NDN, Technical Report NDN-0005, 2012. http://named-data.net/techreports.html
Revision 2: October 5, 2012

ndnSIM: NDN simulator for NS-3

Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang

Abstract—Named Data Networking (NDN) is a newly proposed Internet architecture. NDN retains the Internet's hourglass architecture but evolves the thin waist. Instead of pushing data to specific locations, NDN retrieves data by name. On one hand, this simple change allows NDN networks to use almost all of the Internet's well tested engineering properties to solve not only IP's communication problems but also digital distribution and control problems. On the other hand, a distribution architecture differs in fundamental ways from a point-to-point communication architecture of today's Internet and raises many new research challenges. Simulation can serve as a flexible tool to examine and evaluate various aspects of this new architecture. To provide the research community at large with a common simulation platform, we have developed an open source NS-3 based simulator, ndnSIM, which faithfully implemented the basic components of a NDN network in a modular way. This paper provides an overview of ndnSIM.

1 Introduction

The fundamental changes introduced by the Named Data Networking (NDN) [1] to the Internet communication paradigm call for extensive and multidimensional evaluations of various aspects of the NDN design. While the existing implementation of NDN (CCNx Project [2]), along with the testbed deployment [3], give invaluable opportunities to evaluate both the NDN infrastructure design as well as its applications in a real-world environment, it is both difficult to experiment with different design options and impossible to evaluate the design choices in large scale deployment. To meet such needs and provide the community at large with a common experiment platform, we have developed an open source NDN simulator, ndnSIM, based on NS-3 network simulator framework [4].

The design of ndnSIM has the following goals in mind:

- Being an open source package to enable the research community to run experimentations on a common simulation platform.
- Being able to faithfully simulate all the basic NDN protocol operations.
- Maintaining packet-level interoperability with CCNx implementation [2], to allow sharing of traffic measurement and packet analysis tools between CCNx and ndnSIM, as well as direct use of real CCNx traffic traces to drive ndnSIM simulation experiments.
- Being able to to support large-scale simulation experiments.
- Facilitating network-layer experimentations with routing, data caching, packet forwarding, and congestion management.

Following the NDN architecture, ndnSIM is implemented as a new network-layer protocol model, which can run on top of any available link-layer protocol model (point-to-point, CSMA, wireless, etc.), as well as on top of network-layer (IPv4, IPv6) and transport-layer (TCP, UDP) protocols. This flexibility allows ndnSIM to simulate scenarios of various homogeneous and heterogeneous deployment scenarios (e.g., NDN-only, NDN-over-IP, etc.).

The simulator is implemented in a modular fashion, using separate C++ (set of) classes to model behavior of each network-layer entity in NDN: pending Interest table (PIT), forwarding information base (FIB), content store, network and application interfaces, Interest forwarding strategies, etc. This modular structure allows any component to be easily modified or replaced with no or minimal impact on other components. In addition, the simulator provides an extensive collection of interfaces and helpers to perform detailed tracing behavior of every component and NDN traffic flow.

We started the ndnSIM implementation effort in the fall of 2011. Since then the initial implementation has been used both by ourselves for various NDN design and evaluation tasks, as well as by a few external alpha testers. While we continue active development of ndnSIM, the first release of ndnSIM as an open-source package has been made available since June 2012. We hope that ndnSIM can provide a useful tool to the broader community who are interested in NDN research, and that the community can provide back to us both invaluable bug reports and new feature development.[1] More detailed information about the release, code download, basic examples, and additional documentation is available on ndnSIM website http://ndnsim.net/.

2 Design

The desire to create an open source NDN simulation package largely dictated our selection of the NS-3 network simulator [4] as the base framework for ndnSIM. Although NS-3 is relatively new and still does not have

1. Bug reports and feature requests can be filed through GitHub social coding website interfaces https://github.com/NDN-Routing/ndnSIM.

everything that commercial Qualnet or legacy ns-2 simulator has (e.g., NS-3 does not have native support for simulating conventional dynamic IP routing protocols[2]), it offers a clean and consistent design, extensive documentation, and implementation flexibility.

In this section we provide insights about main components of ndnSIM design, including description of protocol implementation components.

