POLLERE, INC.
network analysis & architecture, performance

Lessons Learned Building a Secure Network Measurement Framework using Basic NDN

Kathleen Nichols

ACM ICN 2019
September 26, 2019

©Pollere, Inc.

Talk in a Nutshell

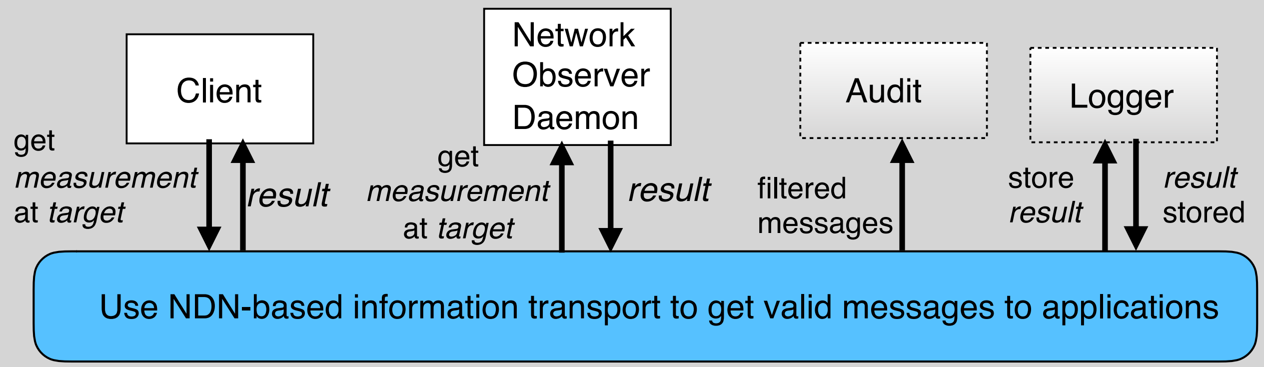
- This is the opposite of “ICN Protocol Enhancements”:
  - the basic NDN protocol provides a complete network layer
  - enhancements belong *above* that layer
- While developing NDN Distributed Network Measurement Protocol (DNMP), observed that:
  - application is well-suited to NDN features: information-centric with fine-grained role-based security
  - implementing with NDN is more difficult than it could be
  - significant bugs in NFD distribution, particularly in multicast handling
  - NDN lacks library of useful communications models
  - it’s difficult to integrate application trust model with NDN codebase
- To address the issues, DNMP was co-developed with a few new tools and some NFD bug patches

Modules that work *within* existing architecture, not new layers or concepts that create a straight-jacket for applications

DNMP is an infrastructure or system application on top of NDN with user-facing applications of its own

This slide is the very short form of the talk. The long form starts with the basic structure of DNMP

DNMP structure: applications and interactions



- Client: solicits measurements, role-based identity, ephemeral
- NOD: collects measurements, device-based identity, persistent
- Audit: captures and saves messages in case later audit is needed (check "who did what to which")
- Logger: provides storage for asynchronous measurements

Only Clients and NODs are required for a working implementation

DNMP is an NDN application; there are currently four types of DNMP applications. Two derive from some earlier NIST work in this area, are necessary, and have been implemented.

The other two are to make the overall measurement architecture more useful and are still in early stages.

Next step: consider the communication model needed

DNMP communication model

- Client measurement requests can be one-to-many
- Replies from NODs can be many-to-one
- ‘Audit’ adds additional listener(s) → many-to-many

Suggests a publish-subscribe communications model

- publishers have no knowledge of subscribers and vice-versa
- topic-based (more general than producer-based)
- applications generally both publish and subscribe
  - clients publish to **command** topic to request measurements and subscribe to associated **reply** topic
  - NODs subscribe to **command** and publish to **reply** topics

Audits could subscribe to command and/or reply, but most likely to follow command for any later forensic audit and (possibly) filter on certain subtopics

DNMP applications view this communications model through an API to the modules that implement it

