Combating NACK Storms and Slow Consumers

Robust Reliable Communication with
RTI Data Distribution Service

Rick Warren
Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089

For further information, please contact:
info@rti.com
+1 (408) 990-7400
www.rti.com

December 7, 2009

Abstract

Reliable one-to-many communication is frequently prone to two serious problems in particular: (*1*) how to prevent a slow consumer from holding up the rest of the system, and (*2*) how to prevent massive amounts of negative acknowledgement (NACK) traffic from swamping the network. These problems are related to one another: both deal with the way in which a communications stack (network protocols combined with a middleware on top of them) maintains reliability across a logical network topology with broad fan-out.

This paper discusses how these problems can be lessened or avoided altogether by leveraging the unique capabilities of RTI Data Distribution Service middleware.

Table of Contents

Introduction to One-to-Many Reliability

NACK storms and slow consumers can plague any reliable one-to-many communications system. To understand why, it's important to understand how reliable protocols typically work in such scenarios. The following is a basic description of reliability in RTI middleware. Although there can be variations—for example, an alternative implementation might interpose brokers between the producer and consumer—many of the concepts and interactions described below hold true for any reliable protocol[1].

When a producer publishes data to a set of consumers, it typically also sends (*a*) "heartbeats" informing the consumers that the producer is still functioning and (*b*) status notifications indicating which data is available from the producer. (RTI combines these functions into a single heartbeat message; the remainder of this paper will assume this design.) Therefore, relative to consuming applications, one of the following things will eventually happen:

- The consumer will receive one or more messages from the producer. By examining the sequence numbers of these messages, the consumer can determine whether any previous messages were dropped.
- The consumer will receive meta-information from the producer indicating that some messages have not arrived.
- The consumer will receive nothing from the producer. It will eventually time the producer out and report an error to the application.

In response, a well-behaved consumer will typically send the producer positive acknowledgements ("ACKs"), indicating the messages it has received, and/or negative acknowledgements ("NACKs") indicating messages that were missed[2].

At this point, one of the following will occur on the producer side:

- The producer will receive ACKs for the data it has sent. Provided that it has no requirement to maintain data for late joiners to the network, it can delete messages from its send queue as soon as all consumers have acknowledged them.
- The producer will receive NACKs for one or more messages. It will respond by resending the missing data.
- The producer will receive no response at all from one or more consumers. It will eventually time-out the offending consumers and report an error to the application.

[1] Much of the description holds true for TCP as well, even though it is based on a stream metaphor instead of explicit datagrams, because it is implemented on top of the datagram-oriented IP. TCP hides the heartbeat and acknowledgment details from the application, but analogous behavior nevertheless takes place.

[2] RTI supports optional ACK suppression to reduce CPU and bandwidth utilization in high fan-out configurations. The description that follows encompasses both positive and negative acknowledgements. For more information about how ACK suppression works, and when it might be appropriate, see Windowed Reliability below.

In this way, producers repeatedly write new messages to a group of consumers and the consumers report back as to whether or not they have received those messages. As messages become fully acknowledged, the producer may discard them.

RTI's Solution

RTI middleware provides an integrated messaging and caching infrastructure. In basic message delivery, "live" messages are delivered directly to the application without brokers or context switches for minimal latency.

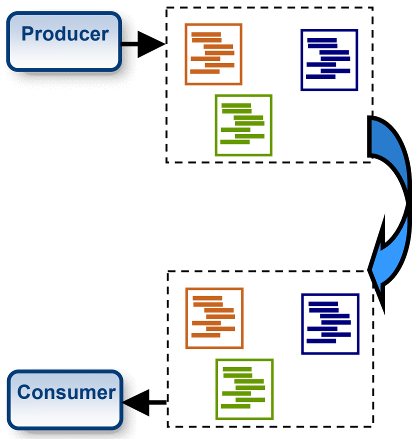

Figure 0 RTI architecture

In-memory data caches, on both the producer and consumer sides, support fine-grained control over the degree of reliability required. Persistence components elsewhere in the architecture, combined with these in-memory caches, minimize latency while optimizing the amount of data actually stored. RTI's approach also provides flexibility in the amount of system resources that are consumed.

