miniNDN with ndn-cxx: some use cases

Matteo Bertolino

October 14, 2016

matteo.bertolino@eurecom.fr

Contents

1 Recommendations

- **Recommendation 1:** miniNDN does not run well with ubuntu kernel version too recent. For my experience, I suggest you to use the version 3.16. Some mininet error examples could be *RTNETLINK error* or *Error: gave up after three attempts.*

- **Recommendation 2:** if, at the beginning, you notice an error on the port 6633, just run: *sudo tcpkill -9 port 6633* for at least 30 seconds. If the problem persists, just run the above command other times until miniNDN starts correctly.

- **Recommendation 3:** Another possible error, during the simulation, is something like *no such forwarder connection.* If so, just *ctrl+c* miniNDN and restart it.

- **Recommendation 4:** Another possible error, during the simulation, is something like *private key does not exists.* If so, *ctrl+c* miniNDN like the recommendation 3, and **do not** delete the node folders in the /tmp folder. I noticed that it happens if the node folder are created for the first time in /tmp. If you noticed this error while or after built the certificate chain, you should reinstall all the certificates. Further details later.

  Important note about recommendation 4 from Ashlesh Gawande:
  For the Recommendation 4, "private key does not exist error" is a bug.
  Remove "&" on this line:
  https://github.com/named-data/mini-ndn/blob/master/ndn/nlsr.py#L49
  It will be fixed soon.

2 How to send personalized NACKs from application

Suppose the following use case: there is a consumer, a producer and the producer desires, for any reason, seningd an application NACK to the consumer. This NACK should have a reason different from the known ones (CONGESTION, NO_ROUTE, DUPLICATE, etc) because, for example, it must be treated in a particular way by an NDN entity (the receiver application or the NFD for example. This latter could understand the application nack because it uses the ndn-cxx too, differently from ndn-cpp). Just to perform an example (not so smart): if a producer gets care that a particular host is sending too many interests, it could inform him that in a short period its packets will be dropped if the rate does not become lower, and it could reach the goal sending a Nack with a reason "TOO_HIGH". Let's see how to do:

1- Add the nack reason in */mini-ndn/ndn-cxx/src/lp/nack-header.hpp*:

```
enum class NackReason {
  NONE = 0,
  CONGESTION = 50,
  DUPLICATE = 100,
  NO_ROUTE = 150,
  TOO_HIGH = 174
};
```

2- Adding the new reason in the definition of operator <<and modifying opportunely the *getReason* method in */mini-ndn/ndn-cxx/src/lp/nack-header.cpp*:

```
std::ostream&
operator<<(std::ostream& os, NackReason reason) {
  switch (reason) {
  case NackReason::CONGESTION:
    os << "Congestion";
    break;
  case NackReason::DUPLICATE:
    os << "Duplicate";
    break;
  case NackReason::NO_ROUTE:
    os << "NoRoute";
    break;
  case NackReason::TOO_HIGH:
    os << "TooHighInterestRate";
    break;
  default:
    os << "None";
    break;
  }
  return os;
}
```

```
NackReason NackHeader::getReason() const {
  switch (m_reason) {
  case NackReason::CONGESTION:
  case NackReason::DUPLICATE:
  case NackReason::NO_ROUTE:
  case NackReason::TOO_HIGH:
    return m_reason;
  default:
    return NackReason::NONE;
  } }
```

3- Sending a NACK with reason TOO_HIGH from the application:

```
if (/*something*/) {
        lp::Nack outNack(interest);
        lp::NackReason nack_reason = lp::NackReason::TOO_HIGH;
        outNack.setReason(nack_reason);
        myFace.put(outNack);
}
```

4- Managing the NACK at any NDN entity (example: the strategy of the NFD):

```
void AnyStrategy::afterReceiveNack(const Face& inFace,
        const lp::Nack& nack,
                const shared_ptr<pit::Entry>& pitEntry) {
                if (nack.getReason()==lp::NackReason::TOO_HIGH) {
                        //do something
                }
        }
```

//NB: for any reason, the above *if* is wrong during the compilation phase. I do not know why, in order to test the application NACK so defined, if you get a compilation error, just exclude all the other nack types, for example:

```
if (nack.getReason()!=lp::NackReason::NONE &&
        nack.getReason()!=lp::NackReason::NO_ROUTE &&
                ...)
```