DNMP application view of communication model

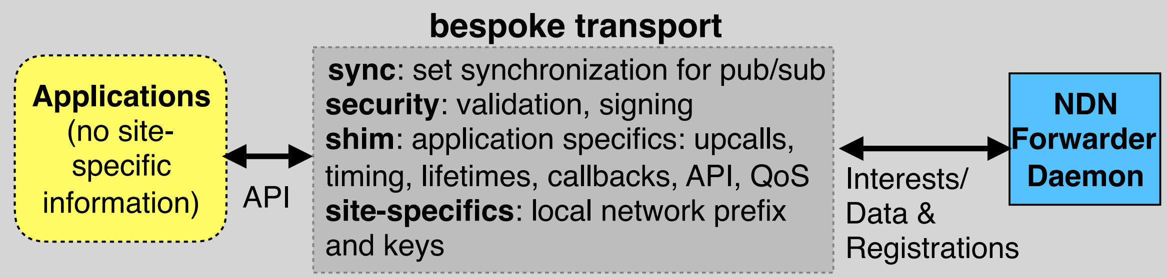
- Client calls pass *probe*, *arguments*, *target*
  - probe: the measurement type (e.g. NFD General Status)
  - arguments: makes more specific (e.g. count of Interests)
  - target: location of measurement (e.g. **local**, **all**, *ID*)
- Clients callbacks pass *result*
- NOD callbacks pass specific measurement request, including *probe* and *arguments*
- NOD calls pass the specific measurement request (to identify result) and its *result*

Match this model to existing codebase

- ???
- Shortage of libraries and examples

Further, modularization of applications and information transport generally lacking though there are some more recent attempts to address parts of this, we found them to be more akin to a straightjacket for applications (and insufficiently information-centric and not integrated with security)

Decompose: communications models are implemented by a bespoke transport containing the sync and application-specific functions



security: run-time validation, signing, and publication construction using the DNMP trust schema

shim: object that provides application specificity using library or template functional modules including publication expiration, API, sending priority and delivery QoS (e.g. at most once, at least once)

=> wanted a simple pub/sub **sync**, but there wasn't one (there is now)

bespoke transport is just a name for the collection of modules that implement an NDN user-space information transport, not a new architectural feature

Divide and conquer: with a modular roadmap, start to fill it in

syncps: A Lightweight Basic Pub/Sub Sync

- Available syncs not topic-based nor fit to ephemeral messages
- syncps is MQTT-like but brokerless and broadcast-efficient
  - As in MQTT, application-specific enhancements like delivery QoS and storage are implemented *on top of* syncps
  - upcalls are used to obtain application-specific information (e.g. lifetimes, priority) and actions (e.g. publication validation, expiry)
  - syncps sends Interests giving the target, topic and an IBLT identifying its (unexpired) publications
  - receivers respond with new (not in IBLT) publications (NDN Data) packaged into a syncps Data ordered by priority
  - DNMP's trust schema applied to publications, **not** the outer Data
  - publications have a limited lifetime to bound state needed to prevent replay. Implemented via timestamps requiring approximate time sync (defaults to1 sec. jitter tolerance)
  - broadcast media (e.g. a wifi network) makes this efficient

Just because Data is unique does not mean that it has to live forever

Storage, like transport, has application specifics that mean making it *transparent* to applications is problematic.

syncps Data uses SHA256, initialized in the syncps constructor with the line:
m_signingInfo(ndn::security::SigningInfo::SIGNER_TYPE_SHA256),
Can be overridden by a shim via the syncps methods setSigningInfo() and setValidator() which set the NDN Interest/Data validation policy.

