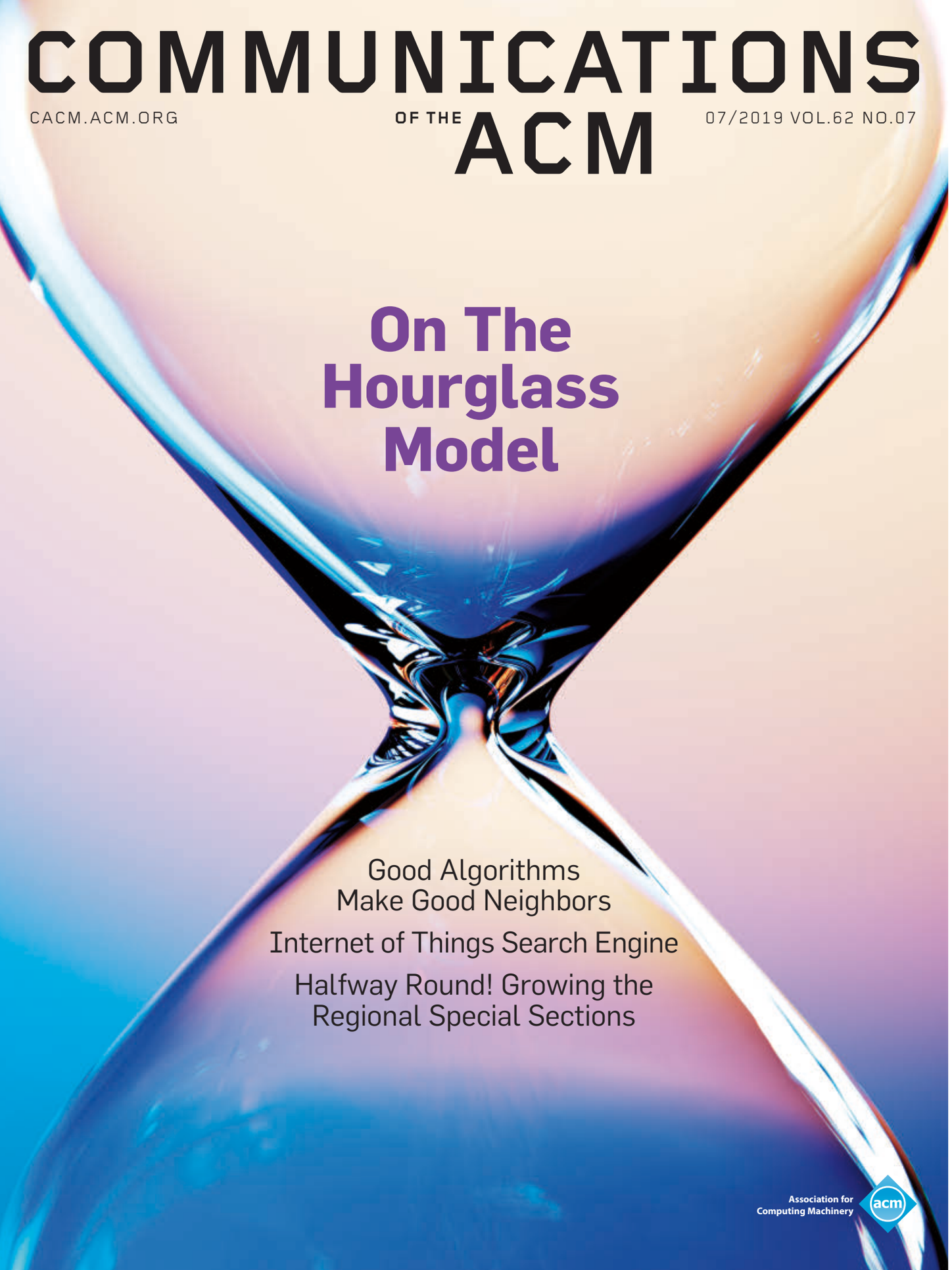
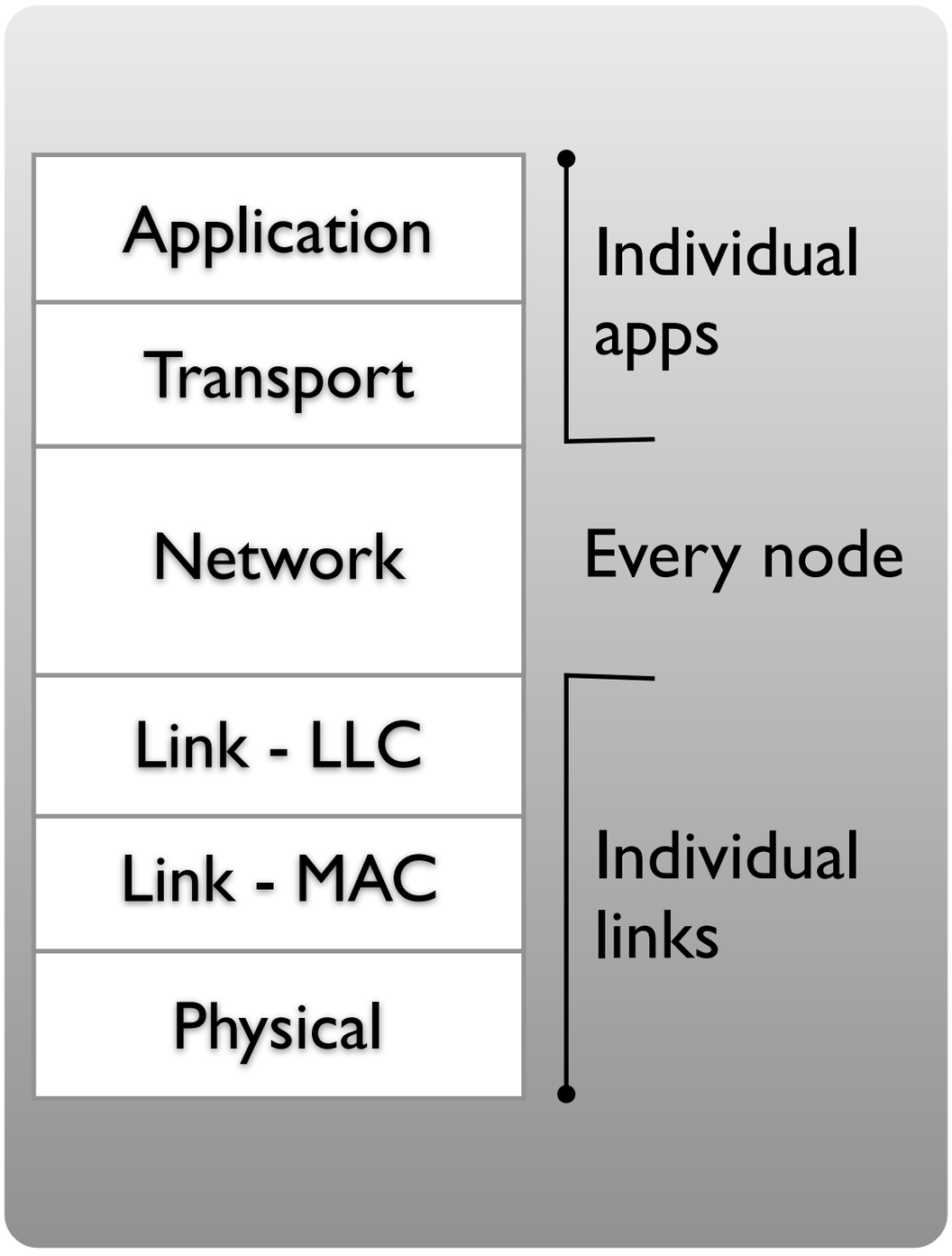
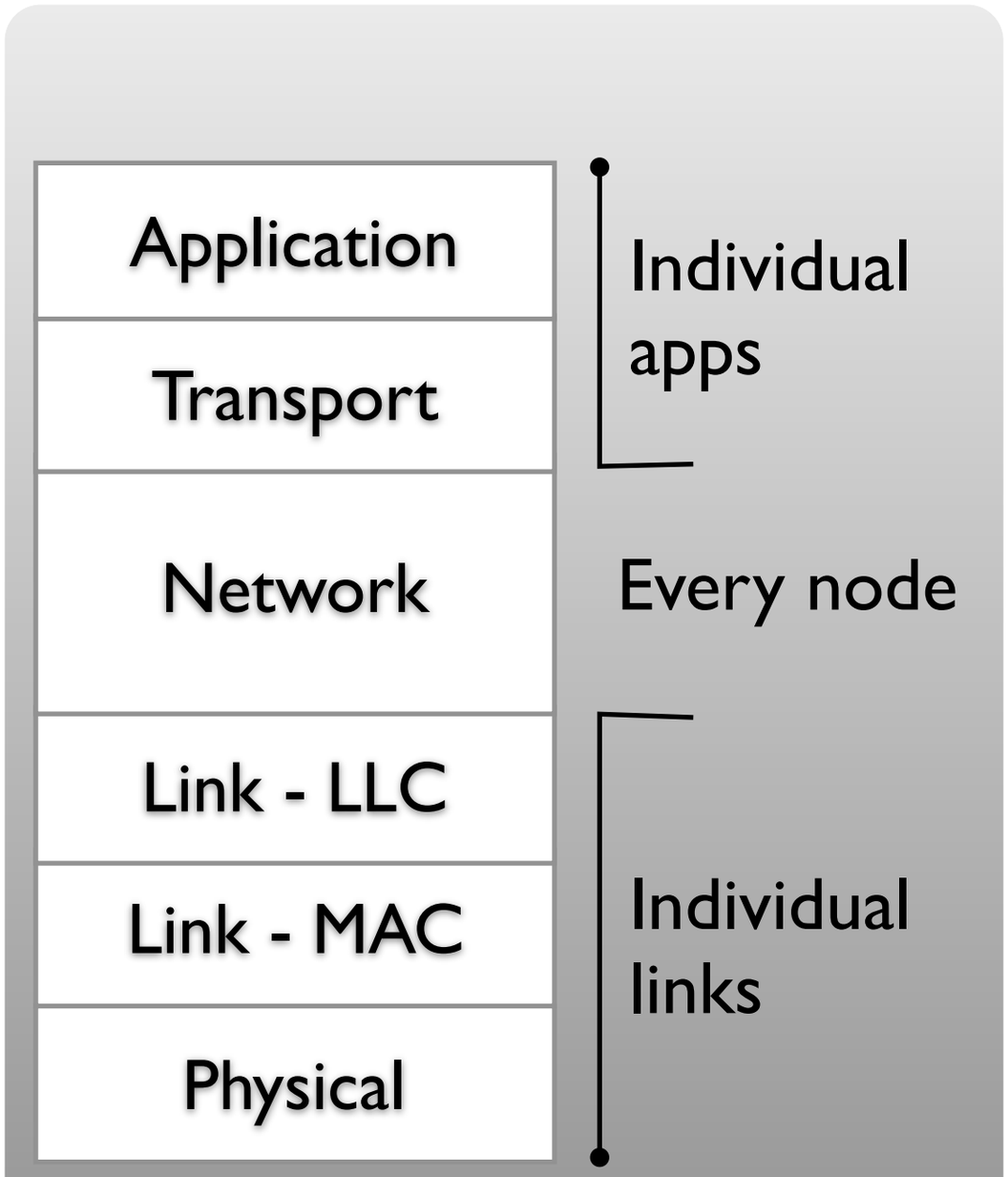
Watching NDN's Waist: How Simplicity Creates Innovation and Opportunity