The more information the application provides to the middleware about the data in which it is interested, and the communication contracts of that data, the more intelligently the middleware can manage and minimize the network traffic that must flow between producers and consumers to fulfill those contracts. Network traffic and CPU loads can be reduced, for example, by exposing filters and time constraints to the middleware, thereby reducing the probability of pathological ACK- or NACK-related traffic patterns. And applications can increase responsiveness—both of automatic behavior adaptation and of application notifications—by carefully configuring heartbeat and acknowledgement rates and timeouts.

These concepts and others are discussed in further detail in the following sections.

Problem: Slow Consumers

Unfortunately, problems can occur if one or more consumers are not able to respond to the producer in a timely manner. If a producer's send queue is full and it has not received a response from a particular consumer, it has only a few choices:

- **Don't expect acknowledgements** in the first place. A message producer can inform its consumers that they don't need to provide positive acknowledgements when they receive messages, just negative acknowledgements when they *don't* receive something. This technique efficiently isolates the producer from slow consumers, but is only appropriate when the producer and consumer are loosely coupled and windowed reliability (see below) is sufficient.
- **Enlarge the queue.** This tactic can be a good one initially, but cannot continue indefinitely.

- **Make room in the queue by discarding data** that has not yet been fully acknowledged. This action puts reliable delivery at risk for all other consumers, *beca*use if a consumer later NACKs a discarded message, the producer will be unable to repair the missing data.
- **Stop waiting for acknowledgements** from the slow consumer. Doing so may amount to failing the consumer over to a best-effort mode—simply not waiting for acknowledgment before flushing sent data from the queue—or, even more severe, refraining from sending future messages to the consumer altogether. This tactic puts reliability at risk, but only for the offending consumer(s).

Avoidance Strategies

The best way to handle this problem is, of course, to avoid it in the first place. In large part, that means keeping packets off the wire if they are not needed by the consumer(s) or likely to be dropped en route.

Help Me Help You

Producers and consumers can work together to help consumers keep up. The surest way to accomplish this is to avoid burdening consumers with unnecessary information. At design time, a variety of filters can be specified to reduce the data that is sent to a consumer or class of consumers to control the impact of a small number of slow consumers on the majority of consumers.

- **Time-Based Filtering (Data Throttling)**

  For certain types of streaming data and certain consumers, it may not be necessary to receive every message. Such would typically not be the case for market data sent to an algorithmic trading server, of course, but for messages destined for a user interface—which can likely only be updated a few times a few times a second anyway—or for streaming media, time-based filtering may be appropriate.

  RTI allows applications to express time-based filters in terms of a “minimum separation”: a minimum time duration that must elapse between messages. For example, a consumer may express to a producer that it is only able to process one message every 50 microseconds. As a result, the middleware will drop intermediate messages to that consumer only. When possible, these messages will be dropped on the producer side so that they never burden the network.

  The middleware will never impose a time-based filter on a consumer automatically, as the “missing” messages would be unexpected by the application logic and could therefore prove harmful. A minimum separation can be configured at the initialization time of a consumer (or class of consumers), as well as dynamically configured during the live operation of the distributed application. An adaptive application can take advantage of this capability to dynamically adjust time-based filters at run time. For more information about detecting and responding to slow consumers, see the section, *Management Strategies,* below.

- **Content-Based Filtering**

  Content-based filtering is a more widely applicable strategy for reducing the amount of data on the network. RTI provides content-aware delivery of messages, so consuming applications can express which specific data values are of interest and which are not. For example, a consumer may be interested in “Offer” data only when the “price” field contains a value greater than 20. RTI applies this intelligence in the messaging layer so that the consumer is only notified of updates which already meet its specific criteria; this reduces the load on the consumer of processing unnecessary updates.

- **Address Partitioning**

  Traffic on a single logical data stream can be partitioned across a number of physical addresses for load balancing purposes. Modern enterprise-class switches support IGMP snooping, which lets them switch multicast traffic as efficiently as they do unicast. RTI can take advantage of this feature in the hardware to partition traffic efficiently and filter unnecessary data without any network or CPU penalty at the network edge.

