NDN Technical Report NDN-0077. http://named-data.net/techreports.html
Revision 1: August 2023

Lessons Learned from Fixing the Dead Nonce List in NFD

Philipp Moll, Varun Patil, Xinyu Ma, Tianyuan Yu, Lixia Zhang
UCLA
Los Angeles, USA
{phmoll,varunpatil,xinyu.ma,tianyuan,lixia}@cs.ucla.edu

Alex Afanasyev
Florida International University
Miami, USA
aa@cs.fiu.edu

Junxiao Shi
NIST
Gaithersburg MD, USA
junxiao.shi@nist.gov

ABSTRACT

Recent experimentation with NDN networking under ad hoc mobile conditions with intermittent connectivity revealed several issues regarding the existing design and implementation of the "dead nonce list" (DNL). DNL is a mechanism to stop Interest packets from looping after they are removed from an NDN node's the Pending Interest Table (PIT). A detailed analysis of the observed problems leads to the discovery of a few design flaws and implementation shortcomings in the existing solution.

This report explains the intended role of DNL in NDN forwarding, the identified flaws in the original design assumption about the DNL usage, and the implementation issues identified. Through this exercise of discovering and fixing the DNL design and implementation problems, we gain a deeper understanding in the complementary roles played by DNL and HopLimit in packet looping mitigation. We also learned that simply running an NDN testbed cannot be a sufficient proof for the absence of the NDN protocol design issues or implementation bugs; instead, it is necessary to perform thorough measurement and analysis to identify them.

1 INTRODUCTION

Our experimentations on the NDN Testbed with emulated high dynamic changes in network connectivity showed an unexpected behavior: Interest packets loop. By design, this should not happen — looping Interest packets should be detected and stopped either by the virtue of Pending Interest Table (PIT) if the looping Interest is still in the forwarding node's PIT, or the Dead Nonce List (DNL) which keeps all the "dead" Interests for a predefined amount of time after an Interest is deleted from PIT by Data packet arrival or lifetime expiration. Further examination exposed incorrect behavior of DNL, in part because of the initially incorrect design assumptions, and in part caused by implementation issues.

NDN's stateful forwarding plane uses the PIT at each forwarder to mitigate Interest looping: upon receiving an Interest packet I, a forwarder F checks its PIT to see whether I has passed by earlier. However, early NDN experimentation with short Interest lifetime, in the context of NDN-RTC application [2], showed the need to keep PIT entries (some record of them) after they are removed either after satisfying with Data packet or expiration [Burke, personal communication]. Thus, a forwarder F needs another mechanism to stop looping Interests. Hence the DNL was introduced. DNL stores the hash of the name and nonce carried in an Interest packet after it is removed from PIT. Upon receiving an Interest packet I, F checks DNL first to see whether I is a looping Interest before putting it through the forwarding process.

The observed issue came to existence because the assumed "simple" DNL design is in fact not that simple. The first design question is how long the DNL should remember "dead" nonces. If they are not kept long enough, Interest packet looping may still occur. On the other hand, if they are kept for too long, not only they would take up memory space, but also there is a potential danger, however slim, that a new Interest with coinciding nonce can be incorrectly treated as a duplicate which, effectively, results in denial of service. As a trade-off, the existing DNL design picks six seconds as the value, which seems appropriate in the NDN Testbed context [4]. We call this value the *longevity of a DNL entry*.[1]

The second design question is which nonces need to be kept on the DNL. Naively, the original design assumed that only unsatisfied Interests with too short a lifetime, which should be a small percentage of the total Interests, need to be stored in DNL. However, a careful analysis shows that the majority of Interests should be moved to DNL after they get removed from PIT, due to the following reasoning: (1) Satisfied Interests (which should be the majority of all Interests in the absence of errors or attacks), by definition, reside in the PIT for less than their lifetime. Although the Data packets retrieved by satisfied Interests are cached in the Content Store (CS) and can be used to stop Interest looping, the effectiveness depends on the CS capacity and caching policies. In particular, this looping mitigation by caching does not work for constrained devices with limited storage capacity. Thus, all the satisfied Interests should be moved from PIT to DNL; (2) Long-lived Interests, by the name, have a long lifetime. However, if their lifetime is shorter than the DNL longevity time, these unsatisfied long-lived Interests should also be moved from PIT to DNL as well. The above two reasons invalidate the initial assumption that DNL only needs to store a small portion of the total Interests passing through an NDN forwarder. In addition to this false design assumption, there also exist implementation bugs, that we elaborate later, reduced the DNL's effectiveness.