3 Creating a certificate chain, publishing the certificates through repo-ng, signing the data and verifying the chain

Use case - our topology is composed by 5 nodes: a consumer *(CS)*, a producer *(PR)*, an intermediate gateway *(GW)*, a root authority *(AR)* and a sub-authority *(A1)*. The client expresses an interest for a certain data that the producer is able to satisfy. The producer signs the data packet and it sends it. The producer certificate is signed by the sub-authority A1, the certificate of A1 is signed by the root authority AR. The Consumer CS trusts just in AR. So, when CS receives a packet for PR, it needs to verify first of all the PR's certificate, then the A1's certificate. The topology file I used is this one:

```
[nodes]
cs: _ nfd-log-level=DEBUG
pr: _ nfd-log-level=DEBUG
ar: _ nfd-log-level=DEBUG
gw: _ nfd-log-level=DEBUG
a1: _ nfd-log-level=DEBUG
[links]
cs:gw delay=10ms
ar:gw delay=10ms
pr:gw delay=10ms
a1:gw delay=10ms
```

The string *nfd-log-level=DEBUG* guarantees the generation of the node file /tmp/node/node.log, that is very useful to see what's happened. Then, let's see and comment the main steps that we need in order to build a certificate chain. Note that this should be done "offline", before the simulation, and it is not necessary repeating it entirely each time. In this scenario, ar represents the /root domain, the sub-authority the sub-domain /root/site1 and the producer /root/site1//site2.

1. *ar ndnsec-keygen /root | tee root.ndncert | ndnsec-cert-install -* : the root authority generates its root certificate and install it.

2. *ar cp root.ndncert ../cs* : the consumer trusts in ar, so it must have a copy of its certificate as *trust anchor*.

3. *a1 ndnsec-keygen /root/site1 >site1.req* : the sub-authority generates a pair of keys and it prepares a request for ar's signature.
   a1 cp site1.req ../ar/
   ar gedit site1.req : Note that if the next command returns an input error, probably it happened that there are some extra-characters in the request file. If it happens, just eliminates the first characters of the request, typically 4-5 (you will notice about the true start of the file).

4. *ar ndnsec-certgen -N /root/site1 -s /root site1.req >site1.ndncert* The root authority generates and signs the sub-authority certificate, starting from the request with the key that the sub-authority previously provided to it.
   ar cp site1.ndncert ../a1/

5. At this time, both root-authority and sub-authority install the sub-authority certificate:

a1 ndnsec-cert-install -f site1.ndncert
ar ndnsec-cert-install -f site1.ndncert
Now it is necessary repeating the same steps between the producer and a1. Here a1 has the same role of ar in the previous case, while pr has the same role that a1 had before. Since they are the same steps already seen, I will not comment it.

6. *pr ndnsec-keygen /root/site1/site2 >site2.req*
7. *pr cp site2.req ../a1/*
8. *a1 ndnsec-certgen -N /root/site1/site2 -s /root/site1 site2.req >site2.ndncert*
9. *a1 gedit site2.ndncert* (if necessary)
10. *a1 cp site2.ndncert ../pr/*
11. *pr ndnsec-cert-install -f site2.ndncert*
12. *a1 ndnsec-cert-install -f site2.ndncert*

The next step is announcing the prefixes with **nlsrc**, for example.

- pr nlsrc advertise /root/site1/site2
- a1 nlsrc advertise /root/site1/KEY

  NB: if, after this command, you obtain **private key does not exist error**, just re-install site1 certificate *(a1 ndnsec-cert-install -f site1.ndncert)*, then re-type the advertise command.

- ar nlsrc advertise /root/KEY

  NB: if, after this command, you obtain **private key does not exist error**, just re-install root certificate *(ar ndnsec-cert-install -f root.ndncert)*, then re-type the advertise command.

At this point is possible verifying that all was distributed properly, with the command *NODE ndnsec list -c*.
Running it on the producer and on the two authorities, we should able to see something or similar:

On the Root Authority:

```
* /root
+ ->* /root/ksk-1475161471091
+ ->* /root/KEY/ksk-1475161471091/ID-CERT/%FD%00%00%01Wvy%BB%E8

/root/site1
+ ->* /root/site1/ksk-1475161481253
+ ->* /root/KEY/site1/ksk-1475161481253/ID-CERT/%FD%00%00%01Wvz5%E2
```

On the sub-authority:

```
/root/site1
+ ->* /root/site1/ksk-1475161481253
+ ->* /root/KEY/site1/ksk-1475161481253/ID-CERT/%FD%00%00%01Wvz5%E2

/root/site1/site2
+ ->* /root/site1/site2/ksk-1475161530970
+ ->* /root/site1/KEY/site2/ksk-1475161530970/ID-CERT/%FD%00%00[...]
```

On the producer:

```
/root/site1/site2
+->* /root/site1/site2/ksk-1475161530970
+->* /root/site1/KEY/site2/ksk-1475161530970/ID-CERT/%FD%00%00[...]
```

Now it is possible launching the applications (producer, authorities, consumer, gateway if desired) on each node: let's see the main features of these application.

