Interest Packet

Module Description

In this module, we will learn about the Interest packet. What are the main fields inside the Interest packet, how it is communicated in the NDN network, and how it is handled by the NDN forwarder.

Procedure

What is an Interest packet?

An Interest packet is a packet that is used by the consumer to request data from the network. It is a packet that is sent by the consumer to the network to request data from the network.

Interest packet is implemented by `Interest` class in interest.hpp file in the ndn-cxx library.

What are the main fields inside the Interest packet?

The main fields inside the Interest packet are:

1. Name
2. Selectors
3. Nonce
4. Interest Lifetime
5. Hop Limit
6. Must Be Fresh
7. Forwarding Hint

```
Name m_name;
std::vector<Name> m_forwardingHint;
mutable optional<Nonce> m_nonce;
time::milliseconds m_interestLifetime = DEFAULT_INTEREST_LIFETIME;
optional<uint8_t> m_hopLimit;
bool m_canBePrefix = false;
bool m_mustBeFresh = false;
```

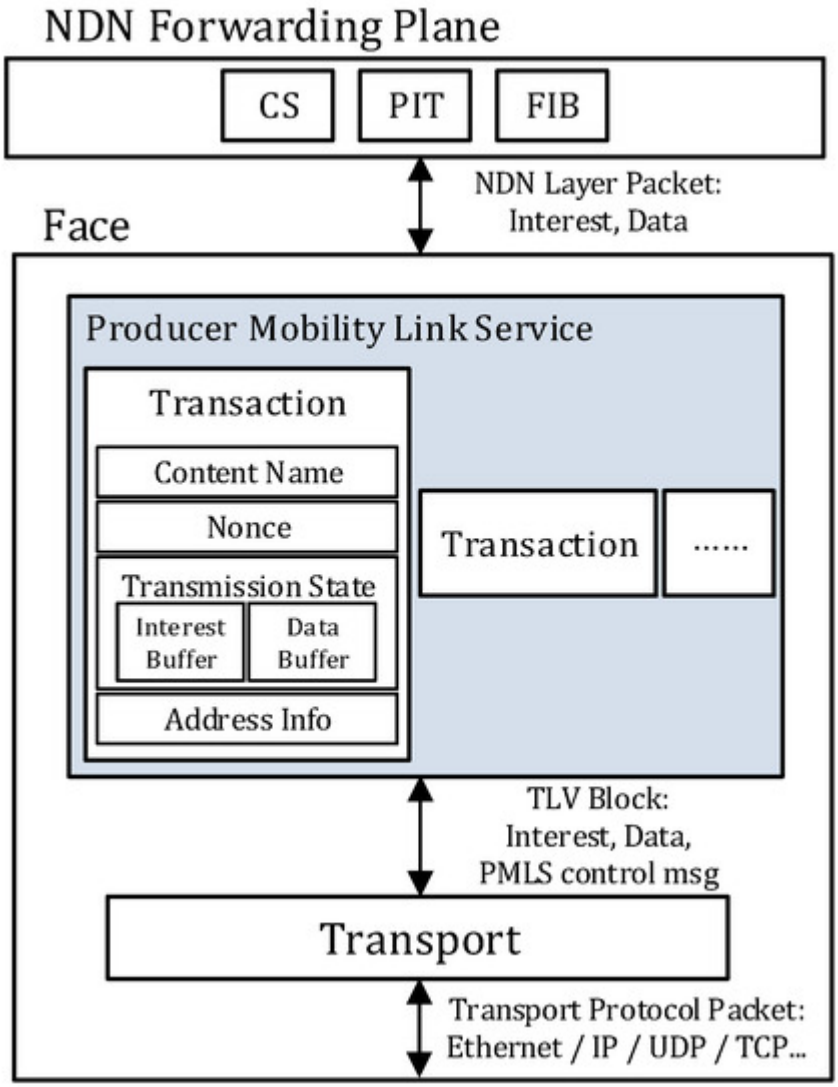
Taken from the private field section of the Interest class.