Specify DNMP Publications

- Request/response interaction of clients and NODs is akin to ephemeral RPC
- Implement this with two publication topics, **command** and **reply**
- Clients publish *cpub* in **command** topic and NODs publish *rpub* in **reply** topic

Name format, components grouped by function (**bold** indicates a literal):

cpub = <domain>/<target>/**command/**<role>/<pType>/<pArgs>/<origin>/<cTS>

rpub = <domain>/<target>/**reply/**<*cmdID*>/<nodId><rTS>

Notes:
- *domain* expands to <root>**/dnmp** where the *root* or *networkID* is site-specific
- *cmdID:* exact copy of **command**'s last five groups, i.e., **reply** takes Name of **command** that initiated it, replaces **command** with **reply** and appends its own two groups
- timestamps (*x*TS): UTC nanosecond timestamps give publication creation time
- *target* specifies where the directive is performed, e.g. all, local, or unique NOD id
- *pType:* measurement probe descriptive name, *pArgs* makes measurement more specific
- more detail in the paper, github code (slight changes from paper)

Note this complex name is only for DNMP publications, not necessary to expose to the DNMP applications.

DNMP Trust Rules for cpub and rpub

cpub

<domain>/<target>/**command/**<role>/<pType>/<pArgs>/<origin>/<cTS>

where: domain = <root>/**dnmp**

requires:

roleCert = <domain>/<role>/<keyinfo>

dnmpCert = <domain>/<keyinfo>

has signing chain ("<=" denotes "is signed by"):

cpub <= roleCert <= dnmpCert <= netCert

rpub

<cpub **command** => **reply**>/<nodId>/<rTS>

where: "=>" denotes "is replaced by"

requires:

nodCert = <domain>/**nod**/<nodID>/<keyinfo>

devCert = <root>/**device**/<devID>/<keyinfo>

configCert = <root>/**config**/<configID>/<keyinfo>

has signing chain:

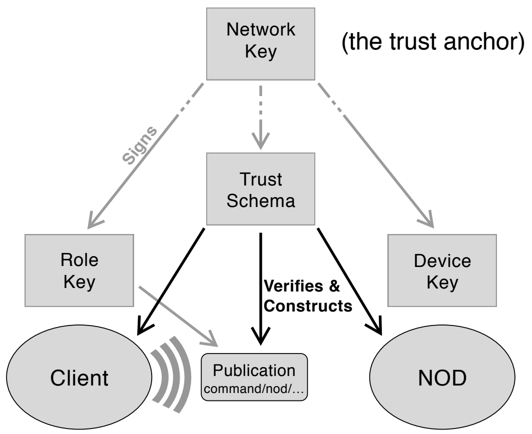
rpub <= nodCert <= deviceCert <= configCert <= netCert

Should mean the hard work is done, but…

- Library’s validator language doesn’t reflect the human specification
- NDN library doesn’t produce a signable trust schema, thus *can’t trust the trust schema*!
- At best, existing validator only checks *some* components, *some* Names and can’t check signing chain as a unit
- But trust rules *<u>define</u>* Names and signing relationships and *should* be usable to:
  - check soundness of the trust schema
  - construct packets and automatically choose signing keys
  - validate *entire* signing chain, syntax and authorizations

A New Approach: Versatile Security Toolkit (VerSec)

- A language and compiler for trust rules that checks entire schema, then outputs it in a *signable* compact binary form as key
- Run-time security object, *schemer*, for validation and Data name construction, also allows applications to reference Name components by names!