Producer: the producer is a normal producer that sign the data with its own identity before sending.

```
private:
  Face myFace;
  KeyChain myKeyChain;

void
  onInterest(const InterestFilter& filter, const Interest& interest)
  {
        //[...]
    shared_ptr<Data> dataPacket = make_shared<Data>();
    dataPacket->setName(dataName);
    dataPacket->setContent(reinterpret_cast<const uint8_t*>
                (content.c_str()), content.size());

    Name prodIdentity("/root/site1/site2");
                //the same used for the certificate distribution
    myKeyChain.signByIdentity(*data, prodIdentity);
    myFace.put(*data);
        //[...]
  }
```

Consumer: the consumer should express an Interest and calling the method validate. This latter has a reference for two callbacks, the first one (onDataValSuccess) is called if the verification was entirely fine, the second one (onDataValidationFail) if the validation failed. Let's see the code:

```
  void
  run()
  {
    Interest interest(Name("/root/site1/site2"));
    interest.setInterestLifetime(time::milliseconds(1000));
    interest.setMustBeFresh(true);
    myFace->expressInterest(interest,
                            bind(&Consumer::onData, this, _1, _2),
                            bind(&Consumer::onTimeout, this, _1));
    myFace->processEvents();
  }
```

```
  void
  onData(const Interest& interest, const Data& data)
  {
    std::cout<< "Calling validate on" << interest << std::endl;
    m_validator->validate(data, bind(&Consumer::onDataValSuccess,
      this,_1) ,bind(&Consumer::onDataValidationFailed,this,_1,_2));
  }
```

```
private:
  void
  onDataValidationFailed(const shared_ptr<const Data>& data,
                         const std::string& failureInfo)
  {
```

```
    std::cout << "failed" << failureInfo << std::endl;
  }

  void
  onDataValSuccess(const ndn::shared_ptr<const ndn::Data>& data)
  {
    std::cout<< "Validate OK " << std::endl;
    std::string message(reinterpret_cast<const char*>
                        (data->getContent().value()),
                        data->getContent().value_size());
    std::cout << "msg: " << message << std::endl;
  }
```

What is the m_validator object? The verification of both interest and data's packets could be done through a *Validator*, that is a virtual class that implements the method *CheckPolicy*, one override for data and one for interest. It checks if both the packet and the signer respect some policies and if their signature could be verified. There are **two types** of Validator, one based on a configuration file and one based on the regex, in-code. For this example I will use the class ValidatorRegex, for the next section (How to validate an interest) I will use the ValidatorConfig class.
So, the class consumer should #include <ndn-cxx/security/validator-regex.hpp> header and it should define how to validate the data packets that it receives.

```
private:
  shared_ptr<Face> myFace;
  shared_ptr<ValidatorRegex> m_validator;

  void init()
  {
    myFace = make_shared<Face>();
    m_validator = make_shared<ValidatorRegex>(*myFace);

    m_validator->addDataVerificationRule(ndn::make_shared
          <ndn::SecRuleRelative>("^(<>*)$",
          "^([^<KEY>]*)<KEY>(<>*)<ksk-.*><ID-CERT>$",
          ">", "\\1", "\\1\\2", true));

    ndn::shared_ptr<ndn::IdentityCertificate> anchor =
    ndn::io::load<ndn::IdentityCertificate>("/tmp/cs/root.ndncert");

    if (static_cast<bool>(anchor))
      {
        BOOST_ASSERT(anchor->getName().size() >= 1);

        m_validator->addTrustAnchor(anchor);
      }
    else {
      throw "invalid certificate";
    }
  }
```

Briefly (you can find more details about how to write a validator on here): The first parameter of the method addDataVerificationRule is a regex that specifies the conditions on the data name, the second is for the conditions on the KeyLocator field, the last two explain how to constrain the KeyLocator using the information extracted from both Data packet and KeyLocator name. I advice to read the document linked to well understand the mechanism, for our experiments is enough knowing that with this regex we can match all the

identity certificates. Then, there is a part concerning the anchor. An anchor is an authority in which the consumer trust, so we linked the position of the root certificate that we copied in the point number 2 of the commands list, above. That means: *the consumer cs trust in the root authority ar, and it has a copy of root's certificate in the folder /tmp/cs. When the consumer meets a certificate signed by ar, it will immediately trust it.*