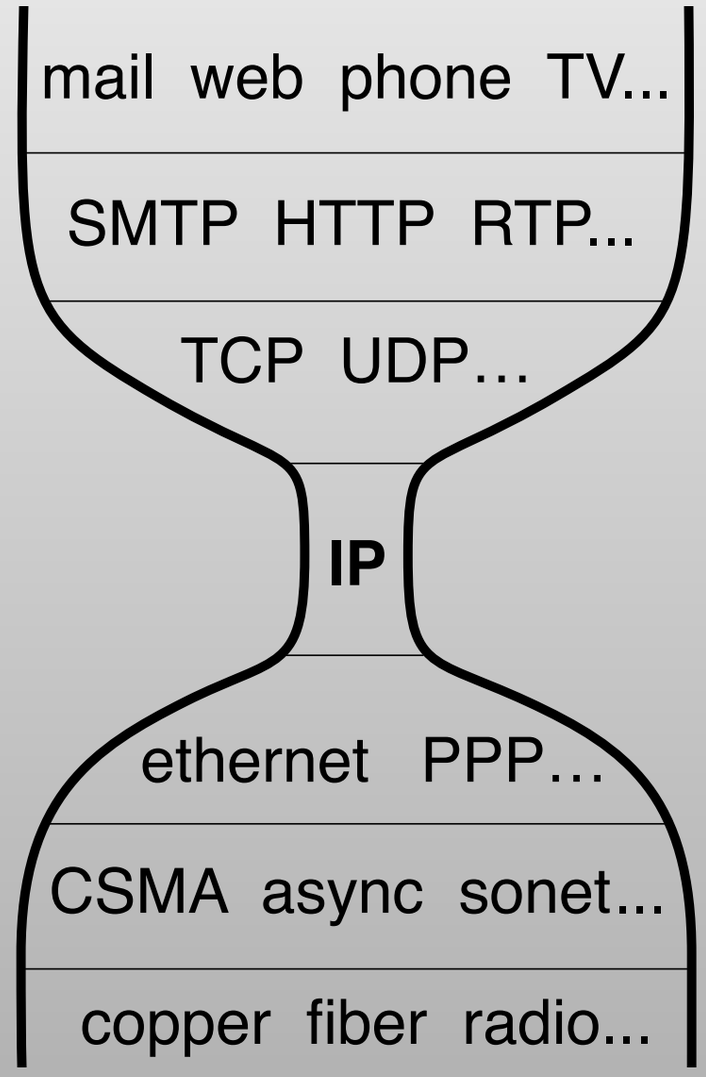
Van Jacobson, UCLA
NSF/Intel ICN-WEN Annual Meeting
12 July 2019, Santa Clara, CA



> “We are screwing up in our design of internet protocols by violating the principle of layering. Specifically we are trying to use TCP to do two things: serve as a host level end to end protocol, and to serve as an internet packaging and routing protocol. These two things should be provided in a layered and modular way. I suggest that a new distinct internetwork protocol is needed, and that TCP be used strictly as a host level end to end protocol.”
>
> — *Jon Postel*, IEN #2, August 1977


Application
Transport
Network
Link - LLC
Link - MAC
Physical
Individual apps
Every node
Individual links

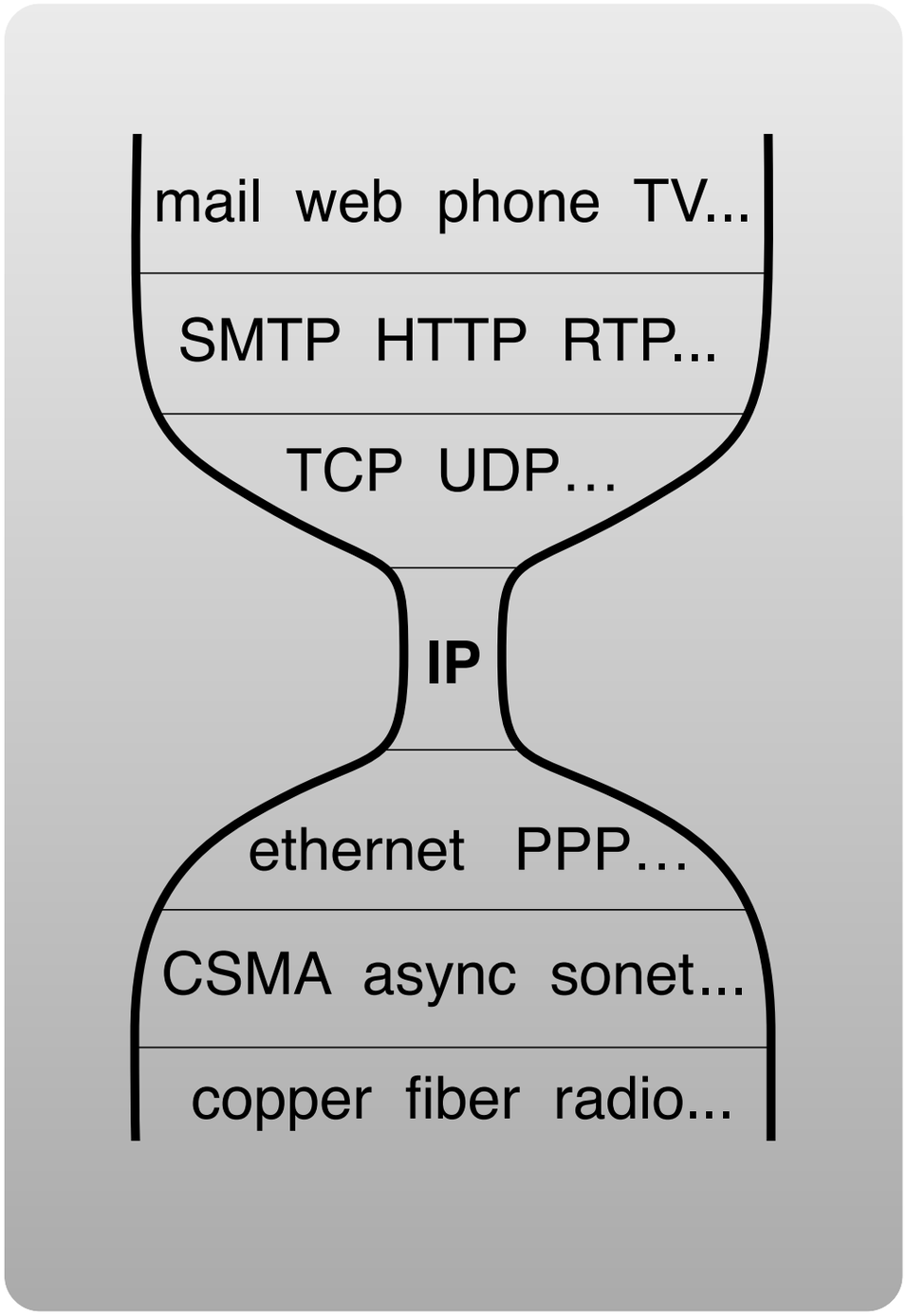

Application
Transport
Network
Link - LLC
Link - MAC
Physical
Individual apps
Every node
Individual links


