are reported here to enable the reader to discern the costs involved in using WSRM.

**Table 2: Costs involved in WSRM**

| Operation | Mean | Standard Deviation | Standard Error |
|---|---|---|---|
| Create an XMLBeans based Envelope Document | 121.29 | 25.77 | 2.65 |
| Create an Axis based SOAPMessage | 85.76 | 79.36 | 8.22 |
| Convert an EnvelopeDocument to a SOAPMessage | 3503.81 | 758.48 | 80.85 |
| Convert SOAPMessage to EnvelopeDocument | 730.08 | 392.35 | 41.58 |
| Create a WS-Addressing Endpoint Reference (EPR) (Contains just a URL address) | 84.61 | 25.61 | 2.67 |
| Create a WS-Addressing Endpoint Reference with ReferenceProperties | 133.13 | 35.64 | 3.71 |
| Create an Envelope targeted to a specific WSA EPR | 157.98 | 12.19 | 1.27 |
| Create an Envelope targeted to a specific WSA EPR with most WSA message information headers | 263.20 | 35.73 | 3.74 |
| Parse an EnvelopeDocument to retrieve WSA Message Info Headers | 711.74 | 231.61 | 23.76 |
| Create a Wsrm Fault | 413.80 | 239.17 | 25.07 |
| Create a Wsrm SequenceRequest | 268.95 | 37.93 | 3.97 |
| Create a Wsrm SequenceResponse | 234.97 | 17.40 | 1.81 |
| Create a Wsrm SequenceDocument | 43.8125 | 2.99 | 0.30 |
| Add a WsrmSequenceDocument to an existing envelope. (Contains sequence identifier and message number) | 13.01 | 0.57 | 0.05 |
| Create a WSRM SequenceAcknowledgement based on a set of message numbers | 461.17 | 172.40 | 18.27 |
| Create a WSRM TerminateSequence | 20.95 | 1.30 | 0.13 |

We are also planning to incorporate support for WSRM within the replicated RDS infrastructure. Sequence identifiers would be managed templates within such an infrastructure. This would make our fault tolerant scheme accessible to regular Web Services.

## 11 Conclusions & Future Work

In this paper we described our scheme for the reliable delivery of events in the presence of node and link failures. This feature has been exploited by native NaradaBrokering applications and also been used to augment third party applications such as GridFTP. We are currently implementing the replicated RDS strategy. This paper, if accepted, will also incorporate results within replicated RDS settings involving multiple replicas. The replicated RDS scheme has several advantages, chief among them is the ability to withstand multiple RDS failures and the ability to increase or decrease the number of replicas dynamically depending on the system requirements.

## 12 References


[1] Shrideep Pallickara and Geoffrey Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of ACM/IFIP/ USENIX International Middleware Conference. 2003.

[2] Geoffrey Fox and Shrideep Pallickara. Deploying the NaradaBrokering Substrate in Aiding Efficient Web & Grid Service Interactions. Special Issue of the Proceedings of the IEEE on Grid Computing. Vol 93, No 3. pp 564-577. March 2005.

[3] Geoffrey Fox, Shrideep Pallickara, Marlon Pierce, Harshawardhan Gadgil. Building Messaging Substrates for Web and Grid Applications. (To appear) in the Special Issue on Scientific Applications of Grid Computing in the Philosophical Transactions of the Royal Society of London 2005.

[4] Shrideep Pallickara and Geoffrey Fox. On the Matching Of Events in Distributed Brokering Systems. Proceedings of IEEE ITCC Conference on Information Technology. April 2004. Vol II pp 68-76.

[5] Shrideep Pallickara and Geoffrey Fox. A Scheme for Reliable Delivery of Events in Distributed Middleware

Systems. Proceedings of the IEEE International Conference on Autonomic Computing. New York, NY. pp 328-329.(Short Paper).
[6] Kenneth Birman. The process group approach to reliable distributed computing. Communications of the ACM, 36(12):36–53, 1993.
[7] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Ithaca, NY, 1994.
[8] Kenneth Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. Technical Report TR 93-1390, Cornell University, 1993.
[9] Kenneth Birman and Keith Marzullo. The role of order in distributed programs. Technical Report TR 89-1001, Dept. Of Computer Science, Cornell University, NY 14853, 1989.
[10] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the "no partition" assumption. Proceedings of the Workshop on Future Trends of Distributed Systems, 93.
[11] Kenneth Birman. Replication and Fault tolerance in the ISIS system. In Proceedings of the10th ACM Symposium on Operating Systems Principles, pages 79–86, 1985.
[12] R Renesse, K Birman, and S Maffeis. Horus: A flexible group communication system. In Communications of the ACM, volume 39(4). April 1996.
[13] D Dolev and D Malki. The Transis approach to high-availability cluster communication. In Communications of the ACM, volume 39(4). April 1996.
[14] Spinglass: Secure and Scalable Communications Tools for Mission-Critical Computing. K. Birman, R van Renesse and W Vogels. International Survivability Conference and Exposition. DARPA DISCEX-2001, CA, June 2001.
[15] Fred Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. In ACM Computing Surveys, volume 22(4), pages 299–319. ACM, 1990.
[16] R. Boichat, P. Th. Eugster, R. Guerraoui, and J. Sventek. Effective Multicastprogramming in Large Scale Distributed Systems. Concurrency: Practice and Experience, 2000.
[17] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, J. Auerbach: Exactly-once Delivery in a Content-based Publish-Subscribe System. DSN 2002: 7-16
[18] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In Proceedings of ACM PODC, pages 219–227, USA, July 2000.
[19] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In Proceedings AUUG97, pp 243–255, Australia, 1997.
[20] The IBM WebSphere MQ Family. http://www-3.ibm.com/software/integration/mqfamily/
[21] Object Management Group, Fault Tolerant CORBA Specification. OMG Document orbos/99-12-08 edition, 99.
[22] Object Management Group, Fault Tolerant CORBA Using Entity Redundancy RFP. OMG Document orbos/98-04-01.
[23] B. Natarajan, A. Gokhale, D. Schmidt and S. Yajnik. "DOORS: Towards High-performance Fault-Tolerant CORBA", Proceedings of International Symposium on Distributed Objects & Applications (DOA), Belgium, 2000.
[24] P. Narasimhan, et al. Using Interceptors to Enhance CORBA. IEEE Computer 32(7): 62-68 (1999)
[25] Michel Cukier et al. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. Symposium on Reliable Distributed Systems 1998: 245-253.
[26] Web Services Reliable Messaging Protocol (WS-ReliableMessaging) March, 2003. From IBM, Microsoft etc.
[27] G. Fox, S. Lim, S. Pallickara and M. Pierce. Message-Based Cellular Peer-to-Peer Grids: Foundations for Secure Federation and Autonomic Services. Journal of Future Generation Computer Systems. Volume 21, Issue 3, pp 401-415 (March 2005).