Example: VerSec compiler input for DNMP

```
# NDN trust schema to validate command and reply publications
# for DNMP v0.5.0

# root key (trust anchor) name
network = myhouse

# schema describes items in the DNMP namespace of this network
domain = <network>/dnmp

# how various entities are identified in the schema

user = user/<Id>               # an authorized user
operator = operator/<opId>     # an authorized operator
configurator = config/<confId> # an authorized device configurer
device = device/<devId>        # authorized, configured, device
nod = nod/<nodId>              # authorized nod on some device


# schema for legal commands and related definitions
#   operators can probe any target.
#   users can probe their local nod or ping any target.
# 'cmd_user' is listed first in the definition of 'cmd' so a
# 'user' signing key is preferred if the command allows it and
# both 'user' & 'operator' keys are available (i.e., force the
# "least privilege" choice).
target = nod/(local|all|<!nodId>) # possible command targets
probe = <!pType>/<!pArgs>         # required components of
                                  #   'probe' command

uprobes = Pinger/<!pArgs>         # probe(s) a user can issue
                                  #    to any nod

uCmd = nod/local/command/<Id>/<probe> |\
       <target>/command/<Id>/<uprobes>
oCmd = <target>/command/<opId>/<probe>
cmd = <domain>/(<uCmd> | <oCmd>)/<or>/<ts>

or = <!origin $sysId>             # command's origin
ts = <!cTS $tStamp>               # command's timestamp
```

```
# schema for nod replies to commands.
# The initial prefix of a reply is the name of the command that
# solicited it except the literal component 'command' is
# replaced with 'reply'.  The following line constructs legal
# reply prefixes by doing this substitution on all legal command
# definitions (cmd = ...) above.
# The result is marked "don't verify" since:
#  - the command was verified on arrival to the NOD
#  - command's originator knows the rule constructing reply
#    name and is subscribed to this exact result.
# Final two components of reply identify the replying NOD (this
# component *is* verified) and the time the reply was generated.
reply = <!cmd: command => reply>/<nodId>/<!rTS $tStamp>

# signing certificate name schemas & related definitions

role = <user> | <operator>      # roles that can sign commands
keyinfo = KEY/_/_/_             # standard NDN key name suffix
                                # (this schema ignores last 3
                                #   components)

netCert = <network>/<keyinfo>   # trust anchor
configCert = <network>/<configurator>/<keyinfo>
deviceCert = <network>/<device>/<keyinfo>
dnmpCert = <domain>/<keyinfo>
roleCert = <domain>/<role>/<keyinfo>
nodCert = <domain>/nod/<nodId>/<keyinfo>

# command Publication signing chain
cmd <= roleCert <= dnmpCert <= netCert

# reply Publication signing chain
reply <= nodCert <= deviceCert <= configCert <= netCert
```

- 26 lines of code, heavily commented, not unlike the rule specifications
- working proof-of-concept handles DNMP, existing NDN trust schema uses, other ICN applications

Note this implements trust rules where operators can do any local or remote (“all”) measurement but the only non-local measurement regular users can do is the Pinger.

```
cmd = {
  /myhouse/dnmp/nod/local/command/<Id>/<!pType>/<!pArgs>/<!origin $sysId>/<!cTS $tStamp>
  /myhouse/dnmp/nod/local/command/<Id>/Pinger/<!pArgs>/<!origin $sysId>/<!cTS $tStamp>
  /myhouse/dnmp/nod/all/command/<Id>/Pinger/<!pArgs>/<!origin $sysId>/<!cTS $tStamp>
  /myhouse/dnmp/nod/<!nodId>/command/<Id>/Pinger/<!pArgs>/<!origin $sysId>/<!cTS $tStamp>
  /myhouse/dnmp/nod/local/command/<opId>/<!pType>/<!pArgs>/<!origin $sysId>/<!cTS $tStamp>
  /myhouse/dnmp/nod/all/command/<opId>/<!pType>/<!pArgs>/<!origin $sysId>/<!cTS $tStamp>
  /myhouse/dnmp/nod/<!nodId>/command/<opId>/<!pType>/<!pArgs>/<!origin $sysId>/<!cTS $tStamp>
}
reply = {
  /<!cmd: command=>reply>/<nodId>/<!rTS $tStamp>
}
roleCert = {
  /myhouse/dnmp/user/<Id>/KEY/_/_/_
  /myhouse/dnmp/operator/<opId>/KEY/_/_/_
}
nodCert = {
  /myhouse/dnmp/nod/<nodId>/KEY/_/_/_
}
deviceCert = {
  /myhouse/device/<devId>/KEY/_/_/_
}
configCert = {
  /myhouse/config/<confId>/KEY/_/_/_
}
dnmpCert = {
  /myhouse/dnmp/KEY/_/_/_
}
netCert = {
  /myhouse/KEY/_/_/_
}
8 cert types, 16 total schemas
 reply: 12 components
 cmd: 10 components, 7 variants
 roleCert: 8 components, 2 variants
 nodCert: 8 components
 deviceCert: 7 components
 configCert: 7 components
 dnmpCert: 6 components
 netCert: 5 components
13 unique literals (67 bytes):
 KEY(7) Pinger(3) all(2) command(8) config(1) device(1) dnmp(11) local(3) myhouse(14) nod(8) operator(1) reply(1) user(1)
5 unique refs (22 bytes):
 Id(5) confId(1) devId(1) nodId(2) opId(4)
7 unique params (30 bytes):
 cTS(7) cmd(1) nodId(2) origin(7) pArgs(7) pType(4) rTS(1)
2 unique built-in functions called (11 bytes):
 sysId(7) tStamp(8)
reference map:
  devId: deviceCert[2]
  confId: configCert[2]
  Id: cmd[5](1,2,3,4) roleCert[3](1)
  opId: cmd[5](5,6,7) roleCert[3](2)
  nodId: nodCert[3] reply[10]
validation chains:
  Id in cmd[5](1,2,3,4) validated by roleCert[3](1)
  opId in cmd[5](5,6,7) validated by roleCert[3](2)
  nodId in reply[10] validated by nodCert[3]
```

