Building a Multiplayer Online Game (MOG) using Named Data Networking (NDN)

MAI Research Project
Stefano Lupo
Supervisor: Dr Stefan Weber

Introduction

IP and NDN

IP: The *"narrow waist"* of the Internet

- IP is the *universal network layer*
- All internet connected devices speak IP

- Entirely host centric abstraction
  - Designed for point-to-point communication
  - Influenced by telephone networks of 20th Century

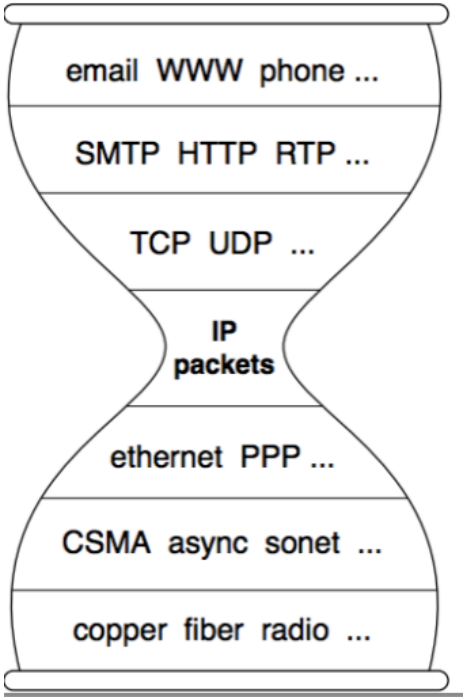


Figure: NDN Executive Summary, https://named-data.net/project/execsummary/

Named Data Networking

- Instance of Information Centric Networking (ICN)
- Move to an entirely data centric abstraction

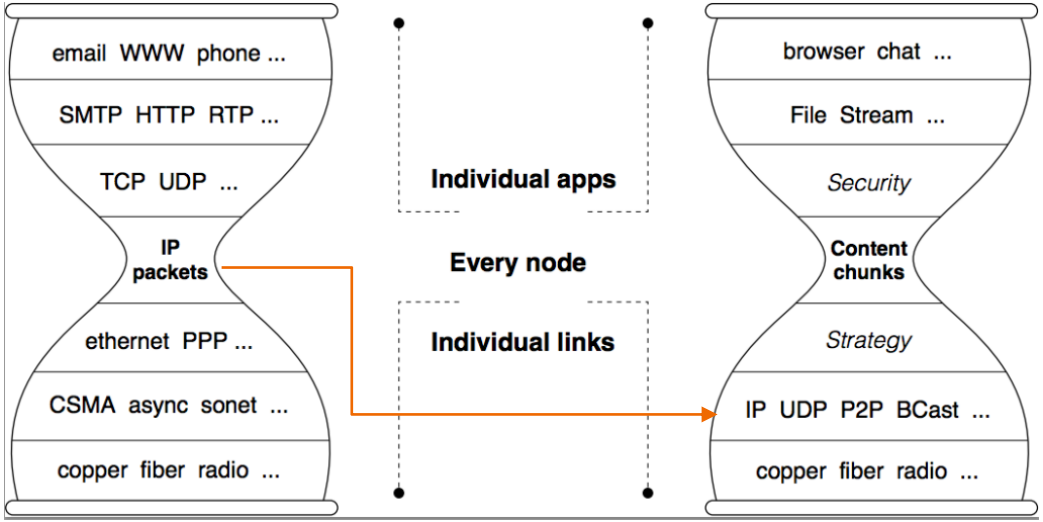
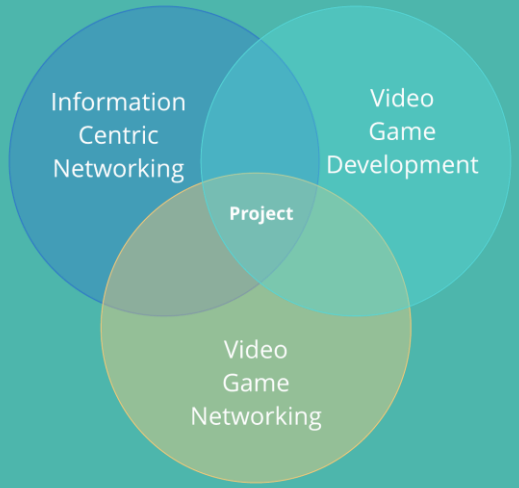


Figure: NDN Executive Summary, https://named-data.net/project/execsummary/

A brief overview of the State of the Art



How does NDN work?