The authorities - ar and a1: The only purpose of the authorities when the scenario runs is just distributing the certificates that they signed offline. In order to implement a scalable certificate distribution, I used the method based on *repo-ng*, the implementation of the NDN Repository. Another possible method is using *ndns*. Then, first of all it is necessary having a look and, if necessary, modifying the configuration file on repo-ng, that has to be stored in */usr/local/etc/ndn* folder. Following, an example of this file for this scenario.

```
repo
{
  data {
    ;list of Data prefixes to register
    prefix "ndn:/root/KEY"
    prefix "ndn:/root/site1/KEY"
  }
  command {
    ;list of command prefixes to register
    prefix "ndn:/root/KEY"
    prefix "ndn:/root/site1/KEY"
  }
  storage {
    method "sqlite"  ;just sqlite is allowed now
    path "/tmp/r"    ;where data will stored
    max-packets 100000 ;max number of packets allowed to be stored
  }
  tcp_bulk_insert {
    ;Section to enable TCP bulk insert capability
    ;We will use this capability to insert the certificates in repo
    host "localhost"
    port 7376
  }
  validator {
    trust-anchor
    {
        type any
    }
  }
}
```

The next step is launching repo-ng in both ar and a1 nodes, and publishing the certificates:

```
a1 /home/bertolino/Desktop/mini-ndn/repo-ng/build/ndn-repo-ng &
ar /home/bertolino/Desktop/mini-ndn/repo-ng/build/ndn-repo-ng &
ar base64 -d site1.ndncert | nc localhost 7376
a1 base64 -d site2.ndncert | nc localhost 7376
ar /home/bertolino/Desktop/mini-ndn/repo-ng/build/tools/repo-ng-ls
```

Finally, just run the producer application, the gateway application if any, and the consumer application. The last command is to verify that the publication procedure performed well. This should work, if not the log files could help.

4 How to sign and to verify a signed interest

This scenario is really similar to the precedent, so I will not repeat the comments already done, for each command. I suggest you to read the previous section before reading this one, just to have a more clear vision. However, in this scenario the topology is composed by a producer P, a consumer C, an authority R and an intermediate gateway GW. Both producer and consumer trust in the authority, that signs their certificates. The Consumer C send a signed interest that P is able to satisfy. Then P verify C's certificate, if it is valid it prepares and it signs a data packet. This data packet arrives to C that validates it and, if all is OK, the data are displayed.

The topology file used is the following one:

```
[nodes]
c: _ nfd-log-level=DEBUG
p: _ nfd-log-level=DEBUG
r: _ nfd-log-level=DEBUG
gw: _ nfd-log-level=DEBUG
[links]
c:gw delay=10ms
r:gw delay=10ms
p:gw delay=10ms
```

Then, it is necessary repeating the certificate distribution. Since it is very similar to the previous case, please refer to it for the comments about the commands. They could be useful, with the recommendations, whether an error occurs.

Differently from the previous section, where a root authority signed the certificate of a sub-authority that signed the producer's certificate, here the root authority signs both the producer's and consumer's certificate. The steps are:

1. *r ndnsec-keygen /root — tee root.ndncert — ndnsec-cert-install -*
2. *r cp root.ndncert ../p*
3. *p ndnsec-keygen /root/site1 >site1.req*
4. *p cp site1.req ../r/*
5. *r ndnsec-certgen -N /root/site1 -s /root site1.req >site1.ndncert*
6. *r cp site1.ndncert ../p*
7. *p ndnsec-cert-install -f site1.ndncert*
8. *r ndnsec-cert-install -f site1.ndncert*
9. Repeat the steps 2-8 substituting every p with c and every *site1* with *site2*
10. *p nlsrc advertise /root/site1* (see previous section if *private key does not exist* error appears)
11. *r nlsrc advertise /root/KEY* (see previous section if *private key does not exist* error appears)

Then, exactly like before, you could launch *repo-ng* on the **authority** node and publish the two certificates:

```
repo
{
  data {
    ;list of Data prefixes to register
    prefix "ndn:/root/KEY"
  }
  command {
    ;list of command prefixes to register
    prefix "ndn:/root/KEY"
  }
  storage {
    method "sqlite"  ;just sqlite is allowed now
    path "/tmp/r"    ;where data will stored
    max-packets 100000 ;max number of packets allowed to be stored
  }
  tcp_bulk_insert {
    ;Section to enable TCP bulk insert capability
    ;We will use this capability to insert the certificates in repo
    host "localhost"
    port 7376
  }
  validator {
    trust-anchor
    {
        type any
    }
  }
}
```

```
r /home/bertolino/Desktop/mini-ndn/repo-ng/build/ndn-repo-ng &
r base64 -d site1.ndncert | nc localhost 7376
r base64 -d site2.ndncert | nc localhost 7376
r /home/bertolino/Desktop/mini-ndn/repo-ng/build/tools/repo-ng-ls
```