Example VerSec compiler diagnostic output

Use of *shim* and *schemer* simplifies Client applications

main loop

```
try {
    // make a CRshim with this target
    CRshim s(target);
    // builds and publishes command and waits for reply
    sendCommand(s);
    s.run();
} catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
}
```

shim builds valid publication from arguments, sets callback

```
void sendCommand(CRshim& shim)
{
    shim.issueCmd(ptype, pargs, processReply);
    if (--count > 0) {
        // wait then launch another command
        timer = shim.schedule(interval, [&shim](){ sendCommand(shim); });
    } else {
        timer = shim.schedule(replyWait, [](){ exit(0); });
    }
}
```

callback: note use of component names

```
void processReply(const Reply& pub, CRshim& shim)
{
    if (const auto& c = pub.getContent(); c.value_size() > 0) {
        std::cout << std::string((const char*)(c.value()), c.value_size()) << "\n";
    }

    // Using the reply timestamps to print client-to-nod & nod-to-client times
    std::cout << "Reply from " << pub["rSrcId"] << ": timing (in sec.): "
              << "to NOD=" + to_string(pub.timeDelta("rTS", "cTS"))
              << "  from NOD=" + to_string(pub.timeDelta("rTS")) << std::endl;
}
```

Use of *shim* and *schemer* simplifies NOD applications

NOD main loop

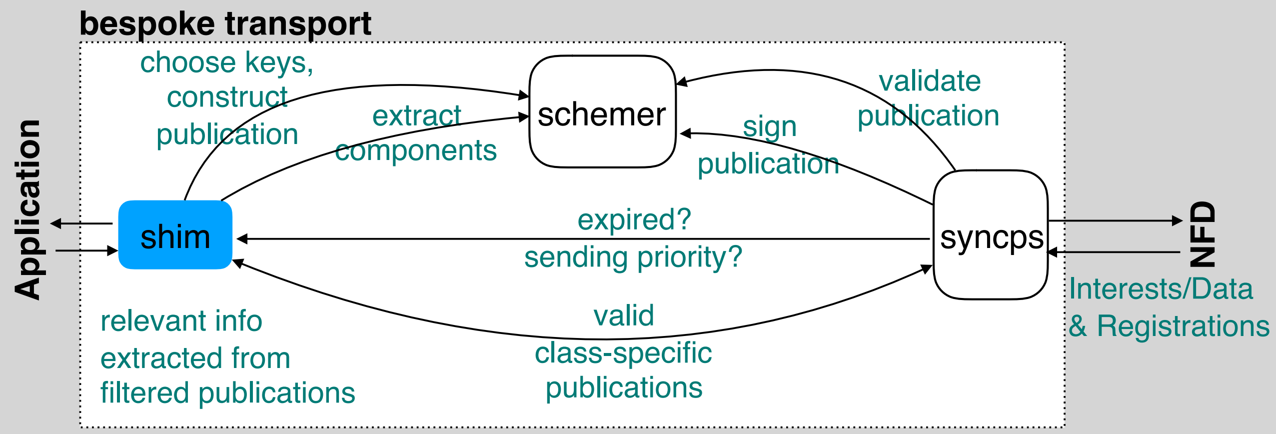
```
auto shims{CRshim::shims("local", "all", CRshim::myPID())};
for (auto& s : shims) s.waitForCmd(probeDispatch);

try {
    shims[0].run();
} catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
}
```