How is Interest packet communicated in the NDN network?

Interest packet is communicated in the NDN network using the `Face` class. The `Face` class is used to communicate with the NDN network. It is used to send and receive Interest and Data packets.

The abstraction behind the `Face` class is that it is a communication channel between the consumer and the network.

The Interest packet is converted into `Block` format which defines ndn specific TLV format and then it is sent to the network using the `Face` class.



The converted block code is also called as the wire format of the Interest packet or encoding of the Interest packet.

The encoding is done by the method specified in the `Interest` class. The method is `wireEncode`.

```cpp
/** @brief Prepend wire encoding to @p encoder.
 */
template<encoding::Tag TAG>
size_t
wireEncode(EncodingImpl<TAG>& encoder) const;

/** @brief Encode into a Block.
 */
const Block&
wireEncode() const;

/** @brief Decode from @p wire.
 */
void
wireDecode(const Block& wire);
```

```cpp
size_t
MetaInfo::wireEncode(EncodingImpl<TAG>& encoder) const
{
  // MetaInfo ::= META-INFO-TYPE TLV-LENGTH
  //                ContentType?
  //                FreshnessPeriod?
  //                FinalBlockId?
  //                AppMetaInfo*

  size_t totalLength = 0;

  // AppMetaInfo (in reverse order)
  for (const auto& block : m_appMetaInfo | boost::adaptors::reversed) {
    totalLength += prependBlock(encoder, block);
  }

  // FinalBlockId
  if (m_finalBlockId) {
    totalLength += prependNestedBlock(encoder, tlv::FinalBlockId, *m_finalBlockId);
  }

  // FreshnessPeriod
  if (m_freshnessPeriod != DEFAULT_FRESHNESS_PERIOD) {
    totalLength += prependNonNegativeIntegerBlock(encoder, tlv::FreshnessPeriod,
                                                  static_cast<uint64_t>(m_freshnessPeriod.count()));
  }

  // ContentType
  if (m_type != tlv::ContentType_Blob) {
    totalLength += prependNonNegativeIntegerBlock(encoder, tlv::ContentType, m_type);
  }

  totalLength += encoder.prependVarNumber(totalLength);
  totalLength += encoder.prependVarNumber(tlv::MetaInfo);
  return totalLength;
}
```

This above code snipptet is from `meta-info.cpp` class in ndn-cxx library.

I can't find the source code implementation of the `wireEncode` method, but i found source code implementation in `python` bindings and also a webpage that explains the encoding of the Interest packet.

The source code implementation of the `wireEncode` method in python bindings is here

The webpage that explains the encoding of the Interest packet is here

```
Interest ::= INTEREST-TYPE TLV-LENGTH
                Name
                Selectors?
                Nonce
                InterestLifetime?
                ForwardingHint?

Selectors ::= SELECTORS-TYPE TLV-LENGTH
                 MinSuffixComponents?
                 MaxSuffixComponents?
                 PublisherPublicKeyLocator?
                 Exclude?
                 ChildSelector?
                 MustBeFresh?

MinSuffixComponents ::= MIN-SUFFIX-COMPONENTS-TYPE TLV-LENGTH
                           NonNegativeInteger

MaxSuffixComponents ::= MAX-SUFFIX-COMPONENTS-TYPE TLV-LENGTH

InterestLifetime ::= INTEREST-LIFETIME-TYPE TLV-LENGTH nonNegativeInteger

Nonce ::= NONCE-TYPE TLV-LENGTH
            NonceValue
```

Only few fields are shown in the above picture.

How is Interest packet handled by the NDN forwarder?

The Interest packet is handled by the NDN forwarder using the `Forwarder` class. The `Interest` packet reaches the realm of the forwarder using the `Face` class. The `Face` class is used to send and receive Interest and Data packets.

On `Face` class receiving the Interest packet, the `OnIncomingInterest` method of the `Forwarder` class is called. The `OnIncomingInterest` method of the `Forwarder` class is used to handle the Interest packet.