NDN Primitives

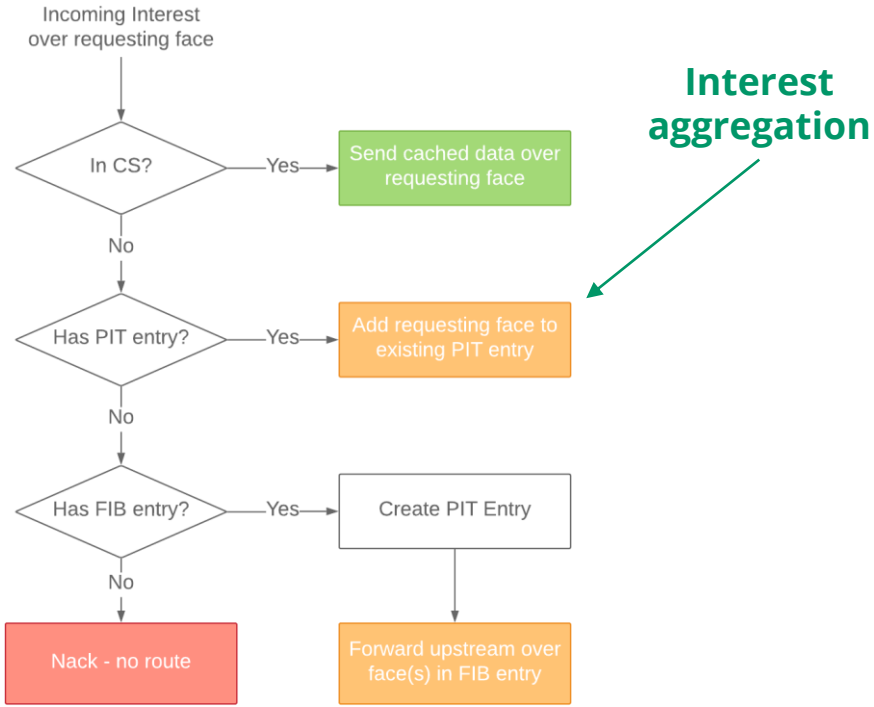
- Interest packet
  - Contains the name of data they refer to
  - Consumers request data from the network by *expressing Interests*
- Data packet
  - Produced in response to Interest packets
  - Contain the name of the data they encapsulate
  - Signed by producers
- Face abstraction
  - Physical interface - e.g. network card
  - Logical interface - e.g. application producing data under namespace

How does NDN work?

Three Key Data Structures

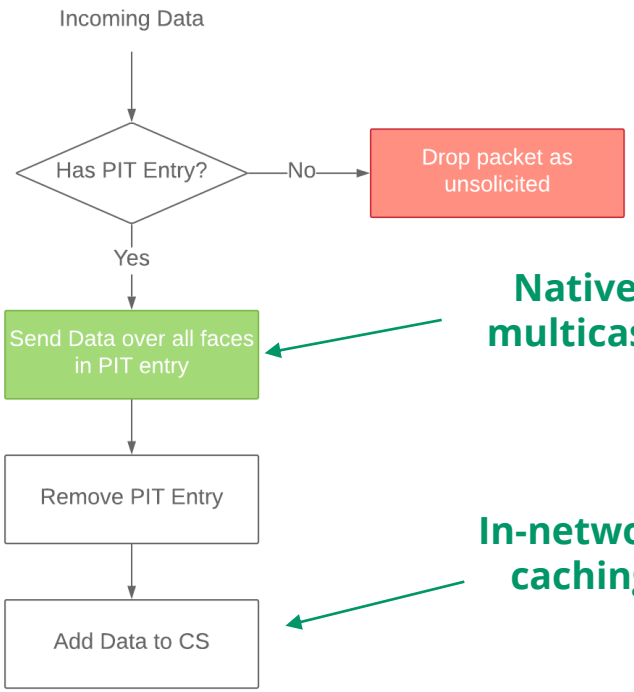
- Content Store (CS)
  - Cache of {Interests → Data} which has been recently obtained
- Pending Interests Table (PIT)
  - Stores outstanding Interests which have been forwarded upstream
  - Contains Name of Interest and list of downstream requesting faces
- Forwarding Information Base (FIB)
  - Similar to IP counterpart
  - Contains next-hop information for Interests

