Specification for NACK Behavior in NLSR

Nick Gordon

October 28, 2016

Abstract

This document describes what behavior NLSR should take when receiving NACKS. The optimal behavior differs from case to case. All important cases are given here.

1 Introduction

There are multiple cases to consider with respect to Interest type when describing NLSR's optimal behavior with regards to NACK processing. Generally, there are two broad strategies. We identify five different NACK situations and the responses for each. Further, in each case, the mention of logging implies a higher level than normal; all important operations of the code generally emit log messages.

It is also important to consider that if NLSR receives a NACK, then all of the outgoing Faces have been NACKed, not just one.

It is important to note that "congestion" as a NACK reason has not been implemented yet. As such, we have elected to treat it as part of "no reason" until the semantics and scenarios associated with it are established.

Finally, all situations have a default of logging the NACK and its context and do nothing else.

2 LSA, key, and validator Interests

This is the most detailed case. They are:

- No route: This may be caused by a link failure, and so does not represent a total failure of routing; if some Face is destroyed, that will be reflected in adj. LSAs and reactions will propagate accordingly. In the case that we have no route, we should wait 60 seconds, giving NFD plenty of time to recover the link, and then resend. There should be a retry limit of 3.

- Duplicate: We should resend the Interest with a limit of 3 in the case we get a duplicate NACK. Of note is that it is impossible to have an Interest sent with multicast strategy return a "duplicate" NACK at every Face at its originating router. Hence, it is likely that "duplicate" NACKs will obscure a more severe NACK. Using the figure, if A sends an Interest for data that is located at F, and the Interest directly from A arrives at D first, then D will "duplicate" NACK the Interests as they arrive through C and E. A's strategy will wait until the Face to D either receives a NACK or data. Then, when D NACKs A's Interest as "no route" because the Face to F is down, the multicast strategy will select the least severe NACK to return to NLSR, here "duplicate." This prevents us from being able to handle more severe NACKs that are the true issue.

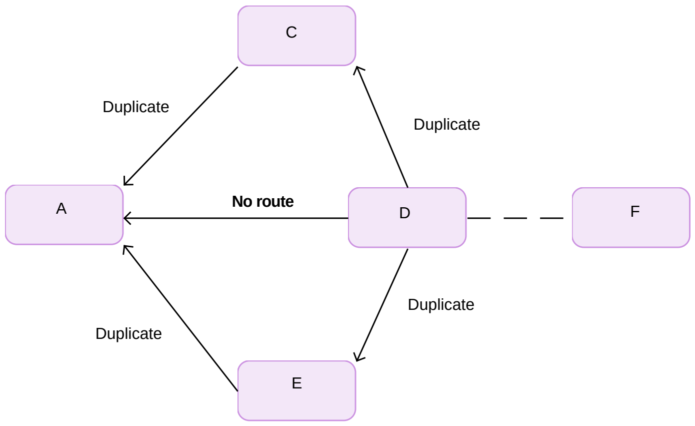

Figure 1: A
situation in which NACK loss can happen.

- <u>No reason:</u> Nothing can be done about this NACK. We should log the NACK message and its context (i.e. the function name that execution is currently in, the logical "task" that the code is performing, etc.), but do nothing else.

3 Hello Interests

This is the second interesting case. However, we note that only two cases can occur. That is, only the "No route" and "No reason" NACKs are possible:

- <u>No route:</u> A NACK with this reason for a Hello Interest indicates that NLSR is down on that router or that the link out of this router is down. That is, NFD has gotten the Interest, but does not know where to forward it. This will only happen if NLSR's prefixes are damaged or not registered altogether. We should emit a particular error message and treat that neighbor as down. We will retry as normal at the next Hello interval.

- <u>No reason:</u> The same strategy that is taken in the LSA case is taken here.

4 Sync Interests

Sync should probably log and otherwise ignore all NACKs gotten. This is a practical reason, as NLSR currently uses a forked version of ChronoSync. Currently, the original ChronoSync does not process NACKs, either, and so we do not want to write patches for outdated code that will be obviated by a merge back, anyway.

5 FaceStatus Dataset

When NLSR starts, it requests a FaceStatus dataset from NFD, which theoretically provides the information NLSR needs to configure its neighbors. In the case that NLSR cannot fetch this dataset, the response should be the same for all NACK reasons; do nothing. This is because NLSR is also listening to the FaceEvent stream, and will get Face information as it as added to NFD. Additionally, NLSR will refetch the dataset at long intervals of typically an hour, to catch the cases where NFD Face events get lost for some reason.

6 Nlsrc

In the case of Nlsrc-originating interests, we should only emit errors for the operator. This is because nlsrc, as a command-line utility, should not autoresolve issues like this. These are the NACK reasons:

- No route: In this case, NLSR is not running on the localhost, and so we should log an appropriate error, but do nothing else.
- No reason: We should log some kind of error, but do nothing else.

7 Centralized Layout

It is clear that the responses are similar or even identical across multiple scenarios. As such, it is ideal to implement a handler that can act as an intermediary between NLSR modules and the NACK return. This will permit lightweight refactoring of the responses that NLSR functions can take, and centralize their definitions for readability and maintenance.

Such a system requires a degree of modularity on the part of NLSR and the methods that provoke NACK situations; they need to be callable independently or mostly independently of their original context, so that the handler has liberty to recover and act accordingly. Then, after operations terminate, to return control to the function that triggered the NACK.