ndnSIM: a modular NDN simulator

Introduction and Tutorial

http://ndnsim.net

ALEX AFANASYEV
ILYA MOISEENKO
LIXIA ZHANG

University of California, Los Angeles, Computer Science Department

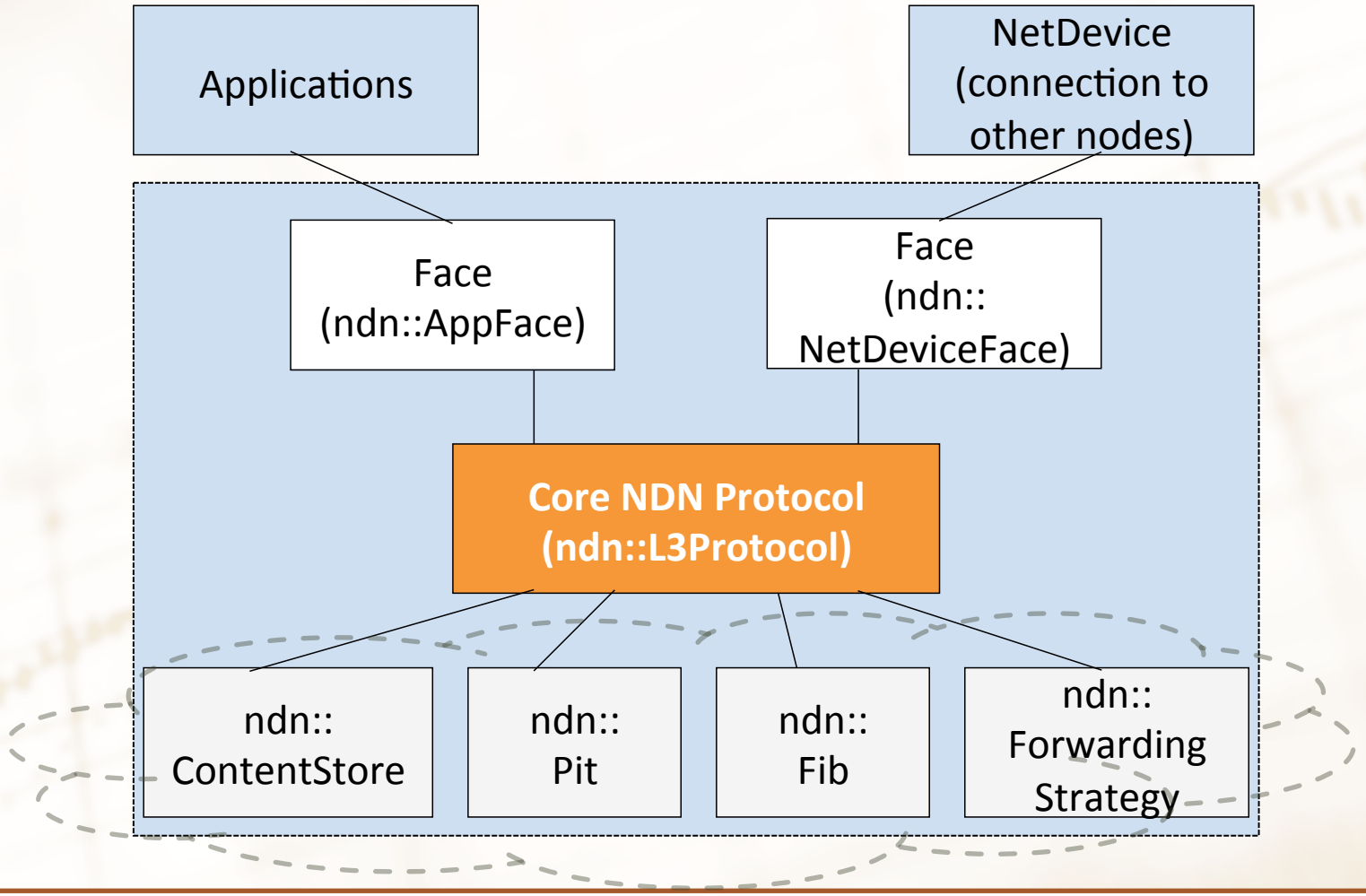
Introduction

- ndnSIM implements all basic NDN operations
- Has packet-level interoperability with CCNx implementation
- Has modular architecture
  - C++ classes for every NDN component
    - Face, PIT, FIB, Content store, and Forwarding strategey
- Allows combining different implementations of core NDN components
  - Different management schemes for PIT
  - Different replacement policies for content store
  - Different forwarding strategies
- Can be easily extended
- Easy to use: plug in and experiment

Ultimate Goal

- Establishing a *common* platform to be used by the community for all CCN/NDN simulation experimentations
  - So that people can compare/replicate results

ndnSIM structure overview



- Abstract interfaces of content store, PIT, FIB, and forwarding strategy.
- Each simulation run chooses specific scheme for each module

ndnSIM usage by early adopters & ourselves

- Forwarding strategy experimentation
  - behavior in the presence of
    - link failures
    - prefix black-holing
    - congestion
  - resiliency of NDN to DDoS attacks (interest flooding)
- Content-store evaluation
  - evaluation different replacement policies
- NDN for car2car communication
  - Evaluations of traffic info propagation protocols
- Exploration of SYNC protocol design
  - Experimentation of multiuser chat application whose design is based on SYNC (chronos)