How does NDN work?



Interest aggregation

On Interest Received



Native multicast

In-network caching

On Data Received

Key Differences Between NDN and IP

NDN

- Packets routed by name only
- Stateful routing (FIB) and forwarding (PIT)
- Multipath forwarding
- Secures data at production time
- In-network caching

IP

- Packets routed by destination IP
- Stateful routing only
- Spanning tree forwarding
- Secures communication channel (TLS)
- No in-network caching

NDN Projects

Core software

- NDN Forwarding Daemon (**NFD***)
- The NDN Common Client Libraries (e.g. **jNDN***, pyNDN2, ndn-cxx)

Routing protocol

- NDN Link State Routing Protocol (**NLSR***)

Dataset synchronization

- CCNx Sync
- **ChronoSync***
- RoundSync, pSync

Real time applications

- Voice over CCN (VoCCN)
- Video Conferencing (NDN RTC)

* used in project

MOG Architectures

Client Server (C/S)

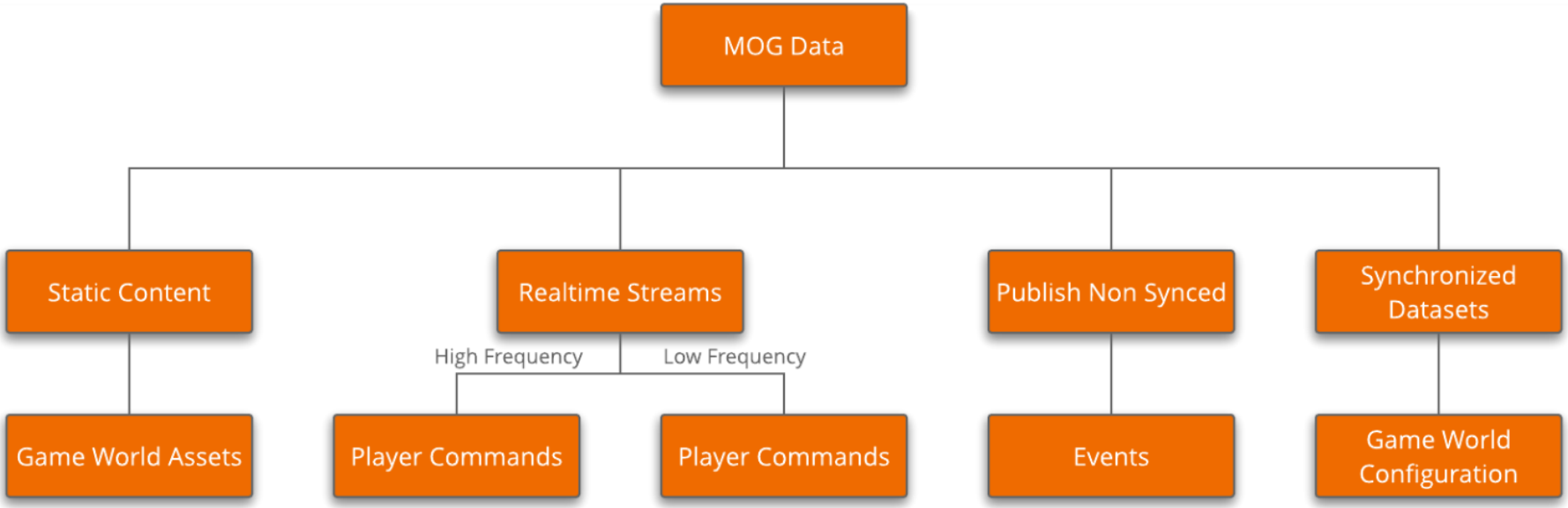
- One, or small number of, centralized servers
- Authoritative source of game state (handles all updates)
- Must send updates to all interested players (expensive)

Application Level P2P

- Distribute workload amongst players
- Peers inform each other of state changes
- Scalable, hard to maintain consistency, easier to cheat.

