

# Applications

---

## Prerequisites

1. Faces
2. Forwarder
3. Interest-in Pipeline
4. Core of ndnSIM

## Module Description

Applications are the dynamic part of the system, that can introduce new content into the system, or can request content from the system. Applications are the part of the system that can be used to simulate the real world applications that are built on top of the NDN stack.

Applications are not the only factor that introduces non-determinism in the ndnSIM system, but they are one of the major factors that introduces non-determinism in the system. Other factors that introduce non-determinism in the system are the forwarding strategies, the network topology, the network conditions, neighbor nodes, etc.

Applications use the event-driven simulator provided by the ns-3 to facilitate the communication between the nodes. These call the simulator to schedule events, and the simulator calls the application to process the events. This is how the communication between the nodes is facilitated.

### How applications communicate with forwarder

Applications communicate with the forwarder using the [Face](#) class. Each application has a face that is used to communicate with the forwarder. The face is used to send the Interest packets to the forwarder, and the face is used to receive the Data packets from the forwarder. The face is also used to receive the Interest packets from the forwarder, and the face is used to send the Data packets to the forwarder. The abstraction used in [Face](#) class but [AppLinkService](#) is used by the application to communicate with [Face](#) which in turns communicate with the forwarder.

The face is considered as having local scope as the face is only used to communicate with the forwarder. It is not used to communicate with the other applications. It is used to receive the Interest and Data packets from the forwarder using the [OnInterest](#) and [onData](#) methods.

The face is created when the applications starts.

```

1 // Application Methods
2 void
3 App::StartApplication() // Called at time specified by Start
4 {
5     NS_LOG_FUNCTION_NOARGS();
6
7     NS_ASSERT(m_active != true);
8     m_active = true;
9
10    NS_ASSERT_MSG(GetNode()->GetObject<L3Protocol>() != 0,
11                  "Ndn stack should be installed on the node " << GetNode());
12
13    // step 1. Create a face
14    auto appLink = make_unique<AppLinkService>(this);
15    auto transport = make_unique<NullTransport>("appFace://", "appFace://",
16                                              "::ndn::nfd::FACE_SCOPE_LOCAL);
17    // @TODO Consider making AppTransport instead
18    m_face = std::make_shared<Face>(std::move(appLink), std::move(transport));
19    m_appLink = static_cast<AppLinkService*>(m_face->getLinkService());
20    m_face->setMetric(1);
21
22    // step 2. Add face to the Ndn stack
23    GetNode()->GetObject<L3Protocol>()->addFace(m_face);
24 }