NDN experimental extensions

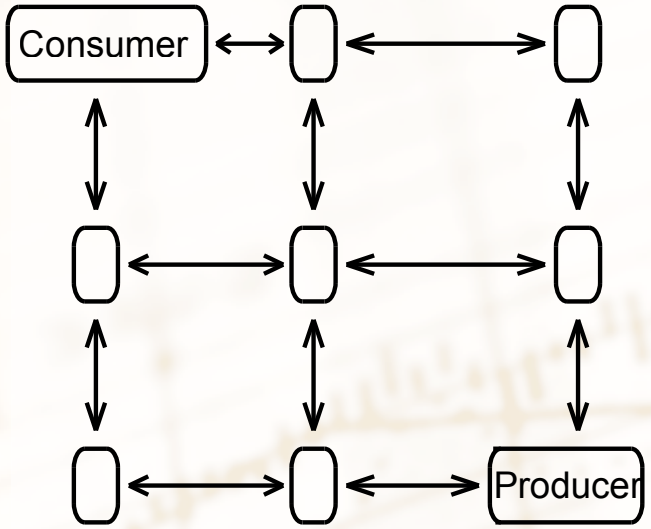
- Interest NACKs to enable more intelligent, adaptive forwarding
- Congestion control by limiting the number of pending Interests
  - per-face
  - per-FIB-entry
  - per-FIB-entry-per-face
- Satisfaction ratio statistics module
  - per-face (incoming/outgoing)
  - per-prefix
  - configurable time granularities
- A initial set of simple application modules

Scalability numbers

- Memory overhead (on average)
  - per simulation node
    - Node without any stacks installed: **0.4 Kb**
    - Node with ndnSIM stack (empty caches and empty PIT): **1.6 Kb**
    - For reference: Node with IP (IPv4 + IPv6) stack: **5.0 Kb**
  - per PIT entry: 1.0 Kb
  - per CS entry: 0.8 Kb
- Processing speed: (
  ~50,000 Interests per
  ~35,000 Interests + Da
- MPI support of NS-3
  - manual network partitioning
  - close to linear scaling with number of cores with good partitioning

Can be optimized by utilizing a simplified packet encoding.
Next release of ndnSIM will have option to choose between ccnx compatibility and processing efficiency

Tutorial by an example



10 Mbps / 10 ms delay

- http://ndnsim.net/examples.html#node-grid-example
- Simple simulation
  - 3x3 grid topology
  - 10Mbps links / 10ms delays
  - One consumer, one producer

NS-3 101: Prepare scenario (C++)

Step 0. Create scenario.cc and place it in <ns-3>/scratch/

Step 1. Include necessary modules

```
#include "core-module.h"
#include "ns3/n
#include "ns3/etwork-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/point-to-point-grid.h"
#include "ns3/ndnSIM-module.h"
using namespace ns3;
```

Step 2. Define **main** function like in any other C++ program

```
int
main (int argc, char *argv[])
{
```

Step 3. Set default parameters for the simulator modules. For example, define that by default all created p2p links will have 10Mbps bandwidth, 10ms delay and DropTailQueue with 20 packets

```
  Config::SetDefault
  ("ns3::PointToPointNetDevice::DataRate", StringValue
  ("10Mbps"));
  Config::SetDefault ("ns3::PointToPointChannel::Delay",
  StringValue ("10ms"));
  Config::SetDefault ("ns3::DropTailQueue::MaxPackets",
  StringValue ("20"));
```

Step 4. Allow overriding defaults from command line

```
  CommandLine cmd; cmd.Parse (argc, argv);
```

Step 5. Define what topology will be simulated. For example, 3x3 grid topology

```
  PointToPointHelper p2p;
  PointToPointGridHelper grid (3, 3, p2p);
  grid.BoundingBox(100,100,200,200);
```

Step 6. Create and install networking stacks, install and schedule applications, define metric logging, etc.

```
  // scenario meat
```

Step 7. Define when simulation should be stopped

```
  Simulator::Stop (Seconds (20.0));
```

Final step. Run simulation

```
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}
```

The same scenario can be also written in Python

C++

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/point-to-point-grid.h"
#include "ns3/ndnSIM-module.h"
using namespace ns3;

int
main (int argc, char *argv[])
{
  Config::SetDefault ("ns3::PointToPointNetDevice::DataRate",
StringValue ("10Mbps"));
  Config::SetDefault ("ns3::PointToPointChannel::Delay",
StringValue ("10ms"));
  Config::SetDefault ("ns3::DropTailQueue::MaxPackets",
StringValue ("20"));

  CommandLine cmd; cmd.Parse (argc, argv);

  PointToPointHelper p2p;
  PointToPointGridHelper grid (3, 3, p2p);
  grid.BoundingBox(100,100,200,200);


  // scenario meat


  Simulator::Stop (Seconds (20.0));

  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}
```

Python

```
from ns.core import *
from ns.network import *
from ns.point_to_point import *
from ns.point_to_point_layout import *
from ns.ndnSIM import *

Config.SetDefault ("ns3::PointToPointNetDevice::DataRate",
StringValue ("10Mbps"))
Config.SetDefault ("ns3::PointToPointChannel::Delay",
StringValue ("10ms"))
Config.SetDefault ("ns3::Drop
StringValue ("20"))

import sys; cmd = CommandL

p2p = PointToPointHelper ()
grid = PointToPointGridHelpe
grid.BoundingBox(100,100,2


# scenario meat


Simulator.Stop (Seconds (20.