mail web phone TV...
SMTP HTTP RTP...
TCP UDP…
IP
ethernet PPP…
CSMA async sonet...
copper fiber radio...

The waist does its job only when it *stays* narrow—simple and stable.

Under Jon's stewardship as RFC Editor, the Internet's first five years were spent *removing* stuff from IP (precedence bits, source routing, redirects, information request, source quench, fragmentation, host & net unreachable msgs, …).

The slimmed-down result has served us well for 35 years.



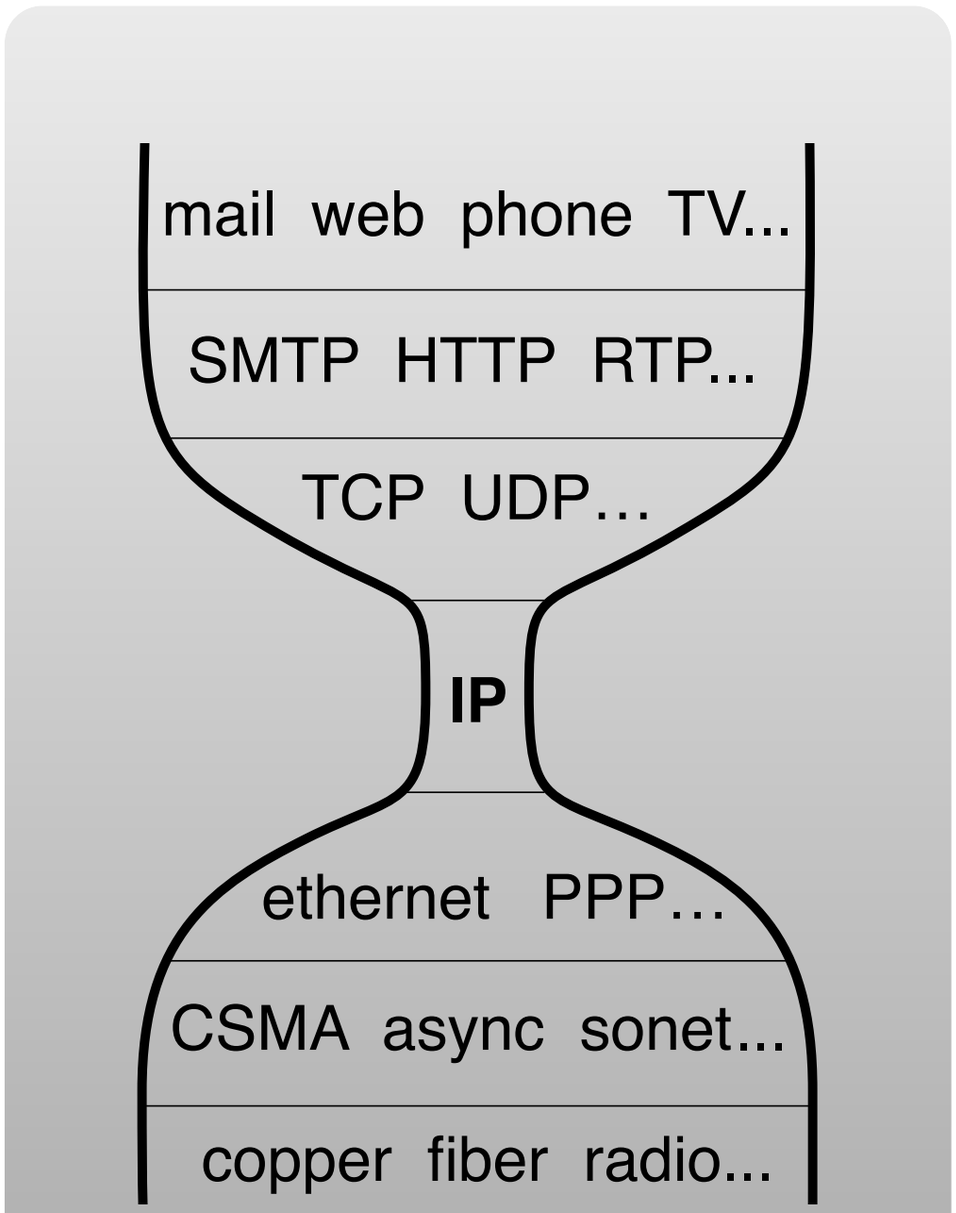
MTP HTTP RTP...

TCP UDP...

IP

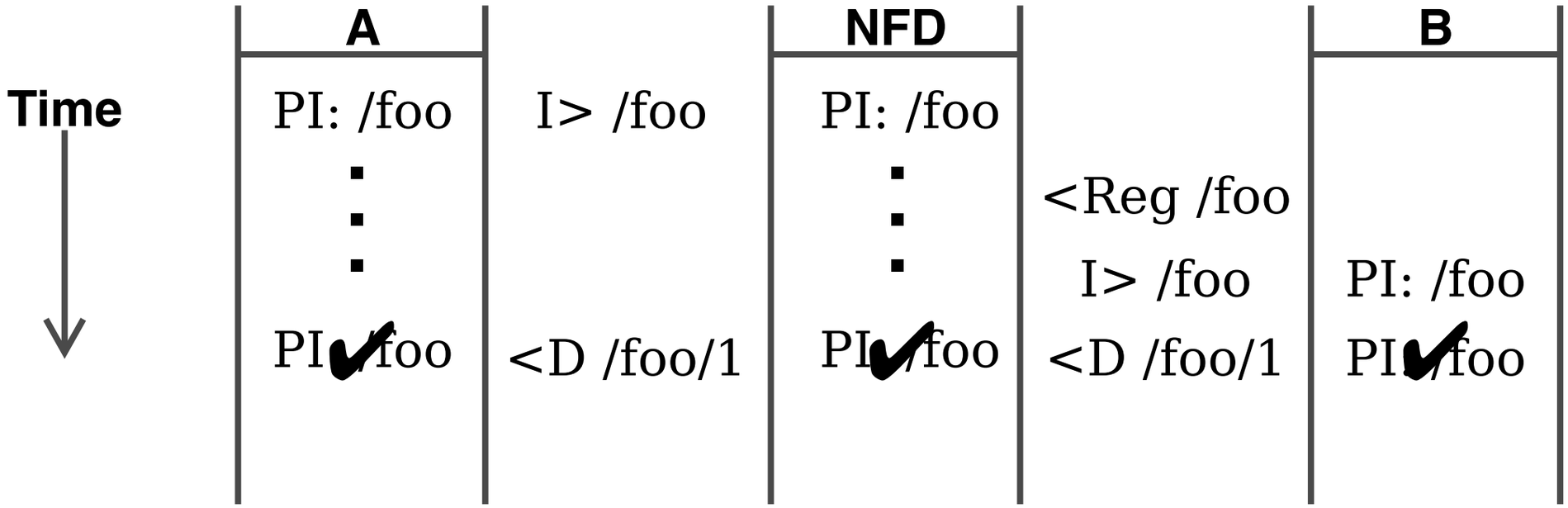
ethernet PPP...

CSMA async sonet...