Game state must ***appear*** consistent to all connected players

MOG Data Taxonomy



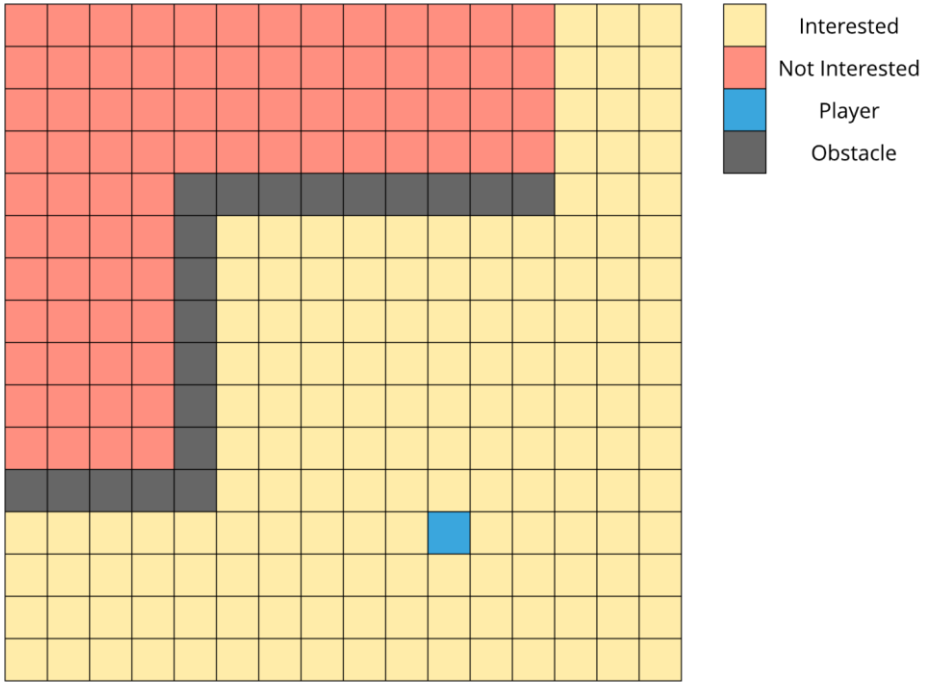
Dead Reckoning and Client Side Extrapolation

- Typical frame rates are 30 or 60 FPS
- Moving remote players based on updates alone is not feasible!

- Remote updates include velocities (and sometimes accelerations!)
- Clients update their copies of remote objects
  - Use dynamics of the remote object to guess at their position between updates
- Dead reckoning convergence mechanism
  - Smoothly transition between extrapolated position and updated position
  - Must be done while continuing to extrapolate to future positions

Area of Interest Management

- Players only interested in sub region of game world
- Simple approach
  - Distance based
- More complex
  - Take game world into account
  - Precompute interest tiles for every tile a player can visit



Tile based interest filtering

Benefits of NDN in a MOG Context

- Interest aggregation at intermediate routers
  - Especially useful in P2P architecture
  - Requires n^2 connections in IP
  - Number of connections dependent on topology in NDN
- Native multicast
  - Publishers (game players) need only produce a piece of data once
  - Data is disseminated by NDN routers
- In-network caching
  - Peers can benefit from common data sent as a result of another peer's interest