Simulator.Run ()
Simulator.Destroy ()
```

Defining scenario in Python is easier and don't require (re)compilation, but not all features of NS-3 and ndnSIM are available in Python interface

The rest of the tutorial is only C++

ndnSIM 101: filling scenario meat

Step 1. Install NDN stack on all nodes (like starting ccnd on a computer)

```
ndn::StackHelper ndnHelper;
ndnHelper.InstallAll ();
```

Step 2. Define which nodes will run applications

```
// Getting containers for the consumer/producer
Ptr<Node> producer = grid.GetNode (2, 2);
NodeContainer consumerNodes;
consumerNodes.Add (grid.GetNode (0,0));
```

Step 3. “Install” applications on nodes

```
ndn::AppHelper cHelper ("ns3::ndn::ConsumerCbr");
cHelper .SetPrefix ("/prefix");
cHelper .SetAttribute ("Frequency", StringValue
("10"));
cHelper .Install (consumerNodes);

ndn::AppHelper pHelper ("ns3::ndn::Producer");
pHelper.SetPrefix ("/prefix");
pHelper.SetAttribute ("PayloadSize",
StringValue("1024"));
pHelper.Install (producer);
```

Step 2. Configure FIB

- manually
- using global routing controller (shown here)

```
ndn::GlobalRoutingHelper ndnGlobalRoutingHelper;
ndnGlobalRoutingHelper.InstallAll ();

// Add /prefix origins to ndn::GlobalRouter
ndnGlobalRoutingHelper.AddOrigins (“/prefix”,
producer);