The **consumer:** the consumer this time is simpler than the producer, because it should verify the data, like the previous case, and just signing the interest. In order to validate the data, this time I used a ValidatorConfig object and not a ValidatorRegex, so I included <ndn-cxx/security/validator-config.hpp> header. The main important parts of the code are:

```
private:
  shared_ptr<Face> myFace;
  shared_ptr<ValidatorConfig> myValidator;
  KeyChain myKeyChain;

  public:
  void init()
  {
    myFace = make_shared<Face>();
    myValidator = make_shared<ValidatorConfig>(*myFace);
    myValidator->load("/home/bertolino/Desktop/mini-ndn/ndn_utils/
validator-config-base.conf");
  }

  void
  run()
  {
  /[...]
    Interest interest(Name("/root/site1"));
```

```
    interest.setInterestLifetime(time::milliseconds(4000));
    interest.setMustBeFresh(true);
    Name consumerId("/root/site2");
    myKeyChain.signByIdentity(interest, consumerId);
    myFace->expressInterest(interest,
                            bind(&Consumer::onData, this, _1, _2),
                            bind(&Consumer::onTimeout, this, _1));
    myFace->processEvents();
  }
```

Then the consumer is equal to the previous one. It should call the validate function and it must define the callbacks whether the data is or is not valid. Here, we need a configuration file for the validator, that we loaded in the code. The file is the following one:

```
rule
{
  id "c_rule"
  for data
  filter
  {
    type name   ; condition on data name
    name /root
    relation is-prefix-of
  }
  checker
  {
    type hierarchical
    sig-type rsa-sha256
  }
}
trust-anchor
{
  type file
  file-name /tmp/c/root.ndncert
}
```

Conceptually it is similar to the ValidatorRegex, but using a configuration file is more user friendly. Please, if you want more details about each field meaning, just consult the link provided in the previous section.

The **producer:** the producer is a little more complicated, because it must be able to verify both interests and data. Indeed it needs to verify the interest of the consumer, but verifying the interest involved in requesting the consumer's certificate, that is a data packet signed by the authority. So the configuration file for the producer validator should have at least two rules! Notice that the interest signed carries the information about signature in the name itself, by standard. The configuration file for the producer is:

```
rule
{
  id "p_rule"
  for interest
  filter
  {
    type name
    name /root/site1
    relation is-prefix-of
  }
  checker
  {
    type customized
```

```
    sig-type rsa-sha256
    key-locator {
      type name
      name /root/KEY/
      relation is-prefix-of
    }
  }
}

rule
{
  id "p1_rule"
  for data
  filter
  {
    type name
    name /root
    relation is-prefix-of
  }
  checker
  {
    type hierarchical
    sig-type rsa-sha256
  }
}

trust-anchor
{
  type file
  file-name /tmp/p/root.ndncert ;
}
```

While the code is identical to the previous one, except for the onInterest method and the ValidatorConfig file

```
public:
 void init()
  {
    myFace = make_shared<Face>();
    myValidator = make_shared<ValidatorConfig>(*myFace);
    myValidator->load("/home/bertolino/Desktop/mini-ndn/ndn_utils
/validator-config-interest.conf");
  }

  void
  run()
  {
    myFace->setInterestFilter("/root/site1",
                  bind(&Producer::onInterest, this, _1, _2),
                  RegisterPrefixSuccessCallback(),
                  bind(&Producer::onRegisterFailed, this, _1, _2));
    myFace->processEvents();
  }

private:
  void
  onInterest(const InterestFilter& filter, const Interest& interest)
  {
    myValidator->validate(interest, bind(&Producer::sendData,
          this, _1), bind(&Producer::onInterestValidationFailed,
                this, _1, _2));
  }
```

```
void
onInterestValidationFailed(const shared_ptr<const Interest>&
                interest, const std::string& failureInfo)
{
  std::cout << "failed " << failureInfo << std::endl;
}

void
sendData(const ndn::shared_ptr<const ndn::Interest>& interest)
{
   //[...]
}

private:
shared_ptr<Face> myFace;
shared_ptr<ValidatorConfig> myValidator;
KeyChain myKeyChain;
```

Finally, just launch the producer and consumer application (and gateway application, if any) and run the simulation.