shims subscribe to **command**s in *target* and set callback

callback uses probe function table to invoke method that returns **reply** content

```
static void probeDispatch(RName&& r, CRshim& shim)
{
    try {
        shim.sendReply(r, probeTable.at(r.str("pType"))(r.str("pArgs")));
    } catch (const std::exception& e) {
        std::cerr << e.what() << " for: " << r << std::endl;
    }
}
```

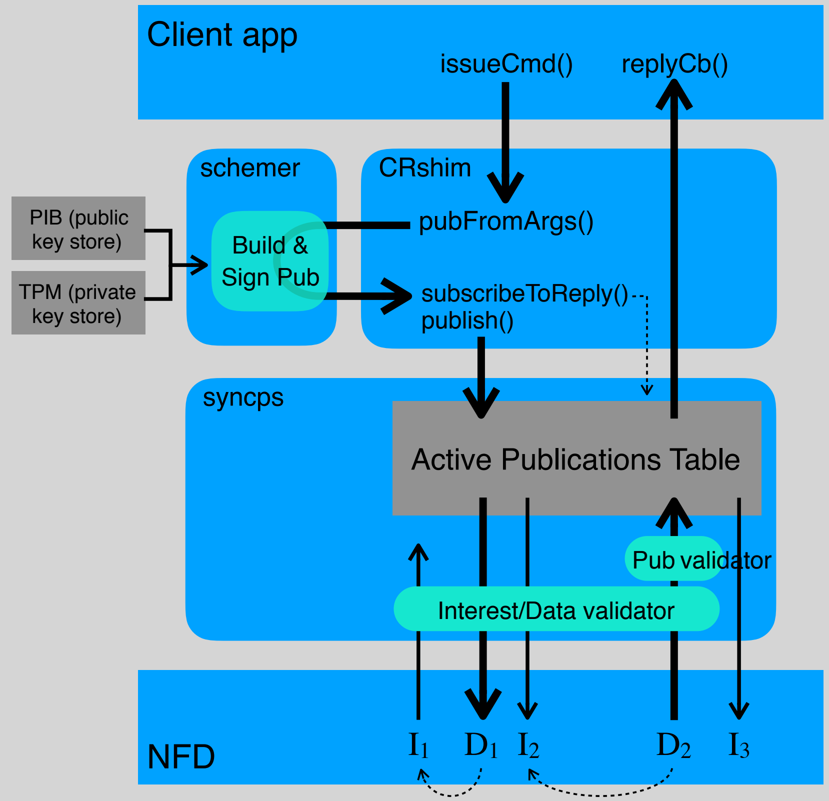
What's in the shim?



shim (173 lines of C++) interacts with applications, schemer, and sync

- provides upcalls to syncps: expire, validate, sign, priority
- uses schemer to construct valid publications
- implements API
  - run() used by applications to pass control to NDN Face through shim
  - waitForCmd() used by NODs to go into state of waiting for a **command** pub
  - sendReply() used by NODs to publish result as **reply** pub
  - issueCmd() used by Clients to turn pType, pArgs, and target into a **command** pub
  - schedule() used by Clients to set timer

Command-to-Reply data flow



- All NODs in sync, waiting for command(s). Single pending Interest **I_1**.
- Client issues a **command** which results in sending Data **D_1** satisfying **I_1** and an updated Interest **I_2** to solicit **reply**(s).
- NOD sends **reply** in Data **D_2** satisfying **I_2** which gets delivered to client via reply callback set when command issued. Also results in updated Interest **I_3** to solicit more **reply**(s).