// Calculate and install FIBs
ndnGlobalRoutingHelper.CalculateRoutes ();
```

Running the simulation (C++)

Option A: like any other program:

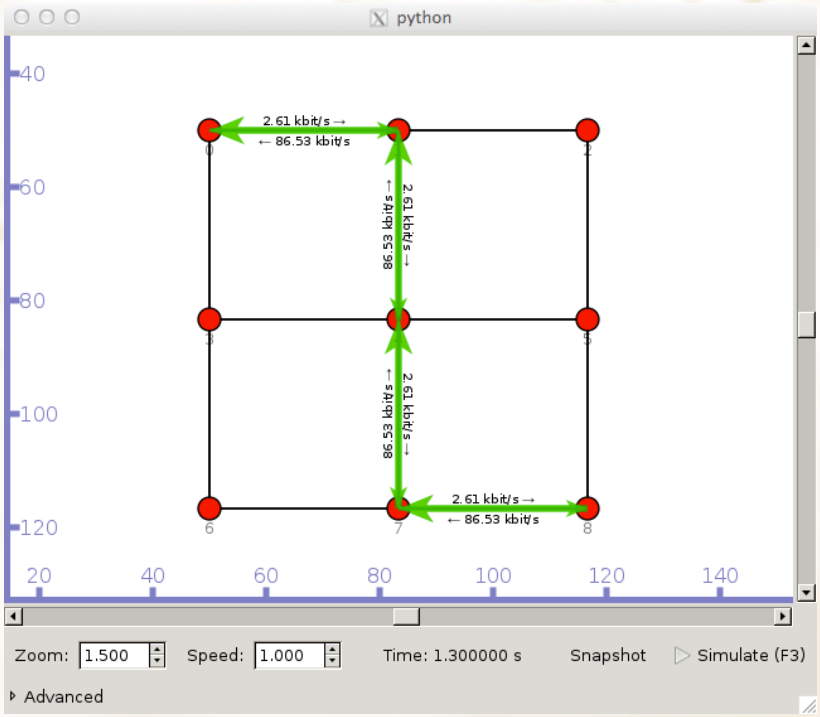
<ns-3>/build/scratch/scenario

Option B: using ./waf helper:

cd <ns-3>; ./waf --run=scenario

Option C: using ./waf helper using visualizer:

./waf --run=scenario --visualize



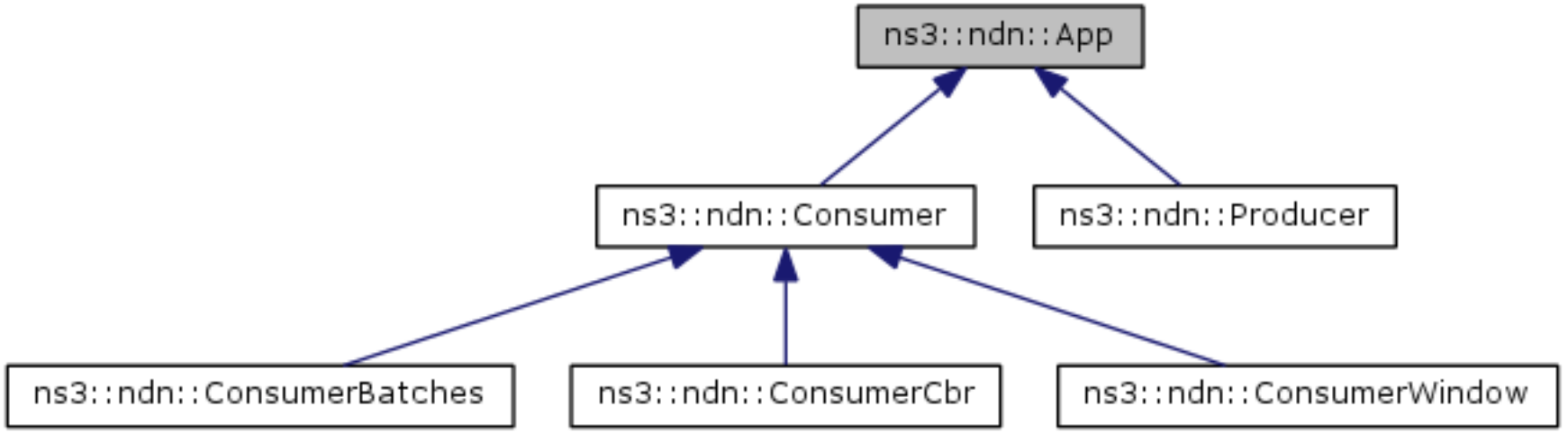
Result if you followed the steps

Same example is on http://ndnsim.net

Beyond the basics

- Select different forwarding strategy, configure cache (size, replacement policy):
  - ndnHelper.SetForwardingStrategy (“ns3::ndn::fw::Flooding”)
    - “ns3::ndn::fw::Flooding”, "ns3::ndn::fw::BestRoute” or your own
  - ndnHelper.SetContentStore ("ns3::ndn::cs::Lru", "MaxSize", "100”)
    - "ns3::ndn::cs::Lru”, "ns3::ndn::cs::Random”, "ns3::ndn::cs::Fifo”
  - ndnHelper.SetPit ("ns3::ndn::pit::Persistent”, “MaxSize”, “1000”)
    - "ns3::ndn::pit::Persistent”, "ns3::ndn::pit::Random"

An initial set of applications



- **ndn::ConsumerCbr**
  - generates Interest traffic with predefined frequency
- **ndn::ConsumerBatches**
  - generates a specified number of Interests at specified points of simulation
- **ndn::Producer**
  - Interest-sink application, which replies every incoming Interest with Data packet

Write your own application (requester)

Step 1. Create a normal C++ class and derive it from ndn::App

Step 2. Define GetTypeId () function (use templates!)
Needed for NS-3 object system

Step 3. Define actions upon start and stop of the application

Step 4. Implement **OnContentObject** method to process requested data:

```
virtual void
OnContentObject (const Ptr<const
ContentObjectHeader> &contentObject,
Ptr<Packet> payload);
```

```
...
class RequesterApp : public App
{
public:
  static TypeId GetTypeId ();

  RequesterApp ();
  virtual ~RequesterApp ();

protected:
  // from App
  virtual void
  StartApplication ()
  {
    App::StartApplication ();
   // send packet for example
  }

  virtual void
  StopApplication ()
  {
     // do cleanup
    App::StopApplication ();
  }
};
```

Write your own application (producer)

Step 0. Do everything as for the requester app

Step 1. Register prefix in FIB (= set Interest filter) in StartApplication

Step 2. Implement **OnInterest** to process incoming interests

```
virtual void
OnInterest (const Ptr<const InterestHeader>
&interest, Ptr<Packet> packet);
```

```
void StartApplication ()
{
...
  Ptr<Fib> fib = GetNode ()->GetObject<Fib> ();
  Ptr<fib::Entry> fibEntry = fib->Add (m_prefix,
m_face, 0);

  fibEntry->UpdateStatus (m_face,
fib::FaceMetric::NDN_FIB_GREEN);
}
```

Write your own forwarding strategy

Step 1. Create a standard C++ class and derive it from ndn::ForwardingStrategy, one of the extensions, or one of the existing strategies

Step 2. Extend or re-implement available forwarding strategy events:

- OnInterest
- OnData
- WillEraseTimedOutPendingInterest
- RemoveFace
- DidReceiveDuplicateInterest
- DidExhaustForwardingOptions
- FailedToCreatePitEntry
- DidCreatePitEntry
- DetectRetransmittedInterest
- WillSatisfyPendingInterest
- SatisfyPendingInterest
- DidSendOutData
- DidReceiveUnsolicitedData
- ShouldSuppressIncomingInterest
- TrySendOutInterest
- DidSendOutInterest
- PropagateInterest
- DoPropagateInterest