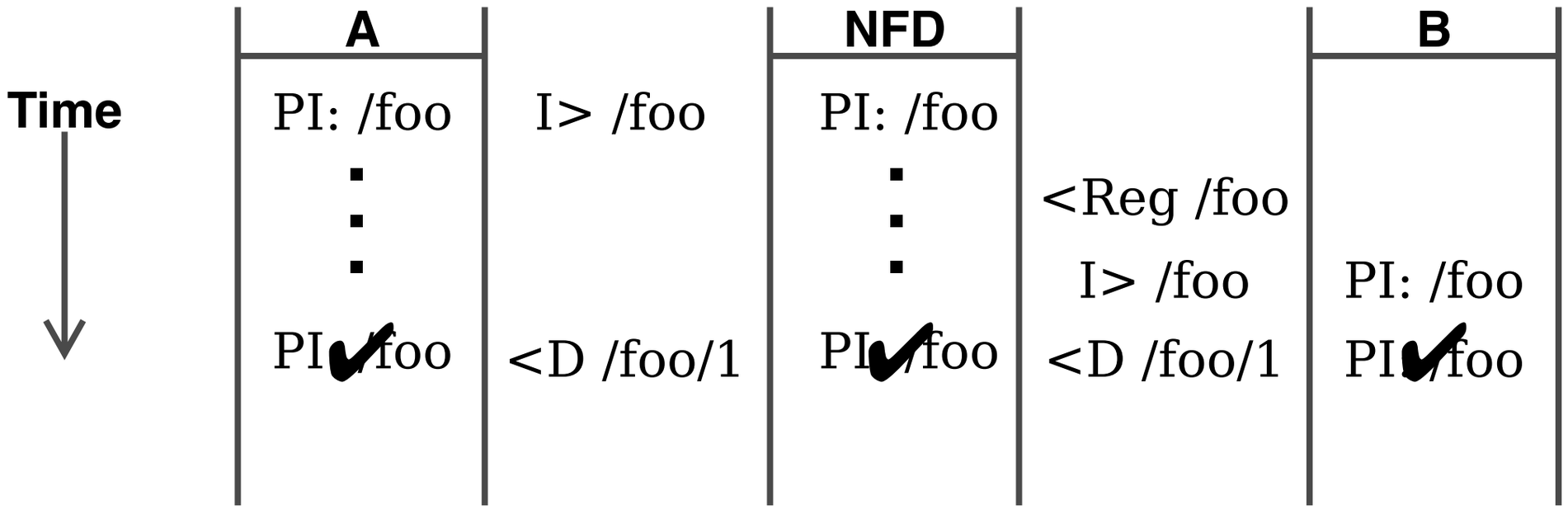
Fattening NDN's Waist: Nacks

What's supposed to happen:

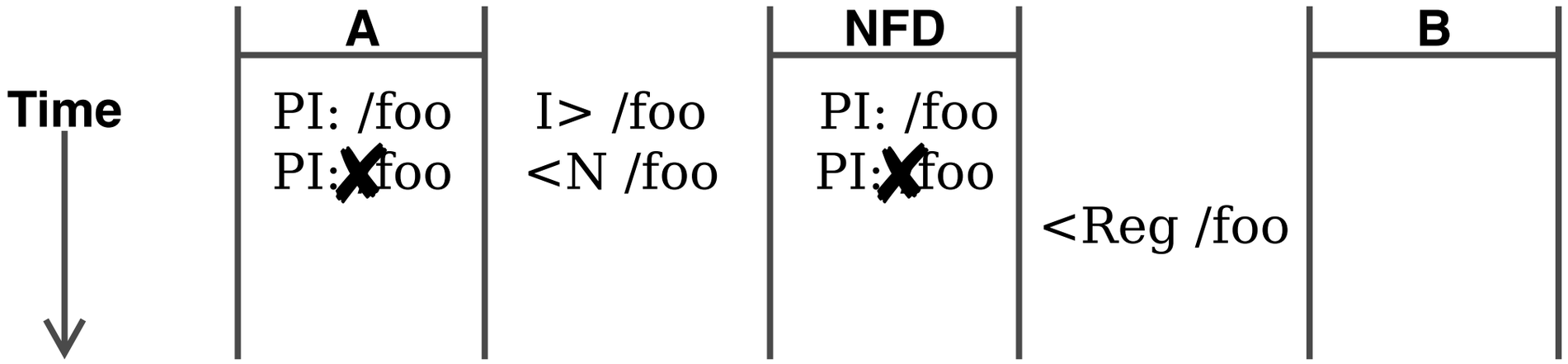


Fattening NDN's Waist: Nacks

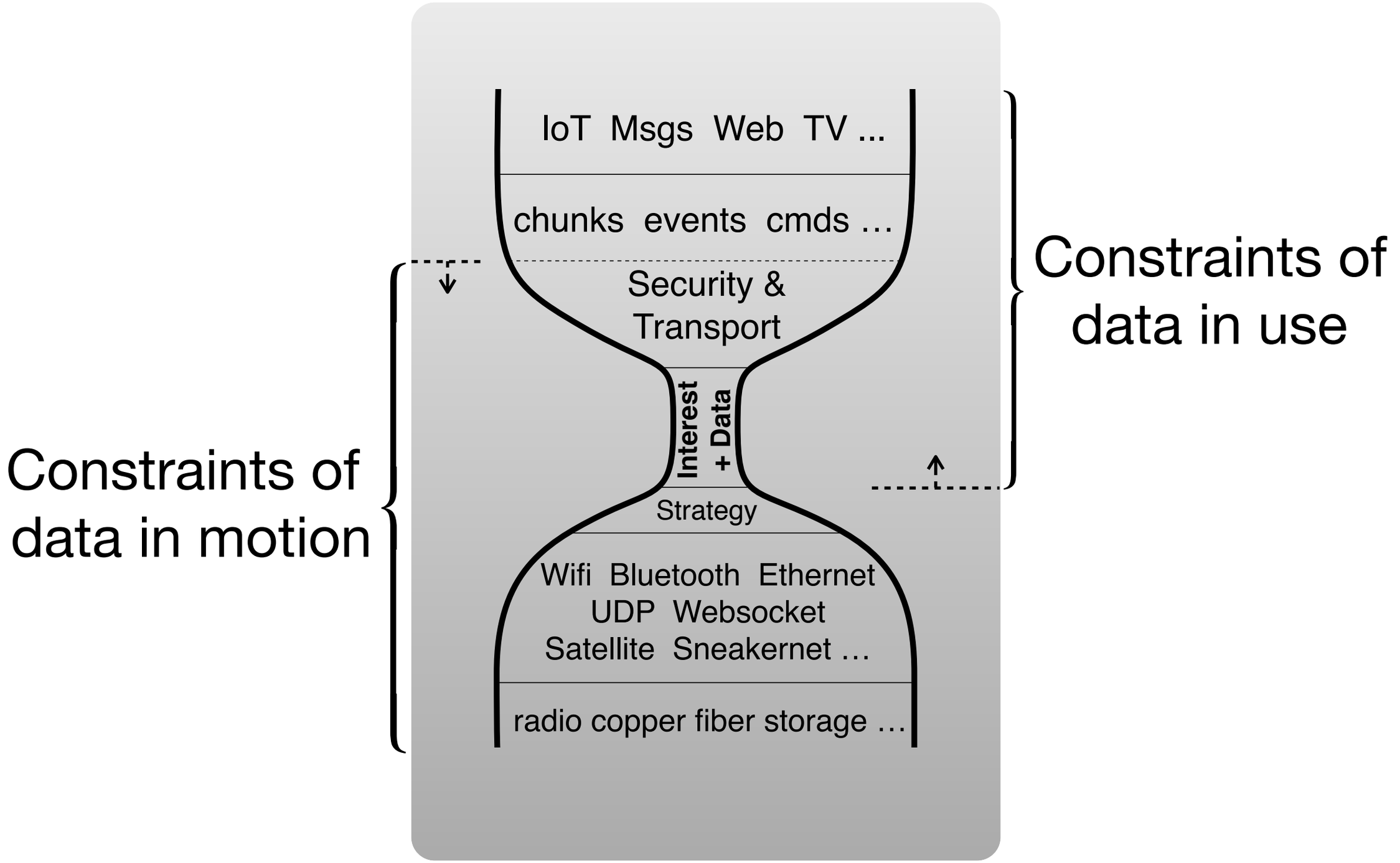
What's supposed to happen:



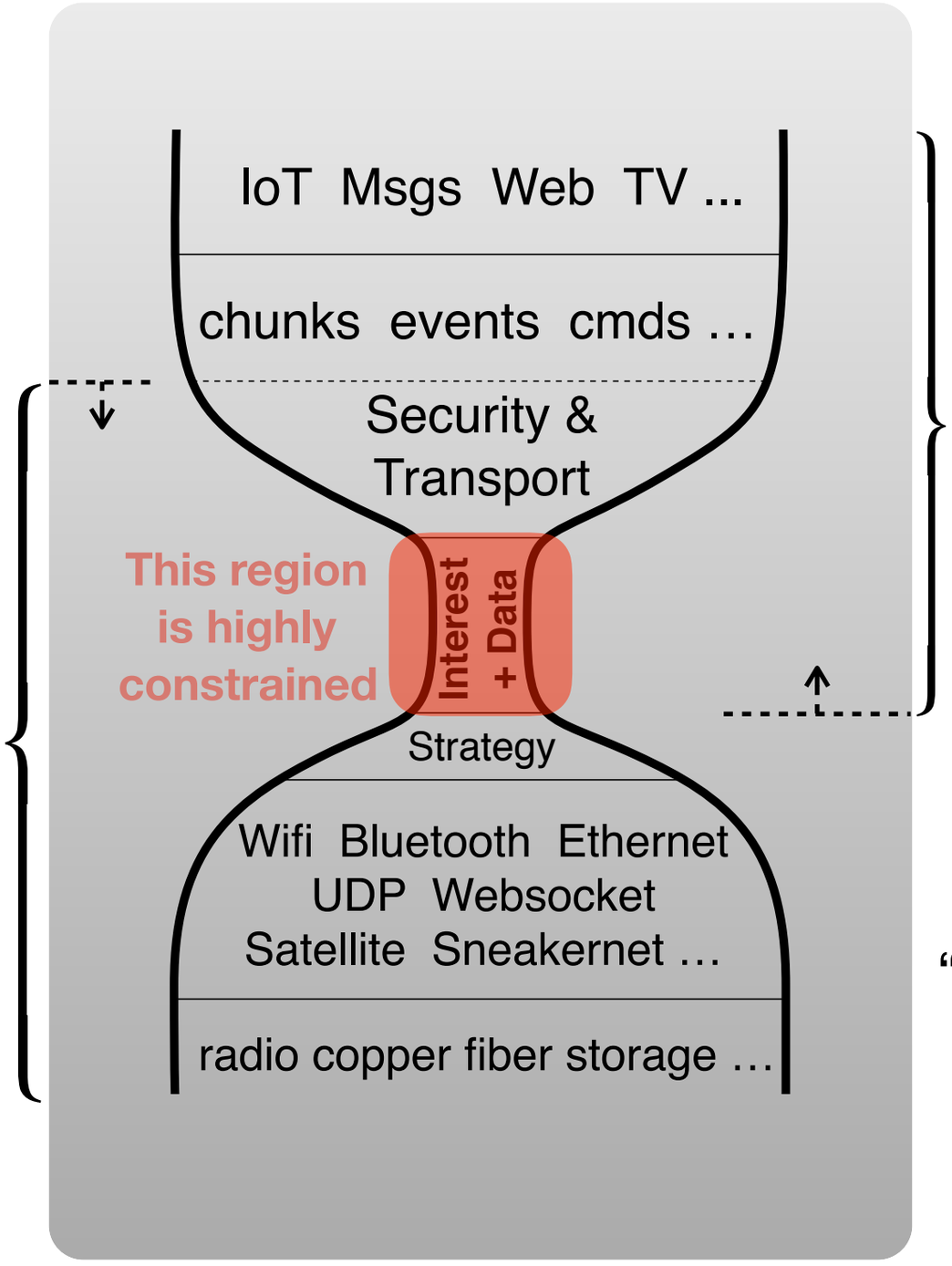
What's currently happens:



What does this kind of problem have to do with the waist?



What does this kind of problem have to do with the waist?