```

The face is destroyed when the application stops.

```

1 void
2 App::StopApplication() // Called at time specified by Stop
3 {
4     NS_LOG_FUNCTION_NOARGS();
5
6     if (!m_active)
7         return; // don't assert here, just return
8
9     m_active = false;
10
11    m_face->close();
12 }
13

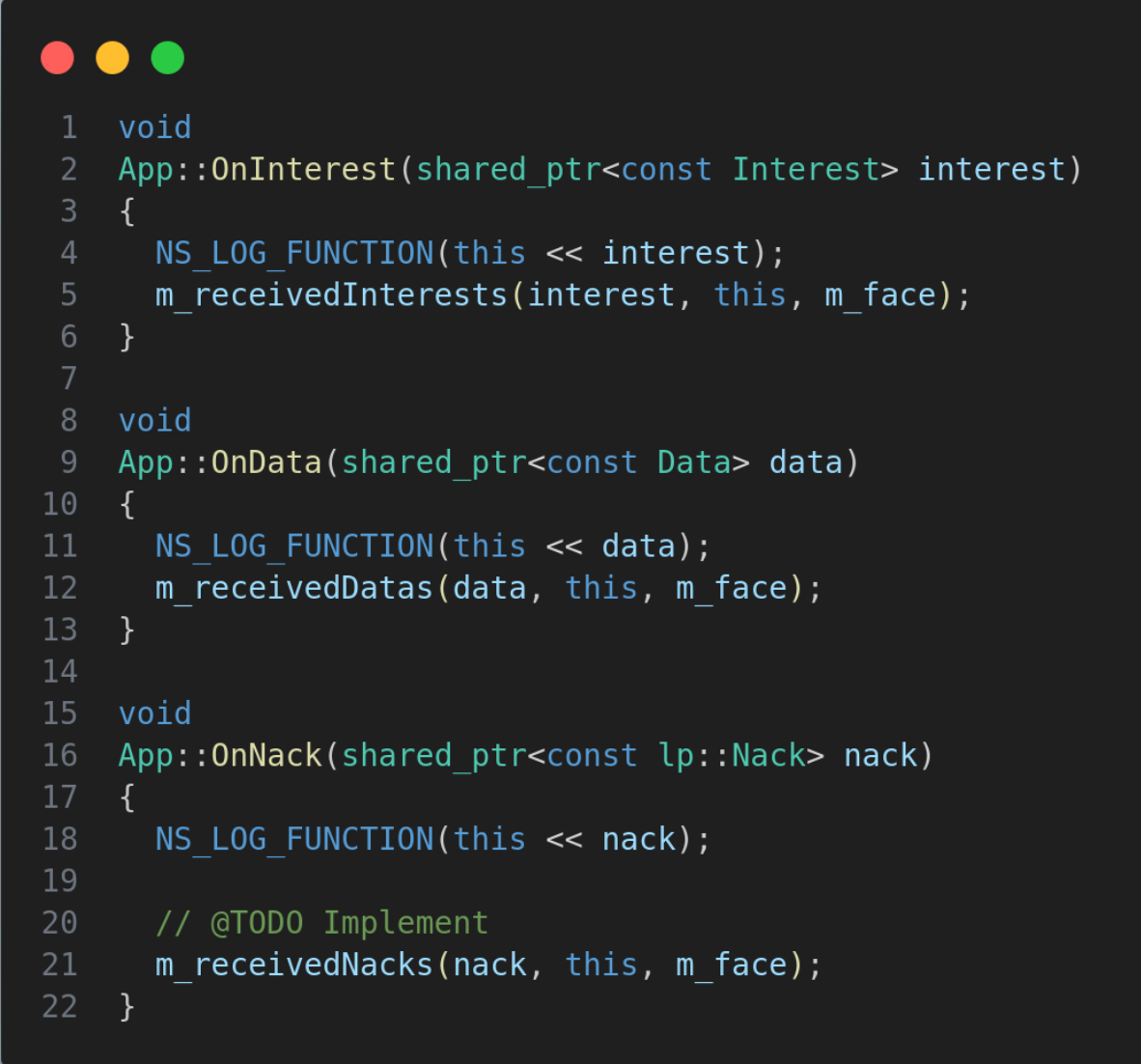
```

There are two types of applications that are part of the ndnSIM system. These are:

1. Producer
2. Consumer

Technically, these two types of applications are not different in terms of the code, but they are different in terms of the functionality. The producer application is used to introduce new content into the system, and the consumer application is used to request content from the system.

Both derive from the same Application base class. The Application base class is used to provide the common functionality that is shared by both the producer and the consumer applications.



```
1  void
2  App::OnInterest(shared_ptr<const Interest> interest)
3  {
4      NS_LOG_FUNCTION(this << interest);
5      m_receivedInterests(interest, this, m_face);
6  }
7
8  void
9  App::OnData(shared_ptr<const Data> data)
10 {
11     NS_LOG_FUNCTION(this << data);
12     m_receivedDatas(data, this, m_face);
13 }
14
15 void
16 App::OnNack(shared_ptr<const lp::Nack> nack)
17 {
18     NS_LOG_FUNCTION(this << nack);
19
20     // @TODO Implement
21     m_receivedNacks(nack, this, m_face);
22 }
```

These 3 functions are used to provide the common functionality that is shared by both the producer and the consumer applications. Few other functions are added to the producer and the consumer applications to provide the functionality that is specific to the producer and the consumer applications.

Example of the functions that are added to the producer and the consumer applications are:

1. Consumer: `SendPacket`, `onTimeout`
2. Producer: `m_freshness`, `m_signature`, etc.

```
1  class Producer : public App {
2  public:
3      static TypeId
4      GetTypeId(void);
5
6      Producer();
7
8      // inherited from NdnApp
9      virtual void
10     OnInterest(shared_ptr<const Interest> interest);
11
12 protected:
13     // inherited from Application base class.
14     virtual void
15     StartApplication(); // Called at time specified by Start
16
17     virtual void
18     StopApplication(); // Called at time specified by Stop
19
20 private:
21     Name m_prefix;
22     Name m_postfix;
23     uint32_t m_virtualPayloadSize;
24     Time m_freshness;
25
26     uint32_t m_signature;
27     Name m_keyLocator;
28 };
```

## Consumer Types

There are many different types of consumers that are part of the ndnSIM system. These are:

1. [Consumer CBR](#)
2. [Consumer Zipf-Mandelbrot](#)
3. [Consumer Consumer Batches](#)
4. [Consumer Window](#)

For more info about the individual applications, please refer to the individual application's documentation in the ndnSIM website.