2.1 Design overview

The design of ndnSIM follows the philosophy of network simulations in NS-3, which devises maximum abstraction for all modeled components. Similarly to the existing IPv4 and IPv6 stacks, we designed ndnSIM as an independent protocol stack that can be installed on a simulated network node. In addition to the core protocol stack, ndnSIM includes a number of basic traffic generator applications and helper classes to simplify creation of simulation scenarios (e.g., helper to installing the NDN stack and applications on nodes) and tools to gather simulation statistics for measurement purposes.

The following list summarizes the component-level abstractions that have been implemented in ndnSIM; Figure 1 visualizes the basic interactions between them:

- **ndn::L3Protocol**: implementation of the core NDN protocol interactions: receiving Interest and Data packets from upper and lower layers through Faces;
- **ndn::Face**: abstraction to enable uniform communications with applications (ndn::AppFace) and other simulated nodes (ndn::NetDeviceFace) with pluggable (and optional) support of link-level congestion mitigation modules;
- **ndn::ContentStore**: abstraction for in-network storage (e.g., short-term transient, long-term transient, long-term permanent) for Data packets;
- **ndn::Pit**: abstraction for the pending Interest table (PIT) that keeps track (per-prefix) of Faces on which Interests were received, Faces to which Interests were forwarded, as well as previously seen Interest nonces;
- **ndn::Fib**: abstraction for the forwarding information base (FIB), which can be used to guide Interest forwarding by the forwarding strategy;
- **ndn::ForwardingStrategy**: abstraction and core implementation for Interest and Data forwarding. Each step of the forwarding process in ndn::ForwardingStrategy—including lookups to ContentStore, PIT, FIB, and forwarding Data packets according to PIT entries—is represented as virtual function calls, which can be overridden in particular forwarding strategy implementation classes (see Section 2.7);
- reference NDN applications, including simple traffic generators and sinks.

2. http://www.nsnam.org/docs/release/3.14/models/html/routing-overview.html

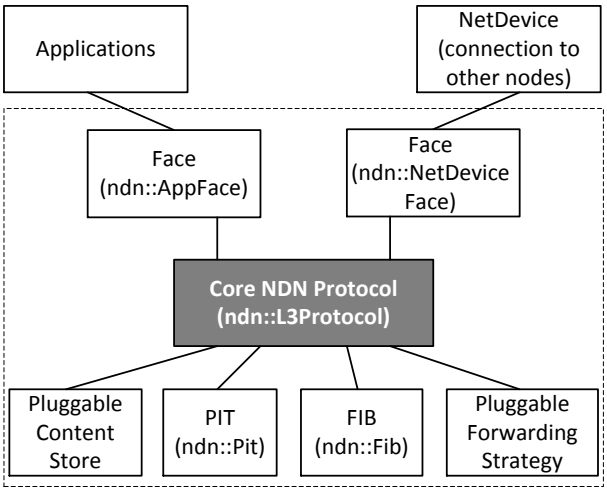

Fig. 1. Block diagram of ndnSIM components

Each component with the exception of the core ndn::L3Protocol has a number of alternative implementation that can be arbitrarily chosen by the simulation scenario using helper classes (see ndnSIM online documentation http://ndnsim.net/helpers.html). For example, ndnSIM currently provides implementations for ContentStore abstraction with Least-Recently-Used (LRU), First-In-First-Out (FIFO), and random replacement policies for cached Data.

The current wire format of Interest and Data packets follows the format of the existing CCNx Project's NDN implementation (CCNx Binary XML Encoding[3]). While this design choice imposes additional overhead (i.e., binary XML decoding and encoding processes have their costs), it allows reuse of the existing traffic analysis tools (ndndump,[4] wireshark ccn plugin[5]), as well as drive simulations using real CCNx traffic traces. Later, we will provide an option to disable packet format compatibility, which will allow conserving resources while running large-scale simulations.

Design of ndnSIM contains a number of optional modules, including (1) a place-holder for data security (the current code allows attaching of a user-specified "signature" to Data packets), (2) an experimental support of negative acknowledgments (Interest NACK) to provide fast feedback about data plane problem, (3) a pluggable Interest rate limit and interface availability component, and (4) extensible statistics module. Interested readers may see [5] for more details on Interest NACKs and Interest rate limit.