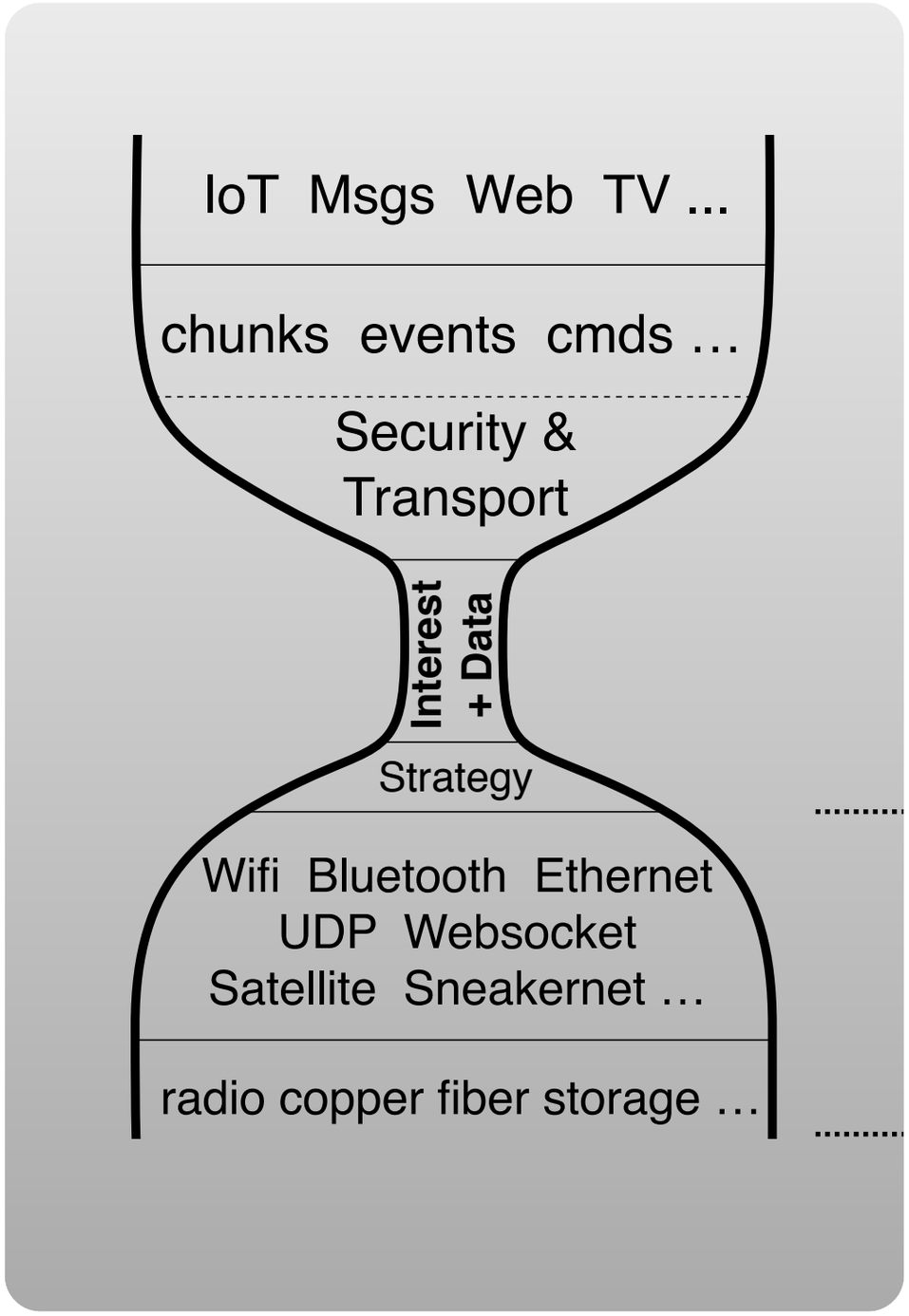


Constraints of data in motion

Constraints of data in use

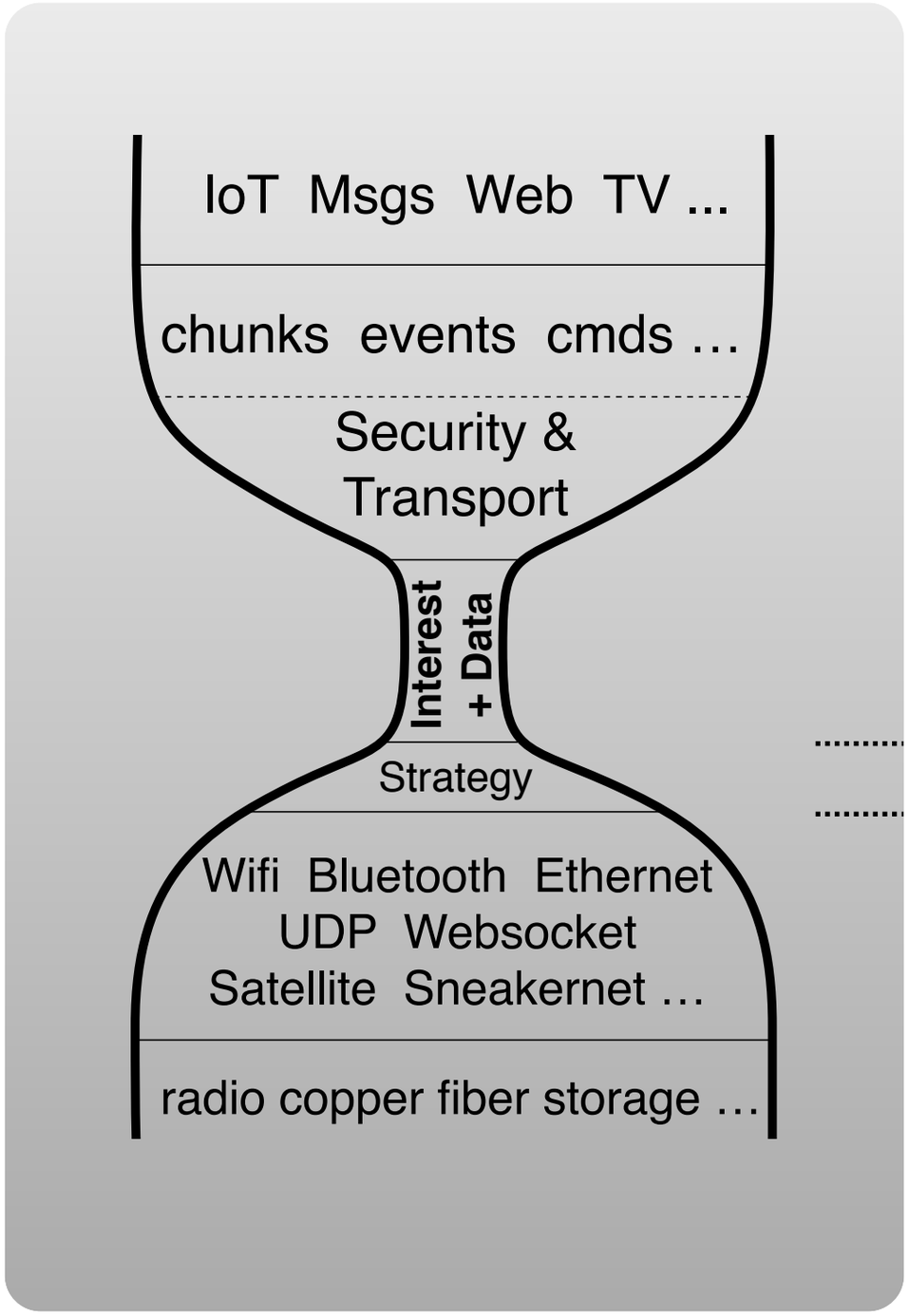
Changes in the waist are far more likely to produce “unintended consequences”

Opportunities?



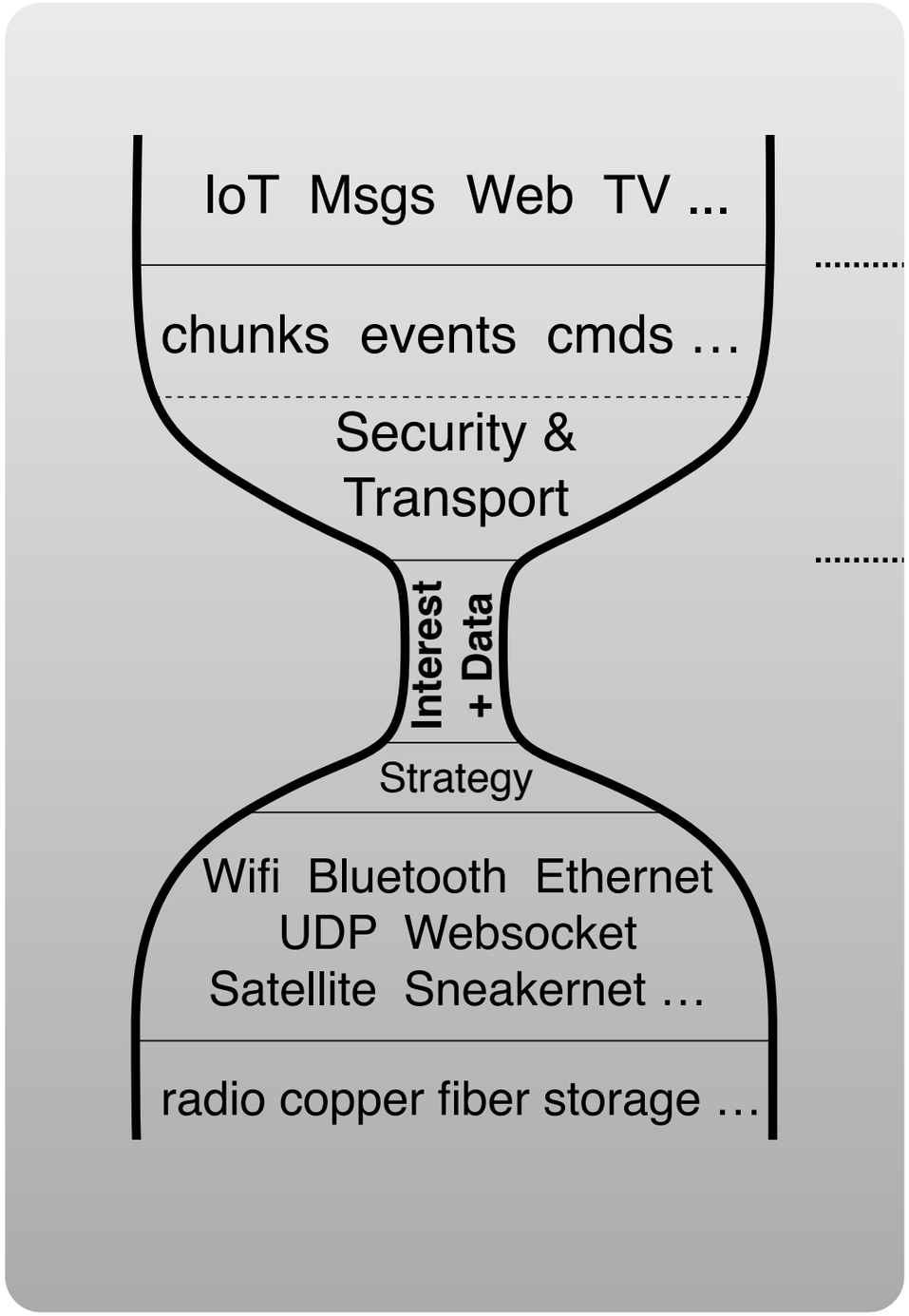
NDN doesn’t have to create ‘endpoint’ abstractions.
It runs over anything with no added latency or overhead.

Opportunities?



NDN cannot create forwarding loops so doesn't require routing to function. A node can always communicate using any & all of its capabilities.

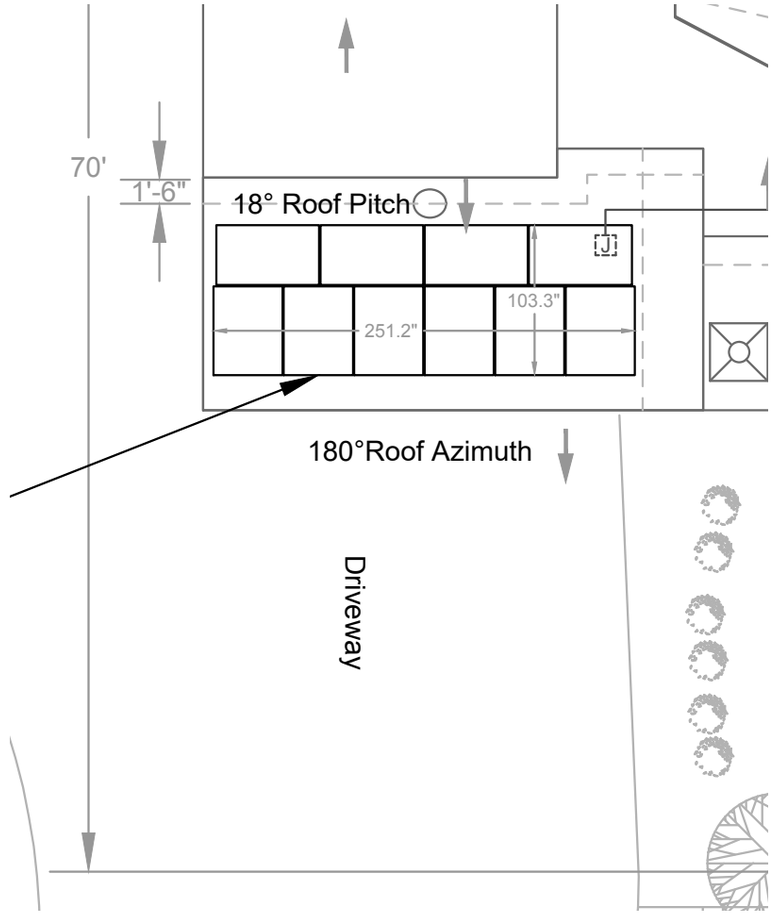
Opportunities?