```
/**
 * \ingroup ndn
 * \brief Strategy implementing per-FIB entry limits
 */
class SimpleLimits :
    public BestRoute
{
private:
  typedef BestRoute super;

public:
  static TypeId
  GetTypeId ();

  SimpleLimits ();

  virtual void
  WillEraseTimedOutPendingInterest ...

protected:
  virtual bool
  TrySendOutInterest ...

  virtual void
  WillSatisfyPendingInterest ...

private:
  // from Object
  virtual void
  NotifyNewAggregate (); ///< @brief Even when object is
aggregated to another Object

  virtual void
  DoDispose ();
};
```

Write your own cache replacement policy

- Option A:
  - create a class derived from ndn::ContentStore, implementing all interface functions
- Option B:
  - use C++ templates of ndnSIM
    - define “policy traits” (example utils/trie/lru-policy)
      - defines what to do
        - » on insert (e.g., put in front)
        - » on update (e.g., promote to front)
        - » on delete (e.g., remove)
        - » on lookup (e.g., promote to front)
    - instantiate cache class with new policy:
      - template class ContentStoreImpl<lru_policy_traits>;
    - see examples in model/cs/content-store-impl.cc

Try out ndnSIM and let us know your thought/comments/bug reports/new feature requests !

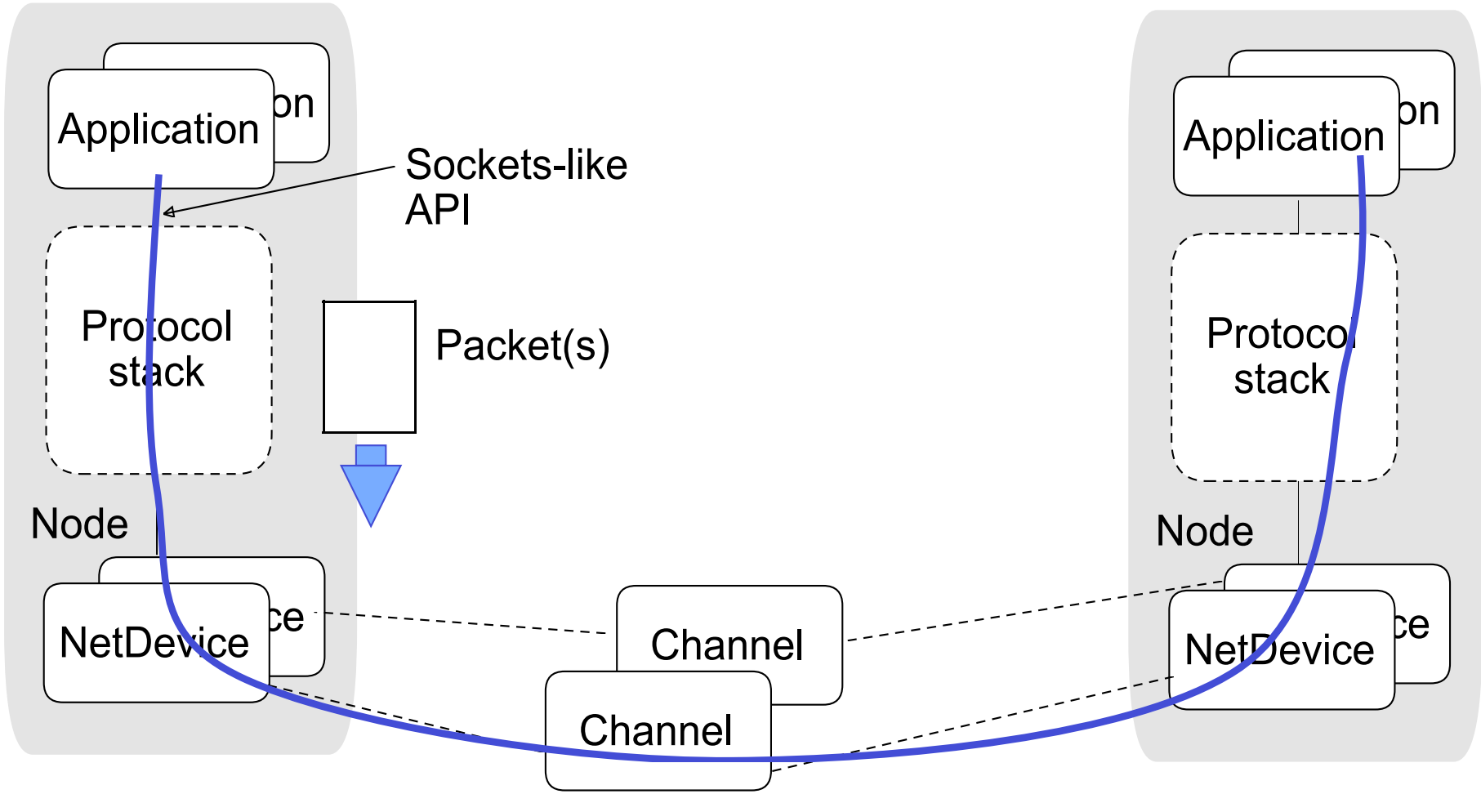
http://ndnsim.net

Come to demo
We'll write a new simple forwarding strategy

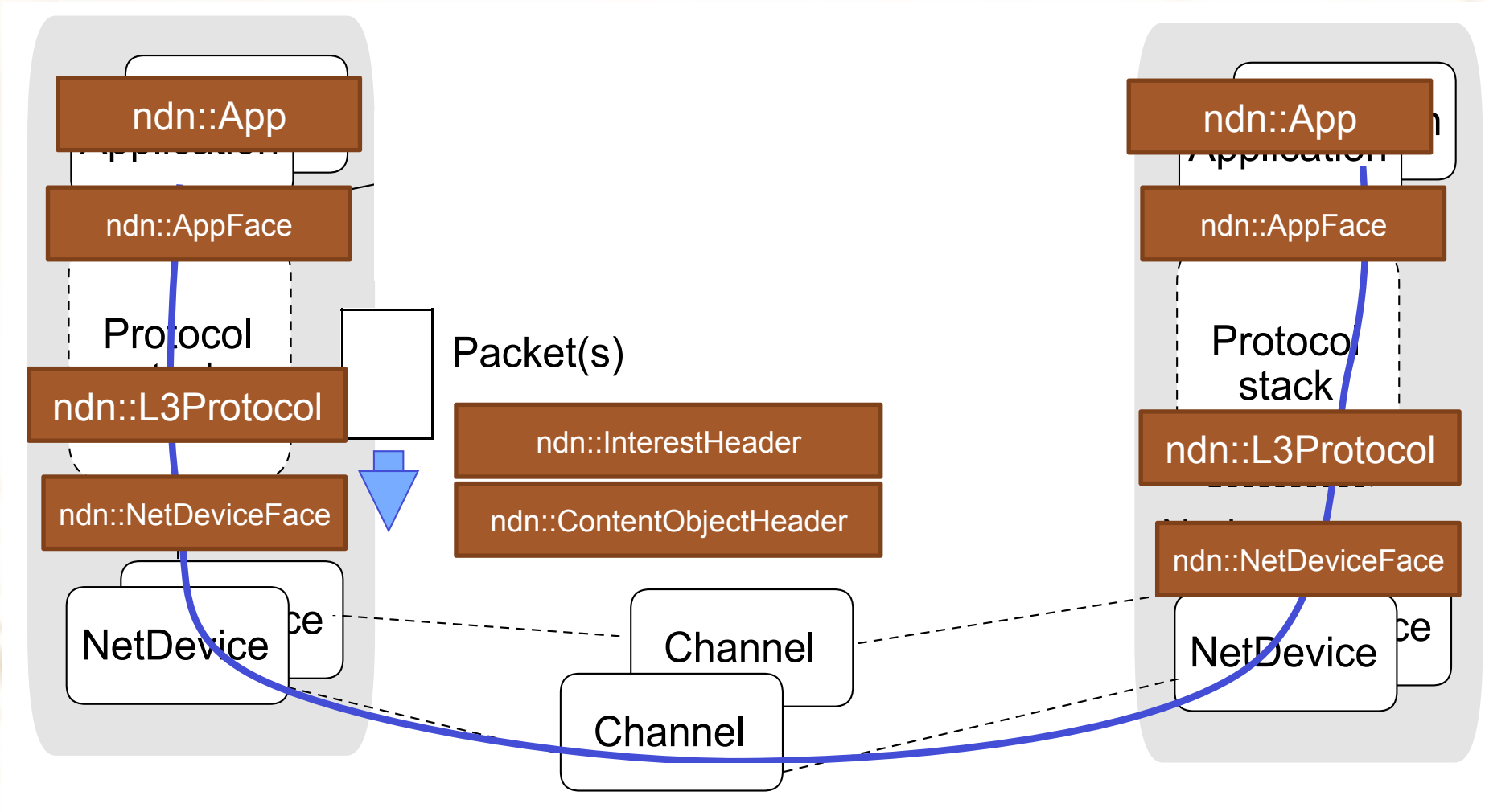
Other CCN Simulators

- ccnSim
  - primarily focused on cache behavior research
  - smaller memory footprint
    - more abstractions and simplifications
    - simplified Interest/Data packet formats (e.g., names restricted to number vectors?)
  - Not very modular for easy extension
- CCNPL-Sim
  - based on custom discrete event simulator (SSim)
  - limited flexibility for extensions
    - needs a content routing scheme as inter-layer between SSim and CCNPL-Sim?
    - How to use this for forwarding strategy experimentation?
- NS-3 Direct Code Execution + ccnd
  - most realistic evaluation of the prototype implementation