## How consumer applications send Interest packets

Consumer applications send Interest packets using `SendPacket` method. This method is used to send the Interest packets to the forwarder. The forwarder then forwards the Interest packets to the other nodes in the network. The Interest packets are forwarded to the other nodes in the network using the forwarding strategy that is used by the forwarder. The forwarding strategy is used to decide which node to forward the Interest packet to.

```

1  void
2  Consumer::SendPacket()
3  {
4      if (!m_active)
5          return;
6
7      NS_LOG_FUNCTION_NOARGS();
8
9      uint32_t seq = std::numeric_limits<uint32_t>::max(); // invalid
10
11      while (m_retXSeqs.size()) {
12          seq = *m_retXSeqs.begin();
13          m_retXSeqs.erase(m_retXSeqs.begin());
14          break;
15      }
16
17      if (seq == std::numeric_limits<uint32_t>::max()) {
18          if (m_seqMax != std::numeric_limits<uint32_t>::max()) {
19              if (m_seq >= m_seqMax) {
20                  return; // we are totally done
21              }
22          }
23
24          seq = m_seq++;
25      }
26
27      //
28      shared_ptr<Name> nameWithSequence = make_shared<Name>(m_interestName);
29      nameWithSequence->appendSequenceNumber(seq);
30
31
32      // Interest packet is created
33      shared_ptr<Interest> interest = make_shared<Interest>();
34      interest->setNonce(m_rand->GetValue(0, std::numeric_limits<uint32_t>::max()));
35      interest->setName(*nameWithSequence);
36      interest->setCanBePrefix(false);
37      time::milliseconds interestLifeTime(m_interestLifeTime.GetMilliSeconds());
38      interest->setInterestLifetime(interestLifeTime);
39
40      NS_LOG_INFO("> Interest for " << seq);
41
42      WillSendOutInterest(seq);
43
44      m_transmittedInterests(interest, this, m_face);
45
46      // send out Interest
47      m_appLink->onReceiveInterest(*interest);
48
49      ScheduleNextPacket();
50  }

```

## How producer applications send Data packets

Producer applications doesn't have specific method to send the Data packets. This is the case because, the producers doesn't send data packet without interest packet. So when an interest packet is received, on the same instance, a data packet is created and sent to the consumer.

This has special usage even when we communicate between the nodes.

```
1 void
2 Producer::OnInterest(shared_ptr<const Interest> interest)
3 {
4     App::OnInterest(interest); // tracing inside
5
6     NS_LOG_FUNCTION(this << interest);
7
8     if (!m_active)
9         return;
10
11     Name dataName(interest->getName());
12     dataName.append(m_postfix);
13     dataName.appendVersion();
14
15     // Data packet is initialized using just the name of the interest received
16     auto data = make_shared<Data>();
17     data->setName(dataName);
18     data->setFreshnessPeriod(::ndn::time::milliseconds(m_freshness.GetMilliseconds()));
19
20     data->setContent(make_shared< ::ndn::Buffer>(m_virtualPayloadSize));
21
22     SignatureInfo signatureInfo(static_cast< ::ndn::tlv::SignatureTypeValue>(255));
23
24     if (m_keyLocator.size() > 0) {
25         signatureInfo.setKeyLocator(m_keyLocator);
26     }
27
28     data->setSignatureInfo(signatureInfo);
29
30     ::ndn::EncodingEstimator estimator;
31     ::ndn::EncodingBuffer encoder(estimator.appendVarNumber(m_signature), 0);
32     encoder.appendVarNumber(m_signature);
33     data->setSignatureValue(encoder.getBuffer());
34
35     NS_LOG_INFO("node(" << GetNode()->GetId() << ") responding with Data: " << data->getName());
36
37     // to create real wire encoding
38     data->wireEncode();
39
40     m_transmittedDatas(data, this, m_face);
41
42     // send packet to forwarder using appLink
43     m_appLink->onReceiveData(*data);
44 }
```