Control the Flow to Avoid Dropped Messages

Applications can shape network traffic and avoid dropped messages by controlling the flow of packets onto the network. Like the meter on a freeway entrance ramp, spacing out the traffic may actually improve latency and throughput overall by eliminating costly resends[3].

RTI provides an optional, comprehensive flow control capability for application data. Applications can indicate how often a message producer can send what amount of data, as well as whether unused capacity may “roll over” and be used later[4]; these parameters can be changed dynamically through the RTI APIs at any time, allowing applications to adapt to real-time conditions. These reusable, per-producer flow controller definitions allow a distributed application to shape network traffic with a high degree of precision.

Management Strategies

Despite the best efforts of an application’s designers and implementers, pathological circumstances may cause consumers to fall behind. The first part of this paper summarized the options a producer has when faced with slow consumers:

- Request negative acknowledgements only; suppress positive acknowledgements
- Enlarge the send queue to store more pending data
- Discard unacknowledged data
- Cut off the consumer(s)

RTI provides applications with fine-grained control over all alternatives.

Send Queue Memory Management

Applications can configure how much memory a producer is allowed to use for its send queue initially, as well as how much memory this queue is allowed to consume maximally if unacknowledged data backs up. As the queue fills and then empties again, the producer will automatically adapt the rate at which it sends heartbeats to its consumers: the fuller the send

[3] While flow control can improve worst-case latency—by helping to prevent readers from falling behind—and improve or shape throughput, it does come at the expense of best-case latency, because network sends must take place in an asynchronous thread.

[4] The algorithm is a variation of the well-known “token bucket” pattern. The application has full control over the size of the bucket, the rate of token accumulation, and other parameters.

queue, the more aggressively the producer will spur the consumers to acknowledge the data it has sent. The application can also receive notifications of these changes. This degree of responsiveness and control allows applications to provide resilience and flexibility in the face of fluctuating message volumes, while preventing a slow consumer from overwhelming the memory resources of the producer.

RTI's memory management facility seeks to reduce churn and memory fragmentation and, more importantly, to minimize the number of heap allocations that occur on the critical send/receive path, thereby decreasing latency and increasing determinism. When the send queue grows, by default it will allocate a block of contiguous buffers up front to reduce the probability of future memory allocations. As the queue empties again, these buffers will be retained for later reuse rather than being immediately freed.

Windowed Reliability

RTI gives applications control over which old data can be removed from the send queue when it fills up. These windows of valid data can be defined in terms of time (the maximum "lifespan" between when a message is written and when it should be consumed) and/or space (the "depth" of old messages to be stored in the "history").

If this level of reliability is sufficient, the message producer can be completely isolated from slow consumers by means of *ACK suppression*. In this reliability mode, a producer informs its consumers that they only need to provide NACKs, not ACKs. Because the producer does not expect ACKs from any consumer, a slow consumer cannot affect it. A finite lifespan and/or history depth fulfills the need for send-queue emptying no longer being met by message acknowledgements.

Consumer Inactivation

At some point, a producer can no longer maintain resources on behalf of a consumer that is not keeping up. RTI provides fine-grained control over:

- The rate at which heartbeats are sent from the producer to its consumers.
- The number of heartbeats a producer will send to a consumer without response before marking it as inactive.

A consumer that is inactivated will not be forgotten entirely, but unacknowledged data will not be maintained solely on its behalf; communication will proceed in a best-effort-like mode with respect to that consumer. Should the consumer become responsive again, any data that it missed and that is still available for other reasons will be made available to it.