  - high per-node overhead
  - Difficult to experiment with different design choices
    - need to be implemented in real code first

Basic network simulation model in NS-3



ndnSIM extension of network simulation model

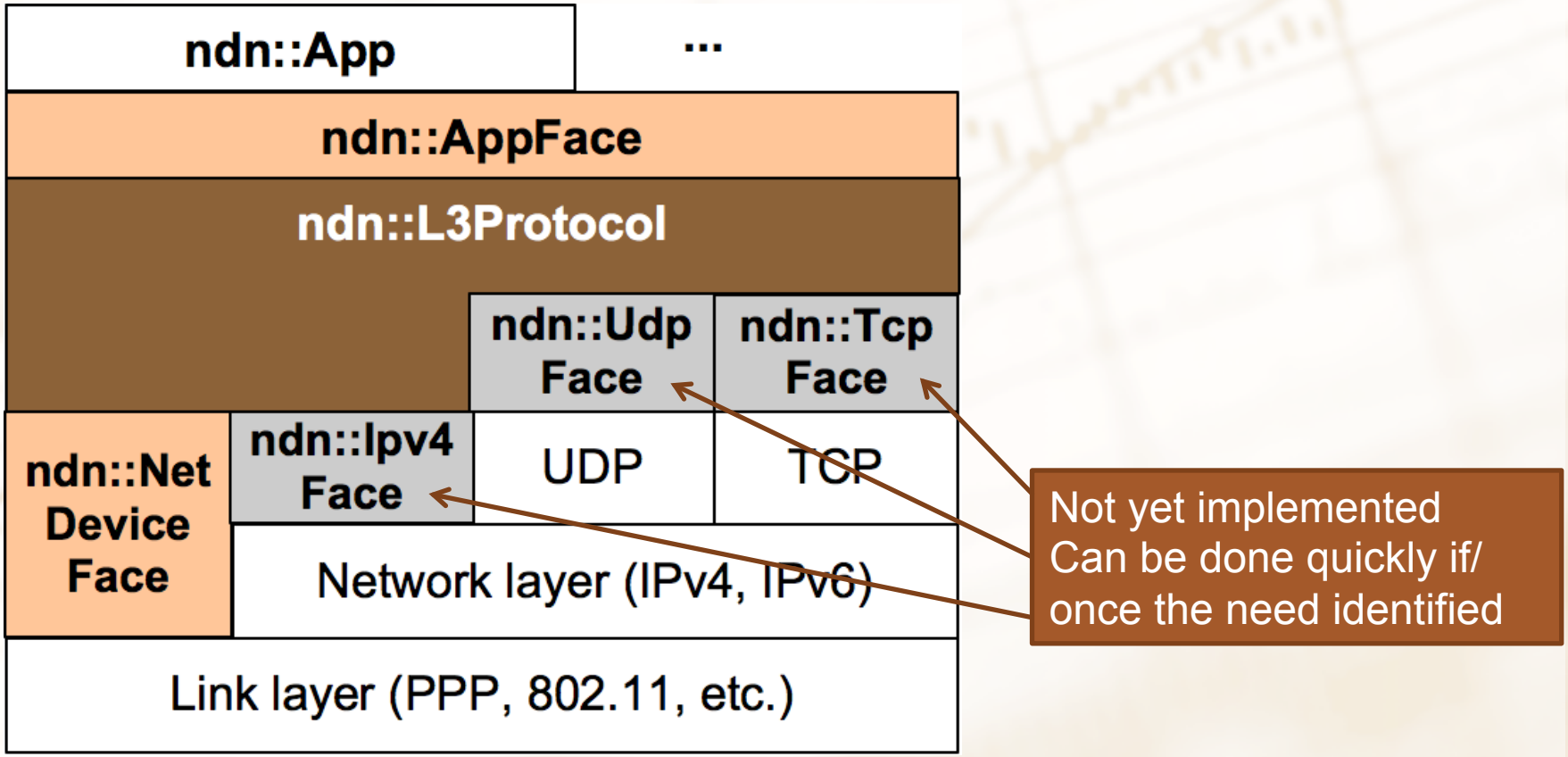


Core NDN protocol (ndn::L3Protocol)

- aggregates and manages all communication channels (Faces)
  - adding faces and registers necessary callbacks
  - removing faces
- receives packets from Faces and direct them to a scenario-selected forwarding strategy

Faces (ndn::Face)

- Abstraction from underlying protocols
  - callback registration-deregistration
  - packet encapsulation



Content Store

- In-network cache abstraction
  - add item
  - lookup item
- Currently available implementations of replacement policies
  - (default) Least-recently used (`ns3::ndn::cs::LRU`)
  - First-in-first-out (`ns3::ndn::cs::FIFO`)
  - Random (`ns3::ndn::cs::Random`)
- A desired content store module is selected and configured in simulation scenario