Application's security & communication needs jointly create a 'Bespoke transport'

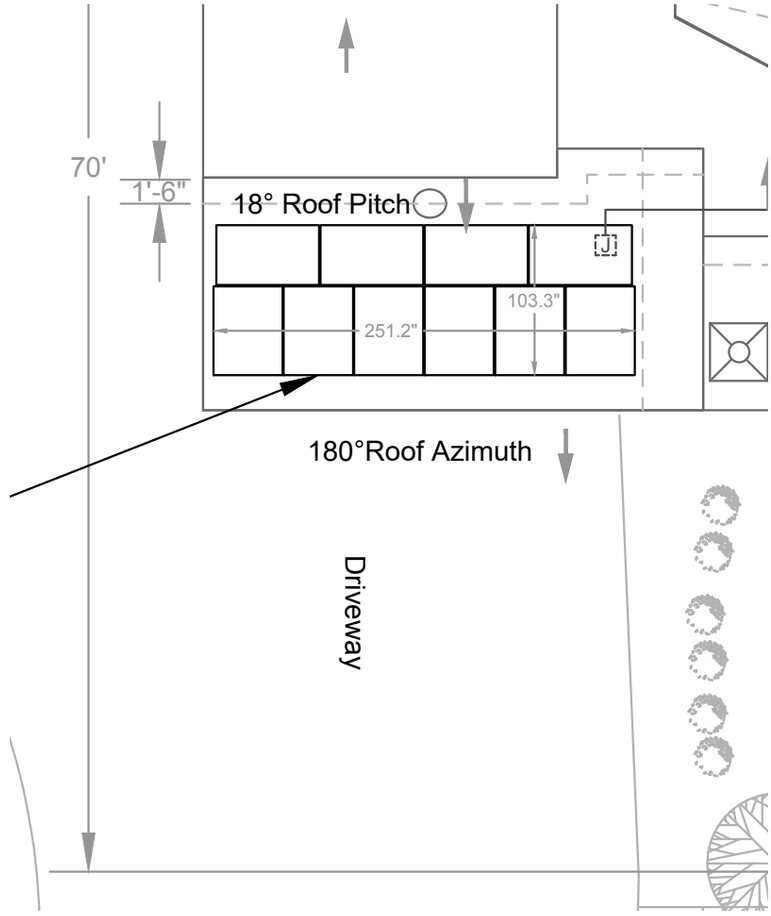
One set of building plans does three things:

- Only permits designs that meet code
- Constructs the building
- Validates that as-permitted matches as-built



One set of building plans does three things:

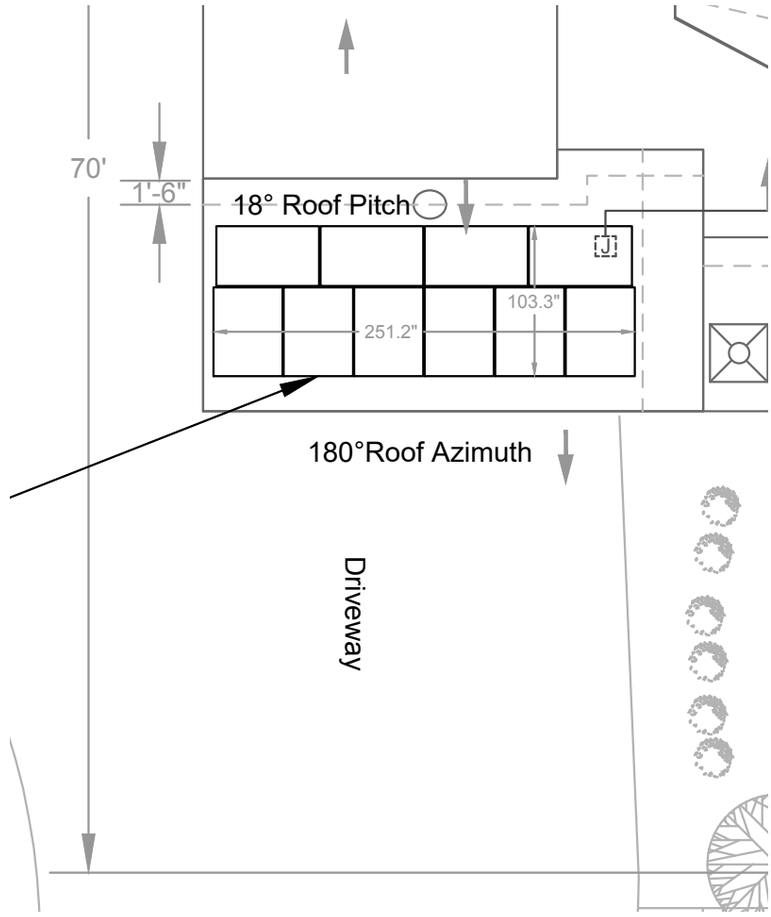
- Only permits designs that meet code
- Constructs the building
- Validates that as-permitted matches as-built



NDN Trust Schemas are detailed, fine-grained rules for *receivers* to validate the syntax, access and authorizations of NDN communications.

One set of building plans does three things:

- Only permits designs that meet code
- Constructs the building
- Validates that as-permitted matches as-built

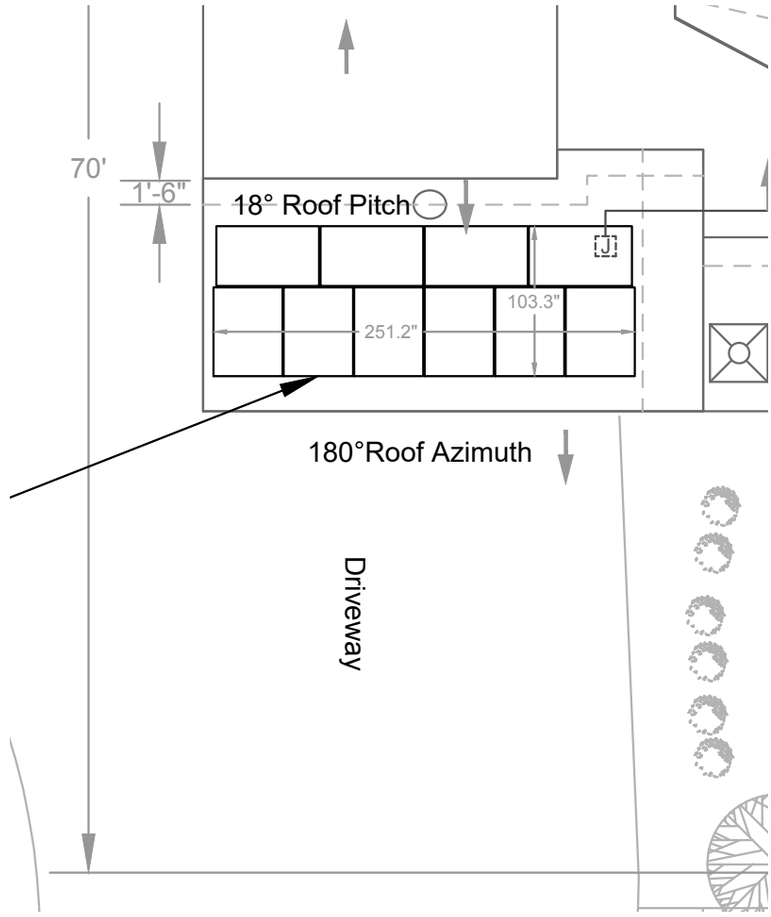


NDN Trust Schemas are detailed, fine-grained rules for *receivers* to validate the syntax, access and authorizations of NDN communications.

With some thought, the same schema can be used by *senders* to automatically choose signing keys and construct valid communications.

