NDNLP-BFD Project Summary Report

Sheng Hu, Bochun Zhang, Fei Peng
{shenghu, bochunzhang, pengfeifc}@cs.ucla.edu

1. Introduction

NDN Link Protocol (NDNLP) [1] is a layer 2.5 protocol (i.e. between link layer and network layer) running between two nodes currently, which allows the NDN applications run directly on Ethernet. Our project NDNLP-Bidirectional Forwarding Detection is a component of NDNLP.

Current NDNLP don't have a mechanism to detect failures quickly. NDNLP-BFD achieves effective failure detection for NDNLP by observing the transmission in data plane and sending extra small packets when link is idle.

2. Problem

Current NDN forwarding engine has no fault detection component in link layer. Link failures between adjacent forwarding engines could exist in Ethernet, Broadcast and multicast network, and TCP/UDP/IP tunnels. In order to detect NDN forwarding failures in link layer, a link layer failure detection protocol is needed.

3. Design

3.1 Overview

NDNLP-BFD works on point to point unicast links including physical links and virtual tunnels. NDNLP-BFD is a part of the NDN forwarding engine that serves the single purpose of failure detection. NDNLP-BFD enabled NDN forwarding engines can only communicate with engines with NDNLP-BFD enabled as well, otherwise the opposite engine that doesn't reply to keep-alive packets are considered down. Figure 3-1 shows the high level work around of the design.

3.2 Packets and Timers

There are two kinds of packets used in NDNLP-BFD, Keep-alive and Ack. Keep-alive packet is sent when one interface want to make sure the other interface is alive. The Ack packet is used to acknowledge Keep-live.

There are two timers used: Idle Timer and Failure Timer. The Idle Timer is used for one interface to send Keep-alive to the other side when it times out. The Failure Timer is used for failure detection. When it times out, the other interface is considered dead.

3.3 Modes

Each interface of the forwarding engines is responsible for making sure the other interface is alive if it is required to know the link failure status. Whenever there are packets received from the other side, the link is considered alive. These packets includes Keep-alive packet, Ack packet, and NDN interest/data packet.

There are two modes in NDNLP-BFD: full mode and passive mode. Two interfaces in the NDNLP-BFD session need not necessarily be in the same mode, however, at least one of the two interfaces shall be in full mode.

No matter which mode an interface is in, it considers the other side dead when there is no packet received for a failure period. An interface keeps a Failure Timer for failure detection. Each kind of incoming packet including Keep-alive, Ack, and NDN interest/data will refresh the Failure Timer.

3.3.1 Full Mode

Full mode is used when the interface want to make sure the other interface is alive. When an interface is in full mode, it will actively send Keep-alive packet to the other interface if there is no packet received for an idle period.

An interface in Full Mode will keep an Idle Timer for idle detection. When the Idle Timer times out, the interface sends Keep-alive to the other interface. Each kind of incoming packet including Keep-alive, Ack, and NDN interest/data will refresh the Idle Timer. A Full Mode interface keeps both Idle Timer and Failure Timer.

3.3.2 Passive Mode

When an interface is in passive mode, it does not have idle timer and would not send Keep-alive packets to the other interface. A Passive Mode interface keeps only Failure Timer.

3.3.3 Use cases

A typical Full Mode to Full Mode link would be between two NDN routers. In this case, each side would like to make sure the other side is alive. If there is no NDN packet in transit on the link, both interface would actively send Keep-alive to the other side and

the other side would will reply Ack. This symantec will not double the number of packets transmitted since a Keep-alive packet will also refresh the Idle Timer on the other side.

A typical Full Mode to Passive link would be between a router and a subscriber where the subscriber wants to make sure the router is alive but the router does not care if the subscriber (such as a cell phone or laptop) is still online. In this case, the subscriber will actively send Keep-alive packets to the router. The router will automatically consider the subscriber dead after Failure Timer times out.

