All Your Arbiter Are Belong To Us

Austin Gilbert

June 7, 2015

1 Introduction

Distributed systems operate over unreliable networks. On these networks, messages can be dropped, corrupted, duplicated, or arrive late, yet many classes of distributed applications have state that *must* be idempotent, that is actions taken in response to arriving messages happen *exactly* once.

There are different approaches to providing idempotency depending on the application specifics. Here we'll discuss a scenario common to the financial industry, where we have an application consuming a message stream needing to apply incoming message transformations to internal state *exactly* once. In particular, we'll focus on sequence arbitration for discarding duplicate messages.

In order to limit the scope of this paper, we'll assume all messages arrive uncorrupted in a timely manor.

In finance, the incoming streams are typically low-latency and high volume, and in order to meet necessary data rates the streams are typically delivered through unreliable protocols such as multicast User Datagram Protocol (UDP) [1, 2]. In this scenario, network redundancy is used to avoid the negative impacts of packet loss, therefore all messages are sent on an *A line* and a *B line*. No guarantees can be made about which line the message will arrive on first. We have to discard the later arriving duplicate message to ensure idempotency of our application state.

A common approach for ensuring idempotency in this scenario is for the publisher to assign each message a monotonically increasing sequence number as a unique identifier. Using the sequence number, the application can determine if a message is new or a duplicate of a previously processed message. Using sequence numbers to make this determination is called sequence arbitration.

2 Sequence Arbiter

A sequence arbiter is the software component responsible for tracking sequence numbers of arriving messages, discarding duplicates and reporting gaps.

In the naive implementation of a sequence arbiter there is typically a circular buffer for storing messages called the *arbiter cache* or the *history buffer*. The depth of the circular buffer is determined by the speed of the lines and the likely difference in arrival times between the A side and the B side, *i.e.* how long we want the application to be able to detect and discard duplicates. At the data rates typically observed, holding more than a few milliseconds of data in the arbiter cache quickly becomes infeasible.

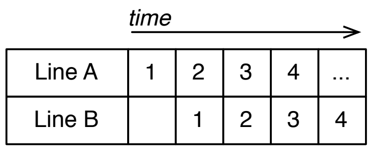
As a message arrives on a line, the arbiter uses the message's sequence number to perform a lookup in the cache to determine if the sequence number has been seen before or not. Any new messages are accepted and are inserted into the cache, duplicates are discarded.

This arbiter implementation works, we detect duplicates and discard them. However, there is a subtle flaw to the naive sequence arbiter implementation which is frequently overlooked. We don't track which line the message sequence numbers arrived on.

Duplicate messages arriving on the same line would be discarded without detection. Duplicate messages arriving on the same line can be indicative of buggy publishing software or nefarious activity happening on the network.

3 Impact of Naive Sequence Arbiter Behavior

Under normal operation, the publisher will send on the *A line* first followed by the *B line*. Typically, the *B line* lags behind the *A line* by a small but measurable amount of time, though this is not guaranteed. A message stream operating normally is pictured in figure 1.



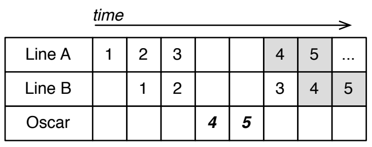
Line A	1	2	3	4	...
Line B		1	2	3	4

Figure 1: Normal Line Behavior

Now, let's assume our message stream is UDP multicast. Let there be an attacker Oscar who can inject arbitrary data on either the A line or the B line. The assumption is Oscar *can* inject arbitrary packets onto the network. We further assume Oscar is able to spoof the sender's Internet Protocol (IP) [3] address as necessary. For example, if Internet Group Management Protocol (IGMPv3) [4] were being used to subscribe to multicast channels, spoofing the sender may be necessary.

There are many ways Oscar can accomplish this level of access. Oscar could live on the same subnet of the same layer 2 networking device which doesn't have Access Control List capabilities or they aren't in use. Oscar could have control of a router between the publisher and the consumer applications. Or perhaps a router in between Oscar and the consumer application allows spoofed packets into the stream. Whatever the case, we assume Oscar has this capability.

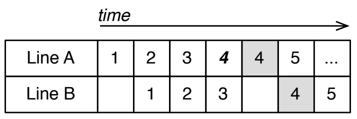
Figure 2 shows what happens if Oscar injects messages into the stream. Here we show Oscar injecting two messages with sequence numbers 4 and 5. The result



Line A	1	2	3			4	5	...
Line B		1	2			3	4	5
Oscar				***4***	***5***			

Figure 2: Abnormal Line Behavior

is messages 4 and 5 are *new* messages which will be accepted by the sequence arbiter. Then a little later, when the legitimate messages with sequence numbers 4 and 5 arrive they are *silently* discarded by the sequence arbiter as duplicates. They are duplicates, but Oscar has masked legitimate messages with his own contents, the results of which could be bothersome to say the least.



Line A	1	2	3	***4***	4	5	...
Line B		1	2	3		4	5

Figure 3: Buggy Publisher Sends Duplicate On Line

In a similar scenario, rather than assuming Oscar exists, we assume a much more likely problem: there is a bug in the publishing software or network causing the delivery of duplicate messages on the same line. Figure 3 shows what this would look like.

The first message having a sequence number of 4 would shadow the next two messages with sequence number 4. The naive sequence arbiter implementation would again *silently* discard the duplicates, leaving no record the situation occurred.

This scenario may or may not be a problem depending on the contents of the messages. If all the messages contain the same content, there is no problem, the arbiter ensured idempotency of the application state correctly. However, if the first message contained different content because of a bug in publishing software, the consumer application may be in an error state - having *not* processed the duplicate message 4 which it should have. The arbiter implementation *silently* fails in this scenario because it cannot detect duplicates on the same line.

4 Corrected Sequence Arbiter

In order to detect duplicate sequence numbers arriving on the same line, a functionally correct sequence arbiter stores the line identifier observed sequence numbers arrive on. When we observe a sequence number is duplicated, we must then check whether the previous observed occurrence arrived on the same line as the current message. If it has, an error is reported to the application, if not, we mark the sequence number as observed on the current line and reject the message as a duplicate.

Without a line identifier in the arbiter cache, it is not possible to detect duplicates on the same line. A sequence arbiter storing the line identifier in its history cache is able to report the *duplicate on a line* error. The *arbiter* library available on https://github.com/paxos1977/arbiter serves as a reference implementation for a functionally correct sequence arbiter capable of reporting *duplicate on a line* errors to its application.

5 Conclusions

The naive implementation of a sequence arbiter is not capable of detecting and reporting duplicate sequence numbers on the same line. This leaves the arbiter component unable to detect and report nefarious network activity or bugs in the publishing software to its application.

Detecting and reporting duplicate sequence numbers on a line is both possible and seemingly performant enough to be practical.

Whether or not there is value in detecting duplicates on the same line is a decision to be made by the operators of the process. In finance, it's not atypical to ignore a flaw with a low probability of occurrence in exchange for a speed advantage for more typical code paths. We're publishing this paper in the hope such decisions are not made in ignorance of the consequences. Further, we hope the shared knowledge and reference implementation will prove useful for implementers both in finance and other problem domains.

References

[1] J. Postel. RFC 768: User datagram protocol, August 1980.

[2] S. E. Deering. RFC 988: Host extensions for IP multicasting, July 1986.

[3] J. Postel. RFC 760: DoD standard Internet Protocol, January 1980.

[4] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. RFC3376: Internet group management protocol, version 3, October 2002.