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

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

Abstract

*Reliable delivery of messages is an important problem that needs to be addressed in distributed systems. In this paper we briefly describe our basic 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.*

1. Introduction

In this paper we present a scheme for the fault-tolerant, 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. In [1] we presented a scheme for the reliable delivery of messages in the presence of node/link failures and unpredictable links. Subscribers retrieve messages issued over the reliable-topic during the subscriber's absence (either due to failures or intentional disconnects). In this paper, we extend this basic scheme to incorporate support for multiple autonomous repositories, thus providing greater redundancy & fault tolerance during reliable delivery.

We have implemented the scheme described in this paper within the context of the NaradaBrokering substrate [2], which is based on the publish/subscribe paradigm. The NaradaBrokering substrate comprises a set of cooperating router nodes known as *broker*s. 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. 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 (in our basic scheme) a given reliable-topic can only be managed by exactly one repository.

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) so that messages and other information pertinent to the reliable delivery algorithm can be stored. A repository stores messages issued over its managed reliable-topics 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

Fourth International Conference on Autonomic Computing (ICAC'07)
0-7695-2779-5/07 $20.00 © 2007 IEEE

also has to compute the intended destinations and ensure the reliable delivery of the stored message to the computed destinations. Figure 1 summarizes interactions in the basic reliable delivery scheme.

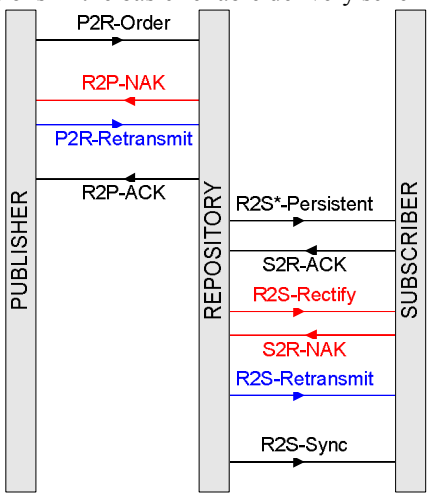

Figure 1: Interactions between entities

The reliable delivery algorithm involves communications between various entities through the exchange of *control- events*, 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`. We use 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`. Multiple destinations are indicated by **bold-face** font.

3. Repository redundancy

In our basic reliable delivery 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 autonomous repositories –- constituting a *repository-bundle* –-for a given reliable-topic; topics managed by these repositories need not be identical. A repository may 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 a repository is available.

Besides additional redundancy, and the accompanying fault-tolerance, a highly-available, distributed repository scheme enables clients to exploit geographical and network proximities. *Closer* repositories ensure reduced latencies.

3.1. 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 *active* mode by initiating error-corrections & retransmissions. *Passive* mode repositories 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 2. In the passive mode, a repository listens to all events initiated by the publishers and subscriber; however, the repository will not issue control events to clients that it does not steer. Operating in the passive mode, allows a repository to take over as the steering repository for any client.

When a client is ready to initiate reliable communications, it designates a steering repository from the set of repositories within the repository-bundle associated with the reliable-topic. This is based on network proximity by computing network round-trip delays to the repositories. The client then issues an event over the repository's communications-topic designating it as the steering repository.

3.2. Ordered storage of published messages

For every published message, the publisher issues a `P2R-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.

3.3. 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. 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 (local ordering) associated with a publisher is identical at every repository within the repository-bundle.

A repository computes destinations associated with every published message based on the registered subscriptions for the reliable-topic. 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 in the original message.

3.4. 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.

3.5. Gossips between repositories

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

3.5.1 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 related to persistent storage of messages. Each entry contains the publisher identifier, the catenation number, the message identifier and the sequence number assigned by the repository.

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 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. Tracking messages not received by other repositories allows a repository to support repository recovery.

3.6. 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 all subscribers. 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.

3.7. Dealing with repository failures

Once a client detects a repository failure, it discovers a new steering repository. A publisher exchanges information about its catenation number with the replacement 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 another repository.

3.8. 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 repository first retrieves updates to the list of registered clients and subscriptions. 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, and the corresponding dissemination list, from the assisting repository. 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 are retrieved. Second, it registers to start receiving messages published in real-time.

3.9. Addition/Removal of a repository

Addition of a repository is very similar to the recovery process described in the preceding section. 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. 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 simply issues a Gossip-LEAVE event.

4. Experimental Results

For our benchmarks, we setup a broker network comprising 3 brokers hosted on different machines. In Topology A, we measured the costs for best effort delivery. In topology B, we setup a repository at the intermediate broker. In topology C, we setup repositories (constituting a bundle) at each of the brokers. All processes executed within Sun's Hotspot™ JVM 1.6 on Linux (2.4.22). In each case, we had 20 subscribers. Also, there is a publisher and a measuring subscriber, which reported the delays involved in communications. The publisher and subscriber were hosted on the same machine, and were connected to different brokers, with the publisher being 3 broker hops away from the measuring subscriber. Machines involved in the benchmark have the following profile: 2 CPU (Quad Core, 2.4 GHz), 8 GB RAM on a 100 Mbps LAN. MySQL 5.0 was used.

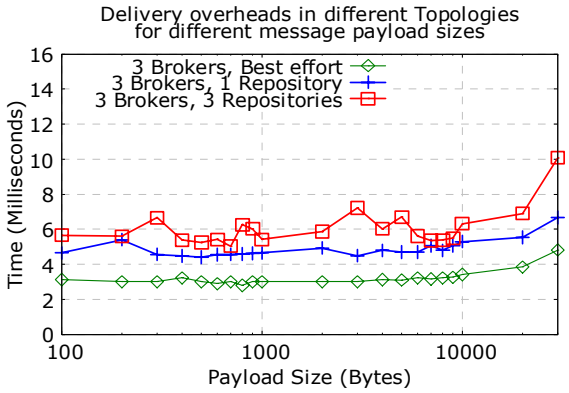

Figure 2: Reliable Delivery Overheads

The overheads (see Figure 2) involved in reliable delivery pertain to the cost of persisting a message to storage, and for the corresponding persistence event to traverse from the repository to the subscriber. The results also demonstrate that the overheads introduced by our repository redundancy scheme are acceptable, since it is identical to the single repository case.

5. Conclusions & Future Work

The scheme for the fault-tolerant, reliable delivery of messages in publish/subscribe systems presented here introduces acceptable overheads. As part of our future work we plan to research issues related to repository placement schemes to ensure that average latencies for reliable communications are reduced.

6. Acknowledgment

This research is supported by a grant from the National Science Foundation's Division of Earth Sciences project number EAR-0446610.

References

[1] S. Pallickara and G. Fox. A Scheme for Reliable Delivery of Events in Distributed Middleware Systems. Proceedings of the IEEE International Conference on Autonomic Computing. NY. pp 328-329. 2004.

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