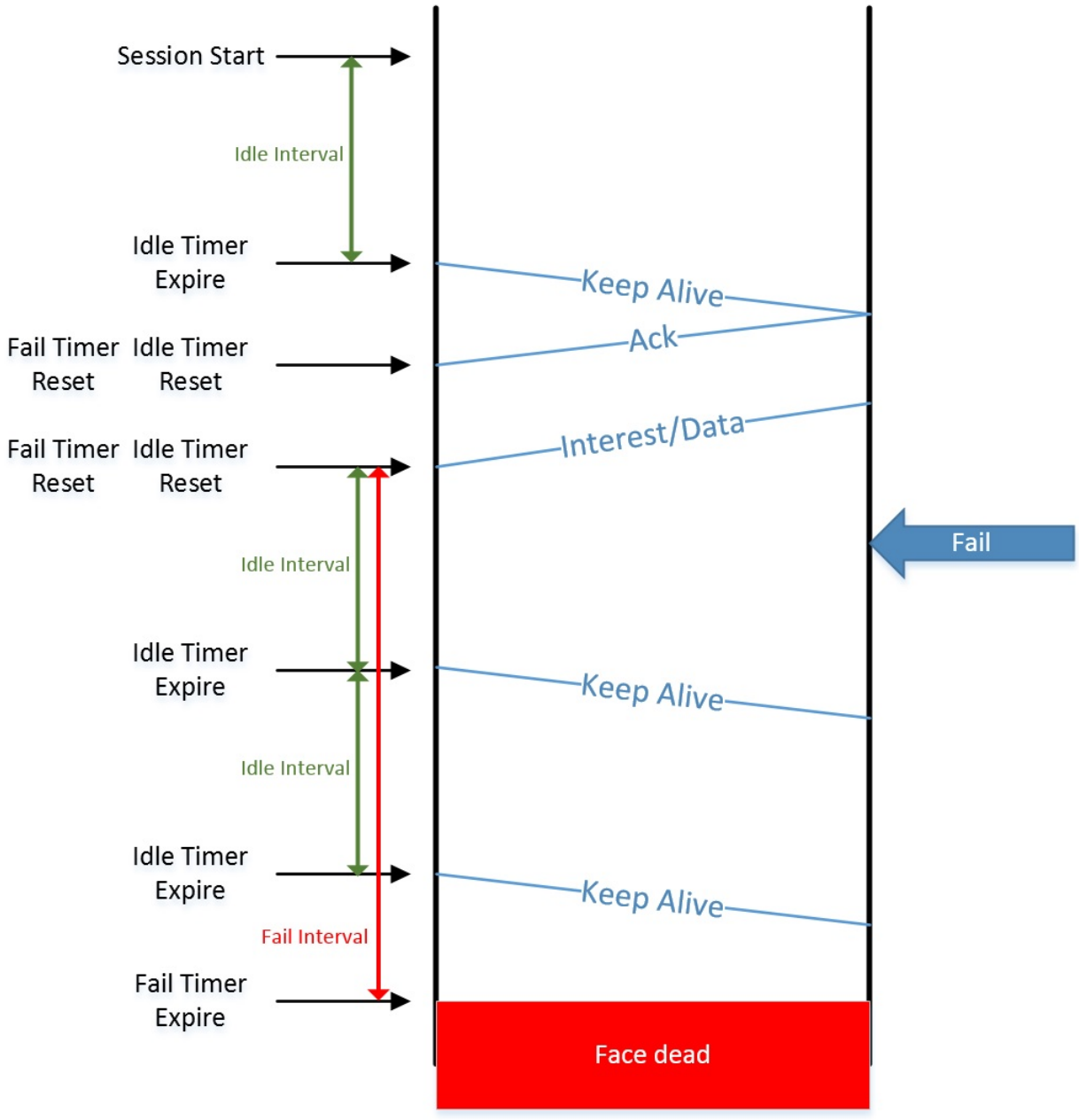

Figure 3-1 NDNLP-BFD work around

3.4 Face API

The basic idea is to extend the nfd::Face API to receive link-layer packet by TLV-TYPE. When NfdFailureDetector is attached onto a face, the detector adds three link-layer receive handlers:

- witness: this is called every time a recognized link-layer packet is received, including Interest, Data, NDNLP-BFD packets.
- processKeepAlive: this is called only when a keep-alive packet is received.
- processAck: this is called only when an ack packet for a keep-alive is received.

The detailed API design is as follows:

```
namespace nfd {

class Face // extends nfd::Face
{
public:
  /** \brief sends a TLV block on link layer
   */
  void
  linkSend(const Block& block);

  /** \brief represents a link layer packet handler
   *  \param block a link layer packet
   *  \return whether the packet has been processed
   *          and shouldn't be processed by another link receive handler
   */
  typedef std::function<bool(const Block& block)> LinkReceiveHandler;

  /** \brief adds a link layer packet handler for a specific TLV-TYPE
   *
   *  Link layer packets of this TLV-TYPE are passed to handlers in ascending
priority,
   *  until a handler returns true to terminate the processing.
   */
  void
  addLinkReceiveHandler(uint32_t tlvType, int priority, LinkReceiveHandler
handler);
};

protected:
  virtual void
  networkSend(Block block);

  virtual void
  linkSend(Block) = 0;
```

```
  virtual void
  networkReceive(Block block);

  virtual void
  linkReceive(Block block);

private:
  std::unordered_map<uint_32, std::multimap<int, LinkReceiveHandler>>
m_linkReceiveHandlers;

  void
  initializeDefaultLinkReceiveHandlers();
} // nfd namespace
```

3.5 BFD API

A face can operate in one of the two BFD modes:

- full mode: in this mode the face can both transmit keep-alive and ack packets
- passive mode: in this mode the face can respond keep-alive packet with an ack packet, but does not transmit keep-alive packet actively

Typically a router's face could operate in passive mode while leaving the job of keeping communication alive to the potentially mobile host, e.g. a laptop, which should operates in full mode.