In this technical report, we first discuss the rationale for having the Nonce field in each NDN Interest packet, and a DNL at each forwarder (§2). We then describe the original DNL implementation, explain what went wrong, what has been done in fixing the errors, and identify additional remaining work for an ideal DNL implementation (§3). In §4 we discuss the lessons learned, including a clarification of the complement roles played by DNL and the HopLimit field in NDN Interest looping mitigation.

[1] Experimentations with ad hoc/delay-tolerant networking showed that this value may not be sufficient.

2 THE NONCE FIELD AND ITS RATIONALE

In an NDN network [9], data consumers send Interest packets to request content chunks by names or name prefixes. While the name is the only required field for a consumer to request a piece of data [3], other fields may be needed to guide the Interest when forwarded through a network. One of such fields is the *Nonce*, a random number that accompanies Interests. If not already present, the Nonce field is added to the Interest by the first forwarder. The combination of an Interest's name and nonce should uniquely identify the Interest packet, which allows forwarders to detect looping Interests.

Although HopLimit already provides looping Interests detection, relying on this field alone shows inefficiency in alternative path exploration by a forwarding strategy or multipath forwarding (especially in ad hoc environment). Therefore, the main purpose of adding a Nonce field to an Interest is to more effectively detect looping Interests. As shown in Fig. 1, forwarding an Interest I_1 creates an entry in PIT, which is used to return a matching Data to the requester(s). As a side effect, whenever I_1 is looped back to a forwarder, the existing PIT entry can be used to detect and stop the loop. However, a PIT entry with the requested data name *alone* on Node 1 cannot tell whether such an Interest is a looping Interest, or is from a different consumer which sends an Interesting with the same name. Adding the Nonce field helps make such distinction. Mechanically, each NDN forwarder F remembers the nonce of every forwarded Interest in the corresponding PIT entry. Upon receiving an Interest, if F finds a matching PIT entry for the Interest's name, it checks the nonce carried in the incoming Interest against all the nonces associated with that PIT entry. If no identical nonce is found, the incoming Interest is considered from a different consumer, whose incoming face is added to the PIT entry and the Interest is dropped. If an identical nonce is found, the incoming interest is treated as looped and is dropped, and an NACK with type *Duplicate* can be sent to the incoming face of this looped Interest.

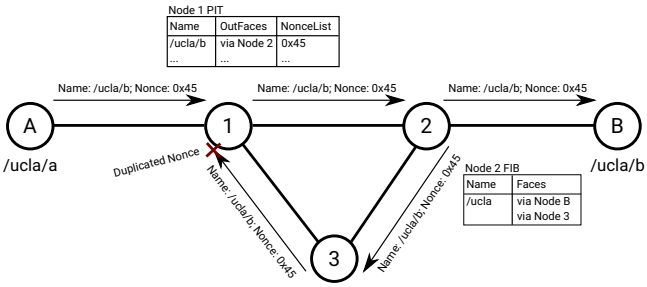

Figure 1: According to Node 2's FIB, Node A's Interest is forwarded to Node B and 3. Node 1 detects the looping interest by using the nonce list in Node 1's PIT.

While it was not apparent at the initial design of NDN, early experimentation highlighted the need for keeping Interest state for longer time period beyond their PIT residence time to effectively mitigate Interest looping. Specifically, NDN forwarders require additional state to detect and break Interest loops that last longer than the round-trip time for data retrieval, and/or longer than the interest lifetime. The DNL was added to serve this purpose in 2014 after looping Interests were observed on the NDN Testbed. The objective

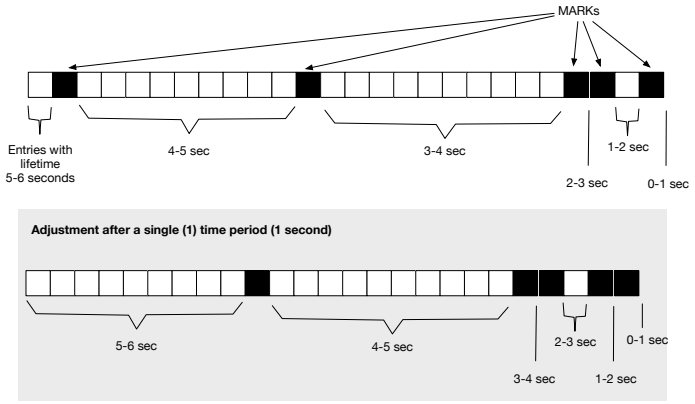

Figure 2: Flow Table Algorithms

of DNL is to preserve nonces collected from removed PIT entries for a predefined time. This time should be long enough to detect Interest loops and short enough not to cause excessive memory use and Interest misclassification. Using the DNL, the incoming Interests processing pipeline is slightly altered: before checking the content store and the PIT, a forwarder first checks against the Interest name and nonce combination in the DNL, treating a DNL match the same way as looped Interest.