1. In steady-state all peers are in sync so there's one Interest being refreshed, **I_1,** containing the IBLT of their common set of pubs.
2. When the client calls issueCmd, the CRshim calls the schemer with its target plus the supplied probe type and args. The schemer finds the set of command publications allowed for target, pType & pArgs then scans the user's TPM looking for key(s) that could validly sign these publications. If one is found, that key, the associated publication's schema and the caller's target, pType & pArgs are enough to construct and sign a complete 'command' publication which is returned to the shim.
3. Given the complete command, the shim know what reply(s) to it look like so it calls syncps to subscribe (which will result in the client supplied reply callback (replyCb above) being called with each arriving reply as soon as it has been validated.
4. Once the reply subscription is in place, the shim calls syncps 'publish' to add the command to the current set of active publications. Since there's now a publication not in the set described by **I_1,** syncps constructs an NDN Data, **D_1,** containing the command pub and satisfies **I_1** with it. It also sends a new Interest **I_2** announcing it holds the common set plus the new command.
5. NODs receive **D_1**, extract and validate the command then generate a reply which results in sending a new Data **D_2** satisfying **I_2**.
6. Syncps receives and validates **D_2** then the reply pub it contains then adds the reply to the active publications which triggers a callback to the client's reply handler. Adding the reply also triggers sending of a new Interest **I_3** soliciting additional replies.

Status / Summary

- Co-development of DNMP with VerSec, bespoke transport modules proved extremely helpful
  - bespoke transport modular model eased development
  - VerSec toolkit takes trust schema design to useful code
  - using information-centric approach to measurement seems intuitive
  - our approach and new tools make application implementation more straightforward
- Features that are underway
  - talk covered *current* release of DNMP, some differences from paper
  - snoop shim for Audit application
  - direct instrumentation of NFD that NOD probes can query
  - keep tuning multicast strategy for broadcast networks
  - would like some sort of protobuf for Data content (please, someone?)
- DNMP is open source GPL-3.0

All the files in DNMP release total 1280 lines of code and 642 comment lines

% linesofcode {,syncps/}*.[ch]pp

lines	code	comment	blank	file
291	173	93	25	CRshim.hpp
157	98	46	13	bh-client.cpp
186	119	54	13	generic-client.cpp
117	57	43	17	nod.cpp
336	230	73	33	probes.hpp
432	265	111	56	syncps/iblt.hpp
614	338	222	54	syncps/syncps.hpp
2133	1280	642	211	total

Links

DNMP release at https://github.com/pollere/DNMP

NFD patches at: https://github.com/pollere/NDNpatches

Versatile Security toolkit at: https://github.com/pollere/versec (by 09.30.19)

Van Jacobson on VerSec (in brief): http://pollere.net/Pdfdocs/ICN-WEN-190715.pdf, https://vimeo.com/354013644