2.2 Core NDN protocol implementation

ndn::L3Protocol in ndnSIM is a central architectural entity and stands in the same line of class hierarchy as

3. http://www.ccnx.org/releases/latest/doc/technical/BinaryEncoding.html
4. https://github.com/cawka/ndndump/
5. https://github.com/ProjectCCNx/ccnx/tree/master/apps/wireshark

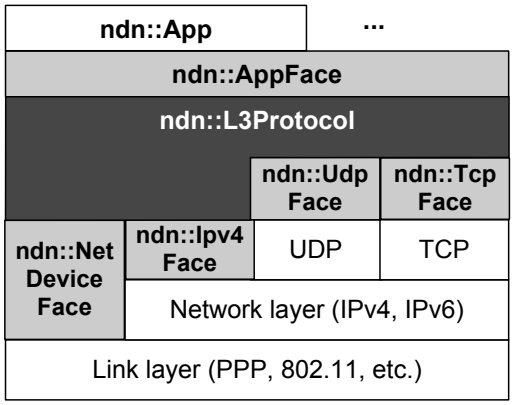

Fig. 2. Communication-layer abstraction for ndnSIM scenarios

the corresponding Ipv4L3Protocol and Ipv6L3Protocol classes of the NS-3 framework that implement IPv4 and IPv6 network-layer protocols. ndn::L3Protocol is a logical component aggregator for all available communication channels with both applications and other nodes (Face abstraction, see Section 2.3) and performs basic handling of incoming packets from Faces to a forwarding strategy.

ndn::L3Protocol class defines the API to manipulate the following aspects of the NDN stack implementation:

- **AddFace/RemoveFace**: to register a new Face realization to NDN protocol or remove an existing Face;

2.3 Face abstraction

To achieve our goal of providing maximum flexibility and extensibility, we make the ndnSIM design independent from the underlying transports through abstracting the inter-layer interactions. That is, all communication between the core protocol implementation (ndn::L3Protocol), network, and applications is accomplished through a *Face* abstraction (ndn::Face), which can be realized in various forms (see Figure 2): link-layer face (ndn::NetDeviceFace) for inter-node communication directly over link layer, network-layer face (ndn::Ipv4Face and ndn::Ipv6Face) and transport-layer face (ndn::TcpFace and ndn::UdpFace) for inter-node overlay communication, and application-layer face (ndn::AppFace) for intra-node communications.

At the first release, the current ndnSIM package provides only a link-layer ndn::NetDeviceFace and an application-layer ndn::AppFace. With these two faces one can simulate a fully NDN-enabled network. It is straightforward to add other types of faces and we expect them to be added as soon as the needs for simulating more complex scenarios (i.e., where NDN nodes are intermixed with nodes that do not run NDN protocol) arise, either by ourselves or by others from the community.

The Face abstraction defines the following API:

- **SendImpl** (realization-specific): to pass packets from NDN stacks to the underlying layer (network or application).
- **RegisterProtocolHandler** (realization-specific): to enable forwarding packets from the underlying layer (network or app) to the NDN stack.
- **SetMetric/GetMetric**: to assign and get Face metrics that can be used, for example, in routing calculations.
- **IsUp/SetUp**: to check if Face is enabled and to enable/disable Face.

In addition to the base API, the Face abstraction provides methods to store arbitrary information, which can be used by the forwarding strategy module. For example, an ndn::fw::SimpleLimits Interest forwarding strategy implements limits on number of outstanding Interest packets based on physical link limits [5].

2.4 Content Store abstraction

Content Store at each NDN router enables in-network storage, providing efficient error recovery and asynchronous multicast data delivery. ndnSIM provides an interface to plug in different implementation of Content Store that can implement different indexing and item look up designs, different size limiting features, as well as different cache replacement policies.

The current version of ndnSIM contains three realizations of the Content Store abstraction with Least-Recently-Used (**ndn::cs::Lru**), First-In-First-Out (**ndn::cs::Fifo**), and Random replacement policies (**ndn::cs::Random**). Each of these realization is based on a dynamic trie-based container with an (optional) upper bound on its size, and hash-based indexing on Data names (per-component lookup on a trie). Other Content Store modules are expected to be implemented either by ourselves or with the help of the community as the need arises.