Closely Related Projects

- Egal Car: A Peer-to-Peer Car Racing Game, Zening Qu, Jeff Burke
  - Uses CCNx Sync to synchronize car positions
  - No support for interaction between game objects
- Matryoshka: Design of NDN Multiplayer Online Game, Zhehao Wang, Zening Qu and Jeff Burke
  - Very limited information (only 2 page paper)
  - No testing or evaluation
- NDNGame: A NDN-based Architecture for Online Games, Diego Barros, Marcial Fernandez
  - Vague outline of general design
  - No actual implementation, no results of simulations outlined

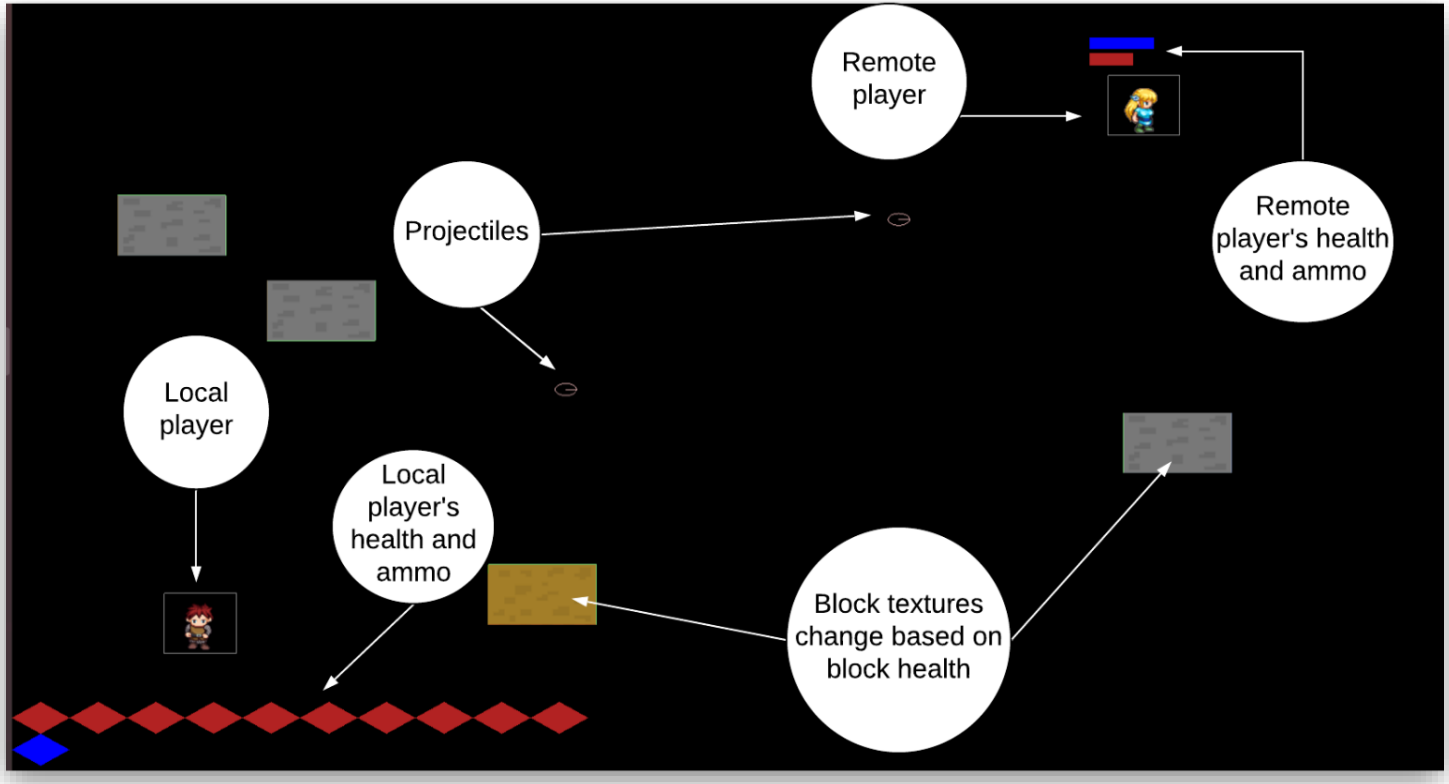
Research Questions