The `Face` class calls `OnIncomingInterest` doesn't directly call the `OnIncomingInterest` method of the `Forwarder` class. It uses the `LinkService` class to call the `OnIncomingInterest` method of the `Forwarder` class.

That is called by the Transport class.

```cpp
void
    Forwarder::onIncomingInterest(const Interest& interest, const FaceEndpoint& ingress)
  {
    // receive Interest
    NFD_LOG_DEBUG("onIncomingInterest in=" << ingress << " interest=" << interest.getName());
    interest.setTag(make_shared<lp::IncomingFaceIdTag>(ingress.face.getId()));
    ++m_counters.nInInterests;

    // drop if HopLimit zero, decrement otherwise (if present)
    if (interest.getHopLimit()) {
      if (*interest.getHopLimit() == 0) {
        NFD_LOG_DEBUG("onIncomingInterest in=" << ingress << " interest=" << interest.getName()
          << " hop-limit=0");
        ++ingress.face.getCounters().nInHopLimitZero;
        // drop
        return;
      }
      const_cast<Interest&>(interest).setHopLimit(*interest.getHopLimit() - 1);
    }

    // /localhost scope control
    bool isViolatingLocalhost = ingress.face.getScope() == ndn::nfd::FACE_SCOPE_NON_LOCAL &&
      scope_prefix::LOCALHOST.isPrefixOf(interest.getName());
    if (isViolatingLocalhost) {
      NFD_LOG_DEBUG("onIncomingInterest in=" << ingress
        << " interest=" << interest.getName() << " violates /localhost");
      // drop
      return;
    }

    // detect duplicate Nonce with Dead Nonce List
    bool hasDuplicateNonceInDnl = m_deadNonceList.has(interest.getName(), interest.getNonce());
    if (hasDuplicateNonceInDnl) {
      // goto Interest loop pipeline
      this->onInterestLoop(interest, ingress);
      return;
    }

    // strip forwarding hint if Interest has reached producer region
    if (!interest.getForwardingHint().empty() &&
      m_networkRegionTable.isInProducerRegion(interest.getForwardingHint())) {
      NFD_LOG_DEBUG("onIncomingInterest in=" << ingress
        << " interest=" << interest.getName() << " reaching-producer-region");
      const_cast<Interest&>(interest).setForwardingHint({});
    }

    // PIT insert
    shared_ptr<pit::Entry> pitEntry = m_pit.insert(interest).first;

    // detect duplicate Nonce in PIT entry
    int dnw = fw::findDuplicateNonce(*pitEntry, interest.getNonce(), ingress.face);
    bool hasDuplicateNonceInPit = dnw != fw::DUPLICATE_NONCE_NONE;
    if (ingress.face.getLinkType() == ndn::nfd::LINK_TYPE_POINT_TO_POINT) {
      // for p2p face: duplicate Nonce from same incoming face is not loop
      hasDuplicateNonceInPit = hasDuplicateNonceInPit && !(dnw & fw::DUPLICATE_NONCE_IN_SAME);
    }
    if (hasDuplicateNonceInPit) {
      // goto Interest loop pipeline
      this->onInterestLoop(interest, ingress);
      m_strategyChoice.findEffectiveStrategy(*pitEntry).afterReceiveLoopedInterest(ingress, interest, *pitEntry);
      return;
    }

    // is pending?
    if (!pitEntry->hasInRecords()) {
      m_cs.find(interest,
        [=](const Interest& i, const Data& d) { onContentStoreHit(i, ingress, pitEntry, d); },
        [=](const Interest& i) { onContentStoreMiss(i, ingress, pitEntry); });
    }
    else {
      this->onContentStoreMiss(interest, ingress, pitEntry);
    }
  }
```

There are many other functions in the `Forwarder` class that are used to handle the Interest and Data packets.

Use of Interest packet

The Interest packet is used by the consumer to request data from the network. It is used by the consumer to request data from the network.

Another use of the Interest packet to communicate with the network. For sending information to neighbouring router's, the Interest packet is used.

This in more detail is discussed in Communicate with different nodes