The Content Store abstraction provides the following operations:

- **Add** (realization specific): caching a new or promoting existing Data packet in cache.
- **Lookup** (realization specific): performing lookup for a previously cached Data.

The Content Store abstraction does not provide an explicit data removal operation. Lifetime of the Content Store entries depends on traffic pattern, as well as on Data packet freshness parameter supplied by data producers.

2.5 Pending Interest table (PIT) abstraction

PIT (ndn::Pit) maintains state for each forwarded Interest packet in order to provide directions for Data packet forwarding. Each PIT entry contains the following information:

- the name associated with the entry;

- a list of incoming Faces, from which the Interest packets for that name have been received, with associated information (e.g., arrival time of the Interests on this Face);
- a list of outgoing Faces to which the Interest packets have been forwarded with associated information (e.g., time when the Interest was sent on this Face, number of retransmission of the Interests on this face, etc.);
- time when the entry should expire (the maximum lifetime among all the received Interests for the same name), and
- any other forwarding-strategy-specific information in form of forwarding strategy tags (any class derived from ndn::fw::Tag).

The current version of ndnSIM provides a templated realizations of PIT abstraction, allowing optional bounding the number of PIT entries and different replacement policies, including

- persistent (**ndn::pit::Persistent**)—new entries will be rejected if PIT size reached its limit;
- random (**ndn::pit::Random**)—when PIT reaches its limit, random entry (could be the newly created one) will be removed from PIT;
- least-recently-used (**ndn::pit::Lru**)—the least recently used entry (the oldest entry with minimum number of incoming faces) will be removed when PIT size reached its limit.

All current PIT realizations are organized in a trie-based data structure with hash-based indexing on Data names (per-component lookup on a trie) and additional time index (by expiration time) that optimizes removal of timed out Interests from the PIT.

A new PIT entry is created for every Interest with a unique name. When an Interest with a name that has been seen previously is received, the "incoming Faces" list of the existing PIT entry is updated accordingly, effectively aggregating (suppressing) similar Interests.

The PIT abstraction provides the following realization-specific operations:

- **Lookup**: find a corresponding PIT entry for the given content name of an Interest or Data packet;
- **Create**: create a new PIT entry for the given interest;
- **MarkErased**: Remove or mark PIT entry for removal;
- **GetSize**, **Begin**, **End**, **Next**: get number of entries in PIT and iterate through entries.

2.6 Forwarding information base (FIB)

An NDN router's FIB is roughly similar to the FIB in an IP router except that it contains name prefixes instead of IP address prefixes, and it (generally) shows multiple interfaces for each name prefix. It is used by the forwarding strategies to make Interest forwarding decisions.

The current realization of FIB (ndn::fib:FibImpl) is organized in a trie-based data structure with hash-based indexing on Data names (per-component lookup on a trie), where each entry contains a prefix and an ordered list of (outgoing) Faces, through which the prefix is reachable. The order of Faces is defined as a composite index, combining the routing metric for the Face and data plane feedback. Lookup for a match is performed on variable-length prefixes in a longest-prefix match fashion.[6]

2.6.1 FIB population

Currently, ndnSIM provides several methods to populate entries in FIB. First, one can use a simulation script to manually configure FIBs for every node in a simulation setting. This method gives the user full control on what entries are present in which FIB, and can work well for small-scale simulations. However, it may become infeasible for simulations with large topologies.

The second method is to use a central global NDN routing controller to automatically populate all routers' FIBs. When requested (either before a simulation run starts, or at any time during the simulation), the global routing controller obtains information about all the existing nodes with NDN stack installed and all exported prefixes, and uses this information to calculate shortest paths between every node pair and updates all the FIBs. Boost.Graph library (http://www.boost.org/doc/libs/release/libs/graph/) is used in this calculation.

In the current version, the global routing controller uses the Dijkstra's shortest path algorithm (using Face metric) and installs only a single outgoing interface for each name prefix. To experiment with multipath Interest forwarding scenarios, the global routing controller needs to be extended to populate each prefix with multiple entries. However, it is up to the particular simulation to define the exact basis for multiple entries, e.g., whether entries should represent paths without common links. We welcome suggestions and/or global routing controller extensions, which can be submitted on GitHub website (https://github.com/NDN-Routing/ndnSIM).