Since the DNL cannot keep growing forever, entries need to be removed after a specified time. The DNL implementation details may vary between different forwarders. The next section described the DNL implementation in the NDN Forwarding Daemon, *NFD* [1].

3 NFD'S DEAD NONCE LIST

3.1 Current DNL Realization and Its Flaws

Inspired by the concept of *hashed wheel timers* [8], we designed an algorithm to approximate lifetime of DNL entries using time buckets. This algorithm captures temporal components by including *MARK*-entries into the DNL, effectively resulting in negligible overhead. However, when designing we overlooked that this algorithm requires a constant flow of nonce insertions to work properly. When facing nonce bursts (when a large number of pending Interests get satisfied), the algorithm may not be able to adjust correctly and results in over-/underflow of items.

As depicted in Fig. 2, the lifetime of entries in the current DNL implementation is controlled dynamically by adjusting the capacity of the container. Every 1-second interval, a special *MARK*-entry is inserted into the DNL, and the quantity of *MARK*-entries in the container is recorded. Effectively, normal entries between *MARK*-entries represent entries with the corresponding lifetimes.

Every 6-second interval, the quantity of *MARK*-entries, as recorded in the past six 1-second intervals, is compared against the target value of 6. If all six recorded quantities are greater than 6, it means the DNL has been holding more entries than necessary and the oldest entries have been held for longer than the expected 6-second entry longevity. In this case, the DNL capacity is adjusted down by 10%, so that its memory usage can be decreased. If all six recorded quantities are less than 6, it means the DNL capacity is insufficient for the expected 6-second entry longevity. In this case, the DNL

capacity is adjusted up by 20%, so that more entries can be accommodated. If some of the six recorded quantities are greater than 6 and some are less than 6, NFD does nothing until the traffic is stabilized.

When the capacity of the DNL is sufficient to store all newly inserted entries, the algorithm works as our expectation. However, when the allocated DNL capacity is too low to accommodate all the dead nonces to be added to DNL, entries will be prematurely removed starting from the left end. To handle this problem, the algorithm periodically adjusts the DNL size:

- If at the time of check, the DNL is fully utilized, increase its size by 1.2.
- If under-utilized, then reduced by 0.9.

The existing implementation chose a period of *six seconds* to perform the above check and adjustment. During this six seconds, if nonce insertion bursts, a low DNL capacity leads to premature eviction of DNL entries. The presented algorithm also requires to be initialized to a pre-configured DNL capacity, which is set to 128 entries in NFD releases before 0.8.0. Assuming a burst of 1000 interests packets, this configuration and the slow capacity adjustments would require NFD 12 adjustment cycles (over a minute) to allow fitting all nonces into the DNL. During this time, effective loop detection using the DNL is not possible.

Moreover, analysis on the NDN Testbed revealed a bug[2] in NFD's DNL implementation that allowed inserting duplicate entries to the DNL in the following ways: (1) When multicast Interest packets are forwarded through a node with high degree of network connectivity, a PIT entry may have a number of out records holding the same nonce. (2) When removing a PIT entry, the nonces of all the corresponding out records are moved to the DNL, without checking for duplicates before inserting. This resulted in redundant DNL entries.

As a result, multicast Interests resulted in the same number of duplicate entries in DNL as the number of faces these Interests were forwarded to, and potentially wasted valuable DNL capacity that is already insufficient to handle bursts, which consequently failed Interest loop detection.

3.2 Patches and Long Term Solution

Patches (NFD 0.8.0+): To address the aforementioned issues observed on the NDN Testbed, we applied two patches to NFD.

(1) *Deduplicating entries inserted into the DNL.* While the forwarding pipelines are still allowed to invoke insertion of duplicate name-nonce pairs (avoiding this requires a separate planned fix), the insertion procedure now ensures that instead of adding duplicate entries to the end of the DNL, the existing entry that matches the looping Interest is relocated to the start of the DNL. Such relocation ensures that the existing entry will not be evicted too soon.
(2) *Increasing the initial capacity of the DNL by a few orders of magnitude to* 2^{14} *entries.* While this value may still be insufficient to handle large bursts, it temporarily fixes the observed issue.