```
ndn::StackHelper ndnHelper;
ndnHelper.SetContentStore (“ns3::ndn::cs::LRU”, “MaxSize”, “100”);
```

Pending Interest Table (PIT)

- Abstraction to maintain state for each forwarded Interest packet
  - Create, Lookup, Erase entry
- Each PIT entry stores
  - Interest packet itself
  - list of incoming faces + associated info
  - list of outgoing faces + associated info
  - forwarding strategy tags
    - e.g., reference to a delayed processing queue
- Size of PIT can be limited in simulation scenario
  - Available policies for new PIT entry creation:
    - (default) persistent (`ns3::ndn::pit::Persistent`): a new entry will not be created if limit is reached
    - LRU (`ns3::ndn::pit::LRU`): when limit is reached, insertion of a new entry will evict the oldest entry
    - Random (`ns3::ndn::pit::Random`): when limit is reached, insertion will evict a random entry

Forwarding Information Base (FIB)

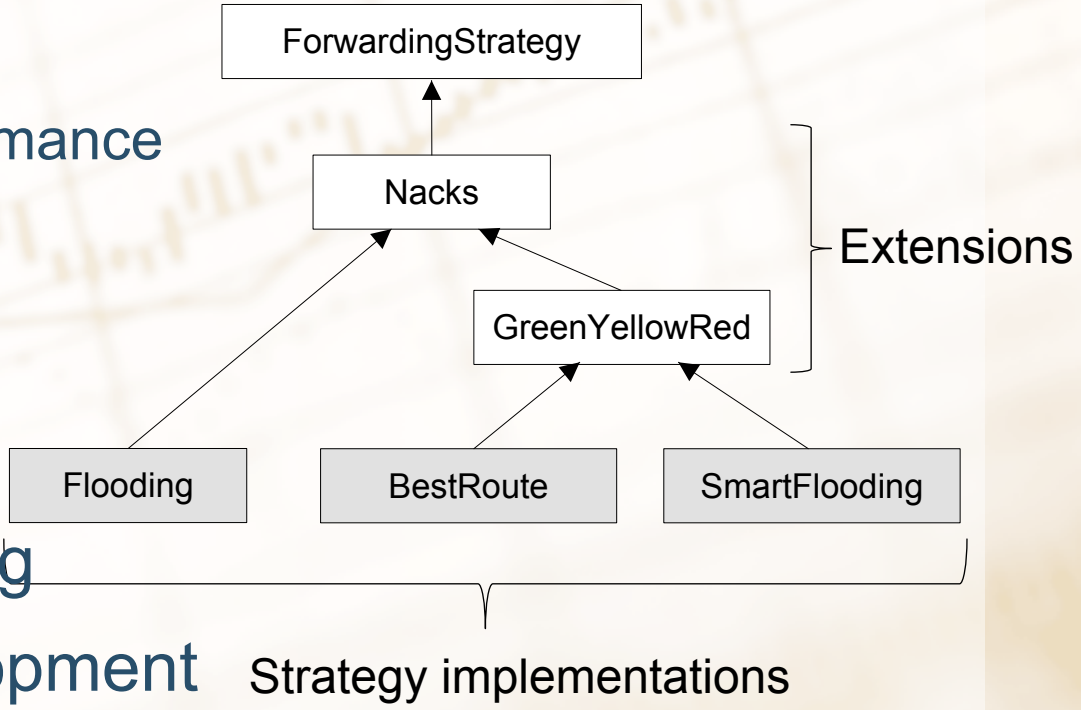
- Abstraction to store information about name prefixes
  - Add, Remove, LongestPrefixMatch
- Every FIB entry stores
  - prefix
  - list of (ranked) Faces
  - forwarding strategy tags
    - per-prefix limits, data-plane stats, etc.
- FIB, PIT, and Content Store implemented as a trie-like structure
  - every name component is a node in a tree
  - node's children organized in a hash map
  - leafs contain pointers to FIB/PIT/CS entries

FIB population

- Manually
- Default route
  - all interfaces added to default route
  - forwarding strategy make a choice
- Global routing controller
  - calculate SPF
  - install a best-route for prefix
- May add support for quagga-based population
  - rely on Direct Code Execution NS-3 module
  - use real routing protocol implementations (e.g.NDN prefixes distribution by OSPFN)

Forwarding strategies

- Abstraction for Interest and Data processing
  - OnInterest, OnData, WillErasePendingInterest, RemoveFace, FailedToCreatePitEntry, DidCreatePitEntry, WillSatisfyPendingInterest, and many other overrideable events
- Extensions
  - NACKs
  - Data plane status performance
- Available strategies
  - Flooding strategy
  - Smart flooding strategy
  - Best-Route strategy
- Several other forwarding strategies under development right now



Interest NACK

- Solves dangling state problem
  - when router cannot satisfy nor forward, it sends Interest NACK
  - removes PIT entry
- Signals downstream to action
  - explore other paths to find destination
  - avoid congested paths
- Details
  - NACK code added to Interest
  - Interest NACK carries the same nonce
    - basic protection against spoofing
  - NACK codes
    - **Duplicate**
    - No data/no prefix
    - **Congestion**
    - ...

Limits on number of pending Interests

- Limit based on bandwidth-delay product
  - Assuming a know size (average) of interest packets
  - # interests = capacity / (AvgDataSize + AvgInterestSize)
- Different granularities
  - per face (incoming/outgoing), per prefix (FIB/PIT)
- Pending Interest removed by
  - received Data
  - timeout
  - (optionally) NACK
- Features
  - prevents congestion
  - provides base for DDoS protection mechanisms
  - may result in link underutilization