Finally, a simple method to populate FIB is to install default route (route to /), which includes all available Faces of NDN stack. For example, this method can be useful for simulations that explore how well Interest forwarding strategy can find and maintain paths to prefixes without any guidance from the routing plane.

2.7 Forwarding strategy abstraction

Our design enables experimentation with various types of forwarding strategies, without any need to modify the core components. This goal is achieved by introducing the forwarding strategy abstraction (ndn::ForwardingStrategy) that implements core handling of Interest and Data packets in an event-like

6. In addition to the longest-prefix match, next release of ndnSIM will implement several Interest selectors, including two types of exclude filters (ordered exclude and Bloom filters) and min/max name components filter [2].

fashion. In other words, every step of an Interest and Data packet handling, including Content Store, PIT, FIB lookups, is represented as a virtual function, which can be overridden in particular forwarding strategy implementation classes.

More specifically, the forwarding strategy abstraction provides the following set of overrideable actions:

- **OnInterest**: called by CcnxL3Protocol for every incoming Interest packet;
- **OnData**: called by CcnxL3Protocol for every incoming Data packet;
- **WillErasePendingInterest**: fired just before PIT entry is removed;
- **RemoveFace**: call to remove references to Face (if any used by forwarding strategy realization);
- **DidReceiveDuplicateInterest**: fired after reception of a duplicate Interest is detected;
- **DidExhaustForwardingOptions**: fired when forwarding strategy exhaust all forwarding options to forward an Interest;
- **FailedToCreatePitEntry**: fired when an attempt to create a PIT entry failed;
- **DidCreatePitEntry**: fired after an successful attempt to create a PIT entry;
- **DetectRetransmittedInterest**: fired after detection of a retransmitted Interest. This even is optional and will be fired only when "DetectRetransmissions" option is enabled in scenario. Detection of a retransmitted Interest is based on observation that a new Interest matching existing PIT entry arrives on a Face that is recorded in incoming list of the entry. Refer to the source code for more details.
- **WillSatisfyPendingInterest**: fired just before pending Interest is satisfied with Data;
- **SatisfyPendingInterest**: actual procedure to satisfy pending Interest;
- **DidSendOutData**: fired every time Data was successfully send out on a Face (can fail during congestion or if rate limiting module is enabled);
- **DidReceiveUnsolicitedData**: fired every time Data arrives, while there is no corresponding pending Interest for the Data's name. If "CacheUnsolicitedData" option is enabled, then such Data packets will be cached by the default processing implementation.
- **ShouldSuppressIncomingInterest**: hook to Interest suppression logic;
- **TrySendOutInterest**: fired before actually sending out Interest on a Face;
- **DidSendOutInterest**: fired after successfully sending out Interest on a Face;
- **PropagateInterest**: basic Interest propagation logic;
- **DoPropagateInterest** (realization-specific): realization-specific Interest propagation logic.

We anticipate that in the future releases more events will be added to the forwarding strategy abstraction. At the same time, additional events can be created in an objective-oriented fashion through class inheritance. For example, an Interest NACK extension (see [5] for more detail) is implemented as a partial specialization of the forwarding strategy abstraction.

Figure 3 shows partial hierarchy of the currently available forwarding strategy extensions (Nacks, GreenYellowRed) and full forwarding strategy implementations (Flooding, SmartFlooding, and BestRoute) that can be used in simulation scenarios. While all current realizations inherited from Nack extension that implements additional processing and events to detect and handle Interest NACKs [5], NACK processing is disabled by default and can be enabled using "EnableNACKs" option.

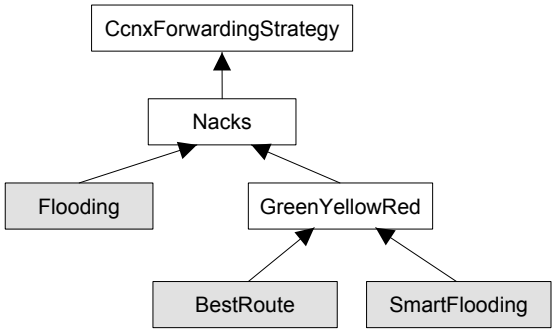

Fig. 3. Available forwarding strategies (Flooding, SmartFlooding, and BestRoute are full realizations)

SmartFlooding and BestRoute realizations rely on color-coding for the status of each Face, based on observed data plane feedback [5].