[2]Reported in [6], reproduced with MiniNDN using the following scenario: https://github.com/phylib/dnl-experiment

An Ideal DNL Implementation: Besides patches, we also plan to provide an ideal DNL implementation as a long term solution. The analysis revealed that relying on container size adjustments as a proxy to control the longevity of DNL entries is ill-suited for the burst traffic pattern on NDN networks. It is necessary to have more direct control on the longevity of DNL entries.

The most straightforward solution is storing a timestamp with each DNL entry. As analyzed in [5], this change would triple the memory usage, resulting in significant memory overhead. Instead, for a DNL with expected longevity L, we propose the following data structure:

- The DNL consists of M buckets, where each bucket is a hashtable-like data structure. We expect insertion and lookup on this data structure have a time complexity of $O(1)$.
- To insert an entry, the entry is inserted into the bucket at index $M-1$. The time complexity of this operation is $O(1)$.
- To lookup an entry, the entry is searched among all M buckets[3]. If it is found in any of the buckets, it exists in the DNL. Otherwise, it does not exist in the DNL. The time complexity of this operation is $O(M)$[4].
- Every L/M duration, the bucket at index 0 is discarded, the remaining buckets are shifted down (bucket $i+1$ becomes bucket i), and an empty bucket is created at index $M-1$. The time complexity of this operation is either $O(1)$ or $O(N/M)$ (where N is the average number of insertions per L), depending on data structure design.

This design is inspired by *timing wheels* [7]. It exploits the requirement flexibility that while each DNL entry shall be kept for a configurable longevity, this longevity need not be precise. With this design, the precision of DNL entry longevity is L/M, which could be reasonably small with appropriate parameter settings.

The parameter settings of this design is a tradeoff between DNL entry longevity accuracy and computational overhead. For the same expected longevity L, choosing a larger M would decrease L/M and achieve more precision in DNL entry longevity, but increase the computational overhead of DNL lookup operation. On the other hand, choosing a smaller M would increase L/M and cause less precision in DNL entry longevity, but decrease the computational overhead of DNL lookup operation.

There are several options for the data structure within each bucket. Using a regular hashtable (`std::unordered_set`), insertion and lookup can satisfy the expected $O(1)$ time complexity, but the memory usage may grow as more entries are inserted into a bucket; the periodical discarding operation may have up to $O(N/M)$ time complexity for memory deallocation. Using a bloom filter, insertion and lookup can likewise satisfy the expected $O(1)$ time complexity and the memory usage is fixed; the periodical discarding operation is guaranteed to have $O(1)$ time complexity because a fixed amount of memory is being zeroed. However. the inherent possibility of false positives in bloom filters may cause the DNL lookup procedure to erroneously report an entry to be exist despite it was not inserted. This in turn causes the forwarder to reject an Interest that should have been accepted, which is a correctness issue. We need to further

[3]Since NFD is a single threaded forwarder, this look up cannot be parallelized or pipelined.

[4]Generally speaking, M is a small factor that is not supposed to scale, so actually $O(M) = O(1)$.

analyze the implication of bloom filter false positives, and also perform benchmark testing to get quantitative results to get a better understanding of the above tradeoffs.

4 DISCUSSIONS AND LESSONS LEARNED

This section discusses design considerations and lessons learned during the development of the DNL and documents the reasons for design decisions.

4.1 Interest Longevity vs. DNL Entry Longevity

Theoretically speaking, the time period to keep the state for an Interest should start from its injection into the network instead of the time it is added to DNL. That is, one should use Interest longevity instead of DNL entry longevity to decide when to remove the state. Doing so can remove those Interests that already have extended stay in PIT from DNL sooner. However, doing so can also add additional complexity into the DNL implementation. Given the DNL's memory consumption is not viewed as an issue at this time, the current design makes use of DNL entry longevity, trading additional memory cost for simplicity[5].

4.2 The Nonce vs. the Hop Limit Field

The Interest packet loop detection by using unique nonces does not guarantee the elimination of Interest packet looping. In rare cases of a nonce collision, or an Interest packet wanders inside the network for long enough time, so that when it is received by forwarder F the second time, its corresponding Data packet has been removed from F's CS, and its nonce deleted from F's DNL, the loop is not detected. Note that Interest loops of short duration can cause high workload to the network, one may choose the DNL size to be large enough to prevent such short loops. An Interest loop of longer duration will be removed when the Interest's HopLimit reaches zero.