NDNcomm 2019 slides: http://pollere.net/Pdfdocs/BuildingBridge.pdf

Example: VerSec compiler input for NLSR

```
# NLSR schema (from github/named-data/NLSR/docs/SECURITY-
CONFIG.rst)

# site-specific config
net = ndn
site = edu/ucla

# site-independent config

# entities
operator = Operator/<opId>
rtr = Router/<rtrName>

# packet names

# (format from nlsr/src/hello_protocol.cpp)
# 3rd parameter is <net>/<site>/<rtr> prefix but stuck into one
# component so it can't be validated.
hello = <net>/<_nsite>/<_nrtr>/nlsr/INFO/<_rtr>/<_version>

# (format from nlsr/src/lsdb.hpp)
discovery = <_seqNo>
segment = <_seqNo>/<_version>/<_segmentNo>

lsa = localhop/<net>/nlsr/LSA/<site>/<rtr>/<_type>/(<discovery>|
&lt;segment&gt;)

packet = <hello> | <lsa>

# key names

# <_KEY> is a built-in definition of the 4 parameters that terminate
# an NDN key name: KEY/<_keyId>/<_issuerId>/<_version> (see
# http://named-data.net/doc/ndn-cxx/current/specs/certificate-
format.html)
# This info is validated by the key's signature, not the schema

netCert = <net>/<_KEY>
siteCert = <net>/<site>/<_KEY>
opCert = <net>/<site>/<operator>/<_KEY>
rtrCert = <net>/<site>/<rtr>/<_KEY>
nlsrCert = <net>/<site>/<rtr>/nlsr/<_KEY>

# signing chain

packet <= nlsrCert <= rtrCert <= opCert <= siteCert <= netCert
```

- 15 lines of code, 15 lines of comments

Performance Issues

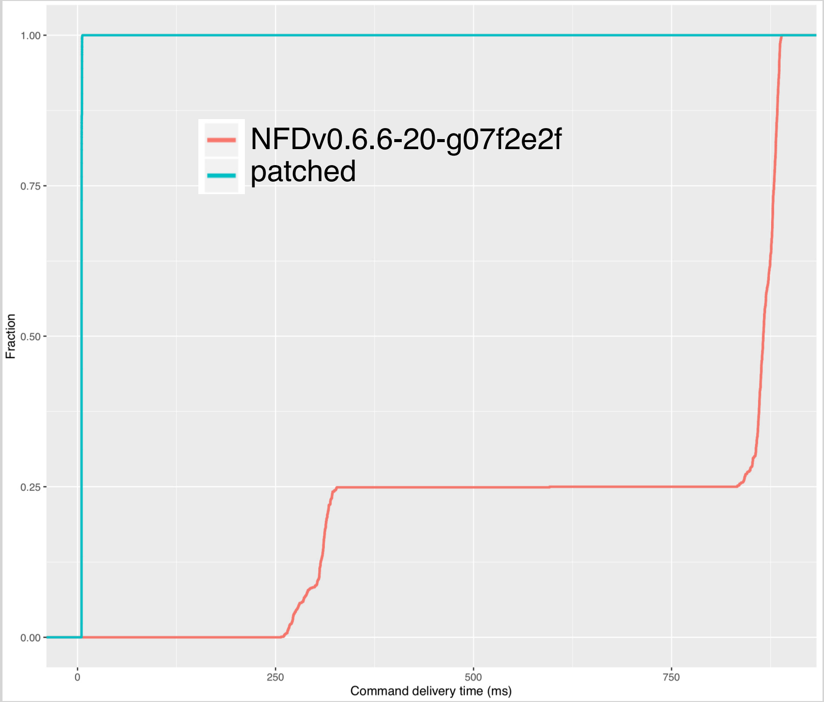
- “This doesn’t work the way you think it does”
- The NFD code doesn’t always match the architecture, particularly devastating impact on multicast
  - Interests are not held in PIT until timeout, but only put in PIT on forward
  - PIT not checked on new FIB entry, e.g. new registration
  - LP::Nacks cause premature Interest death
  - No Interest suppression reduces efficiency
  - RETX suppression causes premature Interest death
- Patches completed for these problems
  - Mostly involve *removing* code
  - Insufficient broadcast testing is being done on codebase additions

Patches for LP::Nack and PIT discard issues

Test uses echo measurement (origination timestamps of both initial Command and its Reply) 20,000 exchanges:

Before patch: mean=730ms, median=866ms

After patch: mean=median=5ms



Takeaway: rigorous application-driven testing and measurement must be performed so that applications get known quantity