- **GREEN**: the Face works correctly (e.g., if an Interest is sent to this face, Data is returned);
- **YELLOW**: the status of Face is unknown (e.g. it may be added recently, or has not been used for a while);
- **RED**: the Face is not working and should not be used for Interest forwarding.

The status information is attached to each Face in FIB entry and initially initialized to YELLOW status. Every time a Data packet is returned as the response to a previous Interest, the corresponding Face in FIB entry is set to GREEN color. Every time an error occurs (PIT entry timeout or, if enabled, NACK-Interest is received), the Face is turned back to YELLOW status. If a Face has not been used for long enough, it is turned back to YELLOW status. RED state is assigned to the Face when the lower layer notifies NDN stack of a problem (link failure, connection error, etc.).

The following list summarized the processing logic in the currently available full forwarding strategy implementations:

- **Flooding strategy** (ndn::fw::Flooding): an Interest packet is forwarded to all Faces that available in FIB entry for the Interest's prefix, except the incoming face of that Interest.
- **Smart flooding strategy** (ndn::fw:SmartFlooding): if FIB entry contains at least one GREEN Face, Interest is forwarded *only* to the highest-rank GREEN Face. Otherwise, all YELLOW Faces will be used to

forward the Interest. RED Faces are not used for Interest forwarding.
This strategy mode can be used in simulations without routing input, and the data plane can use Interest packets to discover and maintain working paths.

- **Best-Route strategy** (ndn::fw::BestRoute) forwards Interest packets to the **highest-ranked** GREEN (if available) or YELLOW face. RED Faces are not used for Interest forwarding.

There is also an experimental SimpleLimits forwarding strategy (inheriting most actions from ndn::fw::BestRoute) that attempts to avoid congestion in the network and maximize network utilization by taking into account the Interest rate limits in Face selection process. For example, if the highest-ranked Face has reached its capacity—more specifically, number of pending Interests for this Face reached a set maximum limit—the strategy will select next Face in the rank order that is under the limit.

In addition to the strategies that have been implemented, we are also working on implementing additional Interest forwarding strategies, including the strategy that precisely model the behavior of the Project CCNx implementation.

2.8 Reference applications

Applications interact with the core of the system using ndn::AppFace realization of Face abstraction. To simplify implementation of specific NDN application, ndnSIM provides a base ndn::App class that takes care of creating ndn::AppFace and registering it inside the NDN protocol stack, as well as provides default processing for incoming Interest and Data packets.

Listed below is the set of reference applications that is currently available in ndnSIM:

- **ndn::ConsumerCbr** an application that generates Interest traffic with predefined frequency (constant rate, constant average rate with inter-Interest gap distributed uniformly at random, exponentially at random, etc.). Names of generated Interests contain a configurable prefix and a sequence number. When a particular Interset is not satisfied within an RTT-based timeout period (same as TCP RTO), this Interest is re-expressed.
- **ndn::ConsumerBatches**: an on-off-style application generating a specified number of Interests at specified points of simulation. Names and retransmission logic is similar to ndn::ConsumerCbr application.
- **ndn::Producer** a simple Interest-sink application, which replying every incoming Interest with Data packet with a specified size and name same as in Interest.

3 Related Work

Over the last few years several efforts have been devoted to the development of evaluation infrastructures for NDN architecture research.

One existing effort by the NDN project team is the support of NDN on the Open Network Lab (ONL) [6]. ONL currently contains 14 programmable routers, over 100 client nodes, connected by links and switches of various capability. Every node and router runs Project CCNx NDN implementation. Users have full access to the hardware and software state of any node of ONL. It is also possible to run and evaluate Project CCNx NDN implementation on nodes of DeterLab testbed [7]. Having a programmable non-virtualized testbed is a very valuable option, though its capability is limited to evaluate relatively small size networks. For larger-scale experiments, researchers may need to resort to simulations.