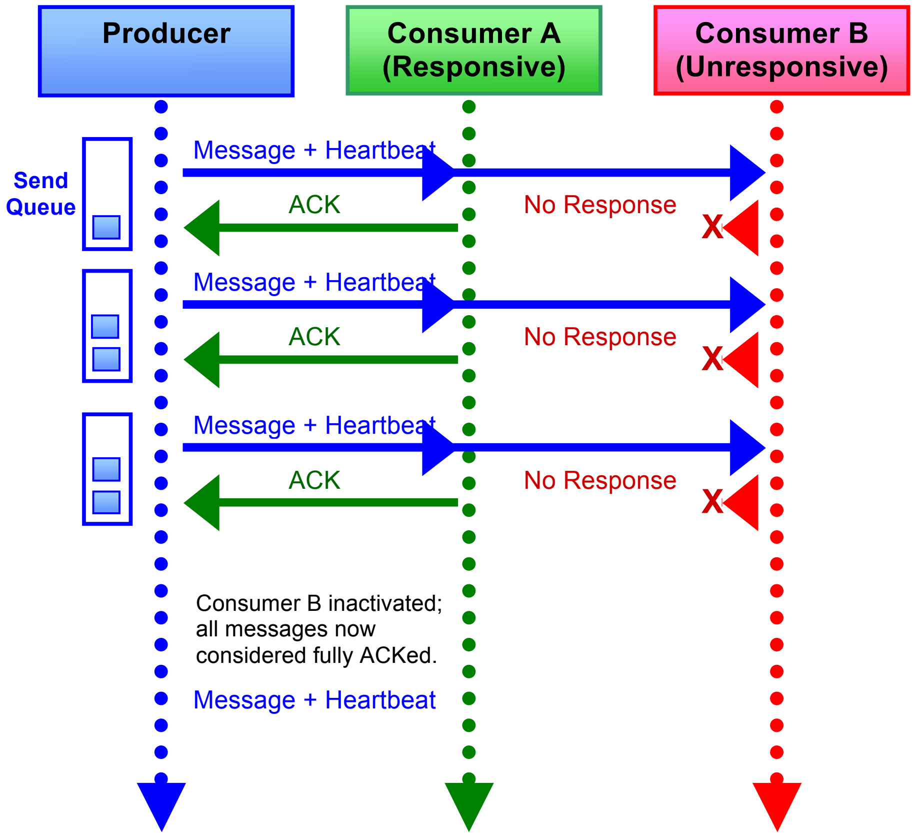

Figure 1 Slow consumer inactivated to clear send queue

As changes in activation and inactivation occur, the application will be notified asynchronously by means of a callback.

In order to provide higher data availability for consumers that fall behind and catch up again, as well as for consumers that may join the network late initially, RTI provides an optional persistence service. This service can be located on any node in the network in order to offload heavy storage requirements from the message producers themselves; service instances can also be federated to provide redundancy and additional levels of data availability. A persistence service interposed between message producers and consumers can seamlessly provide consumers with an arbitrary amount of historical data when they become responsive again.

Problem: NACK Storms

Problems can also occur if consumers respond too promptly. If many consumers miss the same message(s), they may all NACK at once, flooding the network with reliability meta-traffic and preventing application data from flowing.

This problem can be multiplied when using multicast, since resent data will be seen by all consumers, even those that received the previous messages correctly. In the worst case, the processing and storage resources consumed by these unnecessary resends can starve out the processing of new data, leading to a self-perpetuating feedback loop of NACKs and resends ricocheting back and forth across the network.

There are three ways to reduce the damage done by surges in ACK/NACK traffic:

1. Reduce ACK/NACK volumes overall.
2. Smooth NACK spikes to avoid short-term network flooding.
3. Prevent longer-term network flooding caused by poorly targeted NACK responses.

Step 1: Prune and Shape Network Traffic to Reduce (N)ACKs

Some of the strategies for avoiding slow consumers can also help to prevent NACK storms. Specifically, by keeping unnecessary traffic off the network in the first place, the middleware removes the need for a consumer to ACK/NACK it, reducing the probability of a storm. These strategies are discussed in the section *Avoidance Strategies* above.

Step 2: Wait Before Responding to Avoid NACK Storms

Like other vendors, RTI provides for heartbeat and NACK "response delays": back-off times during which a producing or consuming application will refrain from putting traffic on the wire, with the expectation that others may be attempting to write at the same time.

- The "heartbeat response delay" specifies how long after receiving a heartbeat from a producer a consumer will wait before responding with an ACK or NACK.
- The "NACK response delay" governs traffic in the other direction, allowing a producer to wait before resending messages to a consumer.

These delays are specified in terms of minimum and maximum values; the actual delay will be some random value in between them. This use of a randomly timed response, configured across a time window, causes NACKs and resent messages to be spread out in the time window instead of creating peaks of bandwidth usage.