The detailed API is as follows:

```
namespace nfd {

/** \brief indicates the mode in which BfdFailureDetector operates
 */
enum BfdMode {
   /** \brief PASSIVE mode
    *
    *  In this mode, BfdFailureDetector cannot transmit keep-alive packets.
    */
    BFD_PASSIVE,
   /** \brief FULL mode
    *
    * In this mode, BfdFailureDetector can transmit keep-alive packets.
    */
    BFD_FULL
};

/** \brief supplies options to BfdFailureDetector
 */
struct BfdOptions {
```

```
   /** \brief gives operation mode
    */
    BfdMode mode;

   /** \brief gives idle period
    *  In FULL mode, after receiving the last link layer packet or sending
    *  the last keep-alive,
    *  if no link layer packet of any kind has been received during this
    *  period,
    *  a keep-alive packet shall be sent.
    */
    time::nanoseconds idlePeriod;

    /** \brief gives fail period
    *  In FULL or PASSIVE mode, after receiving the last link layer packet
    *  of any kind,
    *  if no link layer packet of any kind has been received during this
    *  period,
    *  a face failure shall be declared.
    */
    time::nanoseconds failPeriod;
};

class BfdFailureDetector : noncopyable
{
public:
    explicit
    BfdFailureDetector(shared_ptr<Face> face, BfdOptions& options);

    EventEmitter<> onFail;

    ~BfdFailureDetector();

protected:
   /** \brief process NDNLPv2 keep-alive packet
    *
    *  This function is added as LinkReceiveHandler for NDNLP-CHUNK-TYPE.
    *
    *  This function:
    *  1. If NdnlpChunk represents keep-alive, respond with acknowledgement
    *
    *  \return true if NdnlpChunk represents keep-alive
    */
    bool
    processKeepAlive(const Block& block);

   /** \brief process NDNLPv2 acknowledgement packet
    *
```

```
     *  This function is added as LinkReceiveHandler for NDNLP-CHUNK-TYPE.
     *
     *  This function does nothing. It's just to drop the packet.
     *
     *  \return true if this packet acknowledges a previous keep-alive packet
     */
     bool
     processAck(const Block& block);

    /** \brief witness a non-NDNLPv2 packet
     *
     *  This function is added as LinkReceiveHandler for all TLV-TYPEs
     *  with small priority number.
     *
     *  This function:
     *  1. reset idle timer and fail timer
     *
     *  \return false
     */
     bool
     witness(const Block& block);

private:
     shared_ptr<Face> face;
     BfdOptions m_bfdOptions;
     ndn::EventId failTimerId;
     ndn::EventId idleTimerId;

    /** \brief send a keep-alive packet on link layer
     */
     void
     sendKeepAlive();

    /** \brief report face failure to the upper hierarchy
     *  This function basically call Face::fail to trigger the failure event
     */
     void
     reportFailure();
};

} // nfd namespace
```

4. Implementation

Since our code is implemented for experiment purpose, currently both Idle Period and Fail Period are hard-coded in the implementation and use values of one second and three seconds respectively. In addition to that, since the the changes in the Face API

affect most of the inherited different faces and we are limited in time, we only integrated our implementation into DatagramFace and therefore carried out the test on UDP channel.

We also hard-coded the BFD mode when testing, but we will look into the upper layer codes in the future to make it automatically configured according to whether the face is created on demand or not.

We have tested both full mode and passive mode on our computers. The basic functions designed above are working correctly. Later we would get rid of the hard-coded timers to make our implementation more flexible to use.

5. Environment Setup

Our project is developed based on ndn-cxx-0.2.0 and nfd-0.2.0, which are the stable release versions. To set up the environment for experiment, one should first check out these repositories and then add or replace the files under NFD/daemon/face with the files provided in this link:
https://drive.google.com/folderview?id=0BwwZjGa6HTWfTFdudnRKMGlDQUE&usp=sharing

Once the files are ready, compile the cxx library as usual and compile NFD with C++11. To test if the BFD failure detector works, run the producer and consumer in the built examples of cxx library and use `nfdc register /example udp://<other host>` to register a face. A log file `log.txt` would be created under the home directory (`~`) and one can use `tail -f ~/log.txt` to monitor the real-time status of detector.

6. Issues

- The stable release of NFD[2] compiled with C++11 works well on 32-bit Linux but crashes upon start on 64-bit machine. We are not sure if this is a bug of g++ or nfd-0.2.0.
- Source codes of `websocket-channel.cpp` and `websocket-face.cpp` do not compile with C++11. Since we are not using these part of face, we simply comment out the lines that do not compile. This need further investigation during the integration in the future.
- Due to time limitation, we haven't unit-tested our implementation. This should be done before merging our code into the major fork.

7. Acknowledgements

A debt of gratitude is owed to Junxiao Shi and Prof. Lixia Zhang, for their big contributions to the design of this project, this project would not been possible without their diligent efforts and expertise.

8. Reference

[1] Junxiao Shi, Beichuan Zhang, “NDNLP: A Link Protocol for NDN”, http://nameddata.net/wpcontent/uploads/TRLinkProtocol.pdf
[2] Alexander Afanasyev et, al. “NDN Developer’s Guide”, http://nameddata.net/wpcontent/uploads/2014/07/NFDdeveloperguide.pdf
[3] Katz, D. and D. Ward, "Bidirectional Forwarding Detection (BFD)", RFC 5880, June 2010, http://www.rfceditor.org/info/rfc5880.