Rossi and Rossini [8] developed ccnSim to evaluate caching performance of NDN. ccnSim is a scalable chunk-level simulator of NDN that is written in C++ under the Omnet++ framework, which allows assessing NDN performance in large-scale scenarios (up to 10^6 chunks) on a standard consumer-grade computer hardware. ccnSim was designed and implemented with the main goal of running experimentations of different cache replacement policies for NDN routers content store. Therefore, it is not a fully featured implementation of the existing NDN protocol. In the current version of ccnSim, PIT and FIB components are implemented in the simplest possible way, thus it is unable to evaluate different data forwarding strategies, different routing policies, or different congestion control schems.

Another NDN simulator was written at Orange Labs by Muscariello and Gallo [9]. Their Content Centric Networking Packet Level Simulator (CCNPL-Sim) is based on SSim that is a utility library which implements a simple discrete-event simulator. Combined Broadcast and Content-Based routing scheme (CBCB) [10] must run as an interlayer between SSim and CCNPL-Sim to enable name-based routing and forwarding over generic point-to-point networks. Though a canonical NDN model was completely reimplemented in CCNPL-Sim in C++, this solution has a drawback of using a custom discrete-event simulator that is unfamiliar to most of the researchers. Additionally, an obligatory usage of CBCB narrows the possible experimentation area, making it impossible to evaluate other routing protocols, such as OSPF-N (OSPF extension for NDN) or routing on Hyperbolic Metric Space [11].

A completely different approach was taken by Urbani et al. [12]. They provided support of Direct Code Execution (DCE) for Project CCNx NDN implementation inside the NS-3 simulator. The general goal of DCE NS-3 module is to provide facilities to execute existing implementations of user-space and kernel-space network protocols within NS-3 simulated environment. The main advantage of this approach is that simulations can use the existing unmodified CCNx code directly, thus providing maximum realism and requiring no code maintenance (as new versions are supposed to run in DCE NS-3 without much effort). However, this approach

also raises a few concerns. First, the real implementation, including Project CCNx code, is rather complex, difficult to modify to explore different design approaches, and contains a lot of code that is irrelevant for simulation evaluations. Second, there is a known scaling problem, because each simulated node has to run a heavy DCE layer and a full-sized real CCNx implementation.

4 Summary

ndnSIM is designed as a set of loosely bound components that give a researcher an opportunity to modify or replace any component, with no or little impact on the other parts of ndnSIM. Our simulator provides a set of reference application and helper classes, allowing evaluation of various aspects of NDN protocol under many different scenarios. The first version of ndnSIM was publicly released in June 2012 and more detailed information about the release and additional documentation is available on ndnSIM website http://irl.cs.ucla.edu/ndnSIM/.

We hope that the community find ndnSIM useful, and we look forward to the community's feedback to help further improve it.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of ACM CoNEXT*, 2009.

[2] (2012, May) Project CCNx. [Online]. Available: http://www.ccnx.org

[3] (2012, February) The NDN project testbed. [Online]. Available: http://www.named-data.net/testbed.html

[4] (2012, May) ns-3. [Online]. Available: http://www.nsnam.org/

[5] Y. Cheng, A. Afanasyev, I. Moiseenko, B. Zhang, L. Wang, and L. Zhang, "Smart forwarding: A case for stateful data plane," Tech. Rep. NDN-0002, May 2012.

[6] L. Zhang et al., "Named data networking (NDN) project 2010 - 2011 progress summary," PARC, http://www.named-data.net/ndn-ar2011.html, Tech. Rep., November 2011.

[7] "DETER network security testbed," https://www.isi.deterlab.net.

[8] D. Rossi, G. Rossini, "Caching performance of content centric networksunder multi-path routing (and more)," Telecom ParisTech, Tech. Rep., 2011.

[9] L. Muscariello. (2011) Content centric networking packet level simulator. Orange Labs. [Online]. Available: http://perso.rd.francetelecom.fr/muscariello/sim.html

[10] A. Carzaniga, M.J. Rutherford, and A.L. Wolf, Ed., *A Routing Scheme for Content-Based Networking*. IEEE INFOCOM, March 2004.

[11] CAIDA. Caida's role in the ndn. [Online]. Available: http://www.caida.org/projects/ndn-fia/

[12] F. Urbani, W. Dabbous, and A. Legout. (2011, November) NS3 DCE CCNx quick start. INRIA. [Online]. Available: http://www-sop.inria.fr/members/Frederic.Urbani/ns3dceccnx/index.html