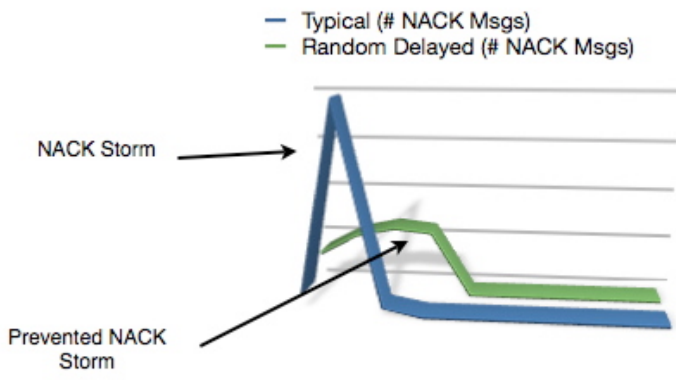

Figure 2 NACK storm prevention with random delays

Without a random response delay, NACKs can occur all at once, causing a spike in network traffic, as shown conceptually in the diagram above. This spike can deny network access to live application data. RTI uses random delays to smooth out those spikes, allowing data to flow normally.

Step 3: Use Multicast Intelligently to Prevent Feedback Loops

In many middleware implementations, messages and their acknowledgements travel either over unicast only or multicast only. In the former case, message volumes may not scale to meet real-world needs. In the latter case, even a mild surge in NACK traffic can result in a follow-on surge of multicast message repairs, which will be received, and must be processed, even by those well-behaved consumers that did not miss the message initially. The associated network and CPU loads of these repeated resends and re-acknowledgements can continue to deny network access to live data long after the initial NACK spike is over.

RTI middleware avoids this problem: it can use both unicast and multicast addresses and *switch from one to the other* seamlessly and intelligently to isolate slow consumers from their better-behaving peers, helping to prevent the feedback loops of redundant resends and re-acknowledgements that can result from a surge in NACK traffic.

First, consumers can be configured (individually or by class) to listen for messages on either unicast or multicast addresses. In topologies in which the number of consumers is limited, unicast addressing can provide superior isolation and decoupling without significantly impacting performance. In this scenario, all repair traffic will be targeted to specific slow consumers, avoiding increased loads on well-behaved consumers. A single producer can communicate with consumers using any combination of unicast and multicast addresses.

Second, even when the middleware is configured to send application messages over multicast, consumers will respond with NACKs over unicast to the specific producer whose data they are missing. The producer, in turn, can respond with message repairs either over unicast, for maximum isolation of a small number of slow consumers, or multicast, for efficiency in the case where many consumers need repairs. How it does this depends on its configured NACK response delay and the number of NACKs it receives before the delay elapses.

This scheme limits the ability of poorly behaved consumers from bringing down the rest of the network in several ways:

- Consumers are decoupled from each other. Since one consumer does not depend on any other to NACK its missed data, one misbehaving consumer cannot cause another to also misbehave or lose data.
- A single slow consumer will never lead to extraneous resends to up-to-date consumers.
- The middleware can provide robustness in the face of multiple slow consumers in several ways: (*1*) by responding to each of them independently over unicast, so that up-to-date consumers receive no duplicate messages that they will have to discard; (*2*) by configuring different groups of consumers with different multicast addresses to allow multiple repairs to be sent efficiently over multicast while limiting the impact on up-to-date consumers; and (*3*) by using ACK suppression (see the section *Windowed Reliability* above) to prevent unnecessary feedback to the producer in the event that redundant resends do occur.

Used together, these three strategies—intelligent data filtering and flow control, random delays to smooth traffic spikes, and adaptive addressing of message resends—can significantly improve an application's avoidance of, and robustness to, traffic spikes.

Conclusion

Users of one-to-many reliable messaging systems are rightly concerned about the negative side effects of pathological traffic patterns. Fortunately, RTI offers a number of capabilities that provide robustness and resilience to applications in order to dramatically reduce the probability of such traffic patterns. In the event that the worst happens, RTI provides applications with full control over the failure and recovery modes.