One set of building plans does three things:

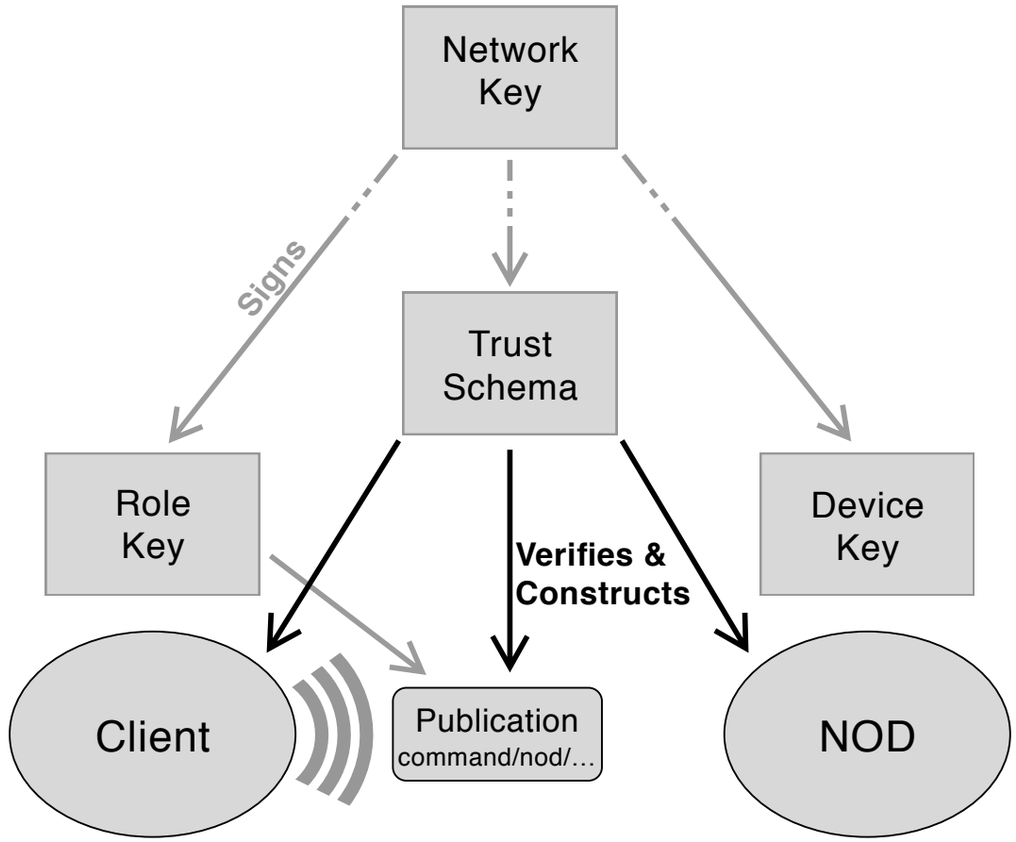
- Only permits designs that meet code
- Constructs the building
- Validates that as-permitted matches as-built



NDN Trust Schemas are detailed, fine-grained rules for *receivers* to validate the syntax, access and authorizations of NDN communications.

With some thought, the same schema can be used by *senders* to automatically choose signing keys and construct valid communications.

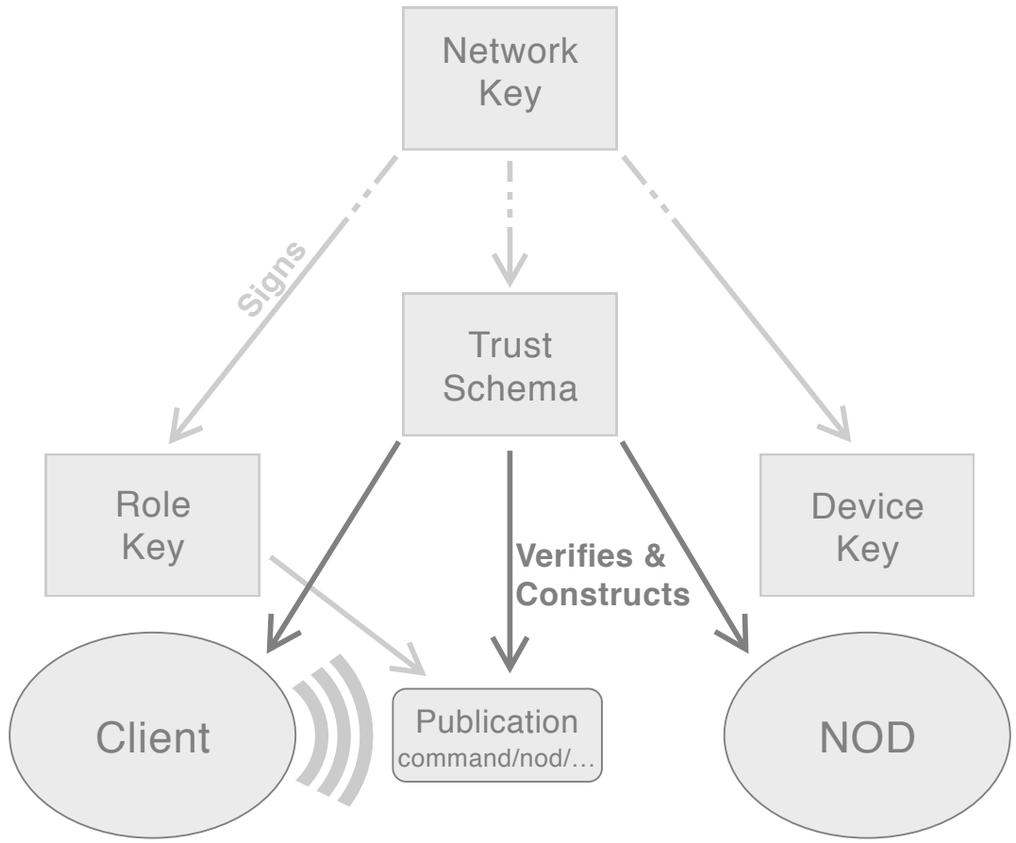
Encouraged by Kathie Nichols, I spent Christmas break building a system to do this together with a IBLT-based pub-sub bespoke transport.



Trust schemas are the local network's law.

They take the form of an NDN 'key' which can be created, accessed & validated using the standard security library.

They must be signed by an 'installer' whose authority derives from the network's trust anchor.



Trust schemas are the local network's law.

They take the form of an NDN 'key' which can be created, accessed & validated using the standard security library.

They must be signed by an 'installer' whose authority derives from the network's trust anchor.

Trust schemas are written in a simple, declarative language.

This is parsed and compiled to a compact binary form stored as an NDN key.

Once signed by an installer, this key is used by a C++ runtime library to construct, deconstruct, sign and validate publications.

command pub definition and signing chain
cpub = <domain>/**nod**/<nodSpec>/**command**/<roleType>/<ID>/<origin>/**probe**/<pType>/<pArgs>/<timestamp>
roleCert = <domain>/<roleType>/<ID>/<_key>
dnmpCert = <domain>/<_key>
domain = <root>/**dnmp**
cpub <= roleCert <= dnmpCert <= netCert
reply pub definition and signing chain
rpub = <cpub **command** => **reply**>/<dCnt>/<rSrcID>/<rtimestamp>
nodCert = <domain>/**nod**/<nodID>/<_key>
devCert = <root>/**device**/<devID>/<_key>
configCert = <root>/**config**/<configID>/<_key>
rpub <= nodCert <= deviceCert <= configCert <= netCert

Since the schema does all the work, programs get simple

```
// print every NFD’s route to some prefix
//  routesTo <prefix>
#include "dnmpCommandAPI.hpp"
int main(int argc, char* argv[])
{
  char* prefix = nullptr;
   … parse args

  try {
    CRshim s("nod/all");
    s.doCommand("NFDRIB", prefix,
        [](const Reply& r) {
           cout << r["nodID"] << " reply took "
                << r.timeDelta("rPubTime") << " secs:"
                << r["content"] << endl;
         });
  } catch (const std::exception& e) {
      cerr << e.what() << endl;
  }
}
```

Since the schema does all the work, programs get simple

This is CLI boilerplate

```
// print every NFD’s route to some prefix
//  routesTo <prefix>
#include "dnmpCommandAPI.hpp"
int main(int argc, char* argv[])
{
  char* prefix = nullptr;
   … parse args

  try {
    CRshim s("nod/all");
    s.doCommand("NFDRIB", prefix,
       [](const Reply& r) {
          cout << r["nodID"] << " reply took "
               << r.timeDelta("rPubTime") << " secs:"
               << r["content"] << endl;
        });
  } catch (const std::exception& e) {
    cerr << e.what() << endl;
  }
}
```

Since the schema does all the work, programs get simple

```
// print every NFD's route to some prefix
//  routesTo <prefix>
#include "dnmpCommandAPI.hpp"
int main(int argc, char* argv[])
{
  char* prefix = nullptr;
   … parse args

  try {
    CRshim s("nod/all");
    s.doCommand("NFDRIB", prefix,
       [](const Reply& r) {
          cout << r["nodID"] << " reply took "
               << r.timeDelta("rPubTime") << " secs:"
               << r["content"] << endl;
        });
  } catch (const std::exception& e) {
     cerr << e.what() << endl;
  }
}
```

This delivers a command to all NODs then collects their replies

Since the schema does all the work, programs get simple

```
// print every NFD's route to some prefix
//  routesTo <prefix>
#include "dnmpCommandAPI.hpp"
int main(int argc, char* argv[])
{
  char* prefix = nullptr;
   … parse args

  try {
    CRshim s("nod/all");
    s.doCommand("NFDRIB", prefix,
      [](const Reply& r) {
        cout << r["nodID"] << " reply took "
             << r.timeDelta("rPubTime") << " secs:"
             << r["content"] << endl;
      });
  } catch (const std::exception& e) {
      cerr << e.what() << endl;
  }
}
```

This is application's callback to handle each arriving reply