4.3 Running Code: Enabling Measurement, Not Substitute for Meamsurement

The DNL is one of the fundamental components required for NFD to work correctly. Having a bug in the DNL implementation, whereas NFD's codebase underlies a code reviewing system, having software test in place, and being deployed for several years on a global NDN Testbed [4] was unexpected and teaches two lessons:

(1) *Premature optimization is the root of all evil*[6]*:* The original DNL implementation used in NFD is based on a sophisticated algorithm originating from IP network research. The rationale for using this algorithm in NFD is unclear, and evaluations showing superior performance compared to simpler DNL approaches are missing. Besides, scarce documentation of the implemented algorithm makes sufficient testing, code reviews, and improvements on the DNL structure challenging. All these aspects may be potential reasons for not having detected the bug in over six years of development. Starting NFD development with a simple yet less efficient DNL structure and evaluated incremental improvements may have led to a better understanding of the DNL, and potentially could have prevented the issue.

(2) Setting up infrastructure, such as the NDN Testbed, is not sufficient for testing a system. Using the infrastructure (e.g., for real applications) and exercising experimentation is required to find issues and limitations.

[5] If one assumes that the PIT residency time for most Interests is within sub-second range, then this additional memory cost should be minimal.

[6] Quoted from Tony Hoare

4.4 Traffic Unpredictability and DNL Memory Allocation

In addition to unpredictable traffic surges, it is also well-known that an NDN network can be DDoSed by Interest Flooding Attacks (IFA). IFAs can potentially exhaust the allocated DNL size, disabling forwarders from carrying out loop protection, which can further intensify the attack.

As discussed in §3.2, the next revision to DNL implementation may either use multiple hash tables or multiple bloom filters, one for each time unit. If allocated hash memory is exhausted, one needs to allocate more and pay more processing cost. If the chosen bloom filter size is no longer adequate, that will result in increased false positives (which effectively leads to denial of services). We leave a note here for future efforts to address this potential complication.

5 CONCLUSION

While the concept of NDN's Interest-Data exchange is well described and understood, few details about the packet handling have been discussed in publications. The only existing description about the packet handling is NFD Developer's Guide [1]. It provides insights specific to the reference implementation. However, the reasons for the design decisions, and especially the lessons learned, are largely buried inside various meeting notes or inline source-code documentation. The issues identified in the DNL design and implementation show the need for design documentation and accessibility to the design documents.

This report fills the knowledge void for the DNL design, its implementation, together with the lessons learned. While reviewing potential reasons for the observed issues, we provide an in-depth discussion of the pros and cons of various design choices. We hope this technical report can serve as an sample for future NFD developers to document their work.

ACKNOWLEDGEMENT

This work is partially supported by the National Science Foundation under award 1719403 and 2019085.

REFERENCES

[1] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yanbiao Li, Spyridon Mastorakis, Yi Huang, Jerald Paul Abraham, Eric Newberry, Teng Liang, Klaus Schneider, Steve DiBenedetto, Chengyu Fan, Susmit Shannigrahi, Christos Papadopoulos, Davide Pesavento, Giulio Grassi, Giovanni Pau, Hang Zhang, Tian Song, Haowei Yuan, Hila Ben Abraham, Patrick Crowley, Syed Obaid Amin, Vince Lehman, Muktadir Chowdhury, Ashlesh Gawande, Lan Wang, and Nicholas Gordon. 2018. *NDN-0021: NFD Developer's Guide.* Technical Report. Named Data Networking. https://named-data.net/publications/techreports/ndn-0021-10-nfd-developer-guide/ Rev. 10.

[2] Peter Gusev and Jeff Burke. 2015. Ndn-rtc: Real-time videoconferencing over named data networking. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. 117–126.

[3] Named Data Networking Project. [n.d.]. *NDN Packet Format Specification version 0.3.* https://named-data.net/doc/NDN-packet-spec/current/index.html

[4] Named Data Networking Project. [n.d.]. *NDN Testbed.* https://named-data.net/ndn-testbed/
[5] NDN Team. 2017. Redmine Issue 2235. https://redmine.named-data.net/issues/2235#note-1 accessed: 2023-07-30.
[6] NDN Team. 2021. Redmine Issue 5167. https://redmine.named-data.net/issues/5167 accessed: 2023-07-30.
[7] George Varghese. 2004. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[8] George Varghese. 2005. *Network Algorithmics: an interdisciplinary approach to designing fast networked devices.* Morgan Kaufmann.
[9] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *ACM SIGCOMM Computer Communication Review (CCR)* 44, 3 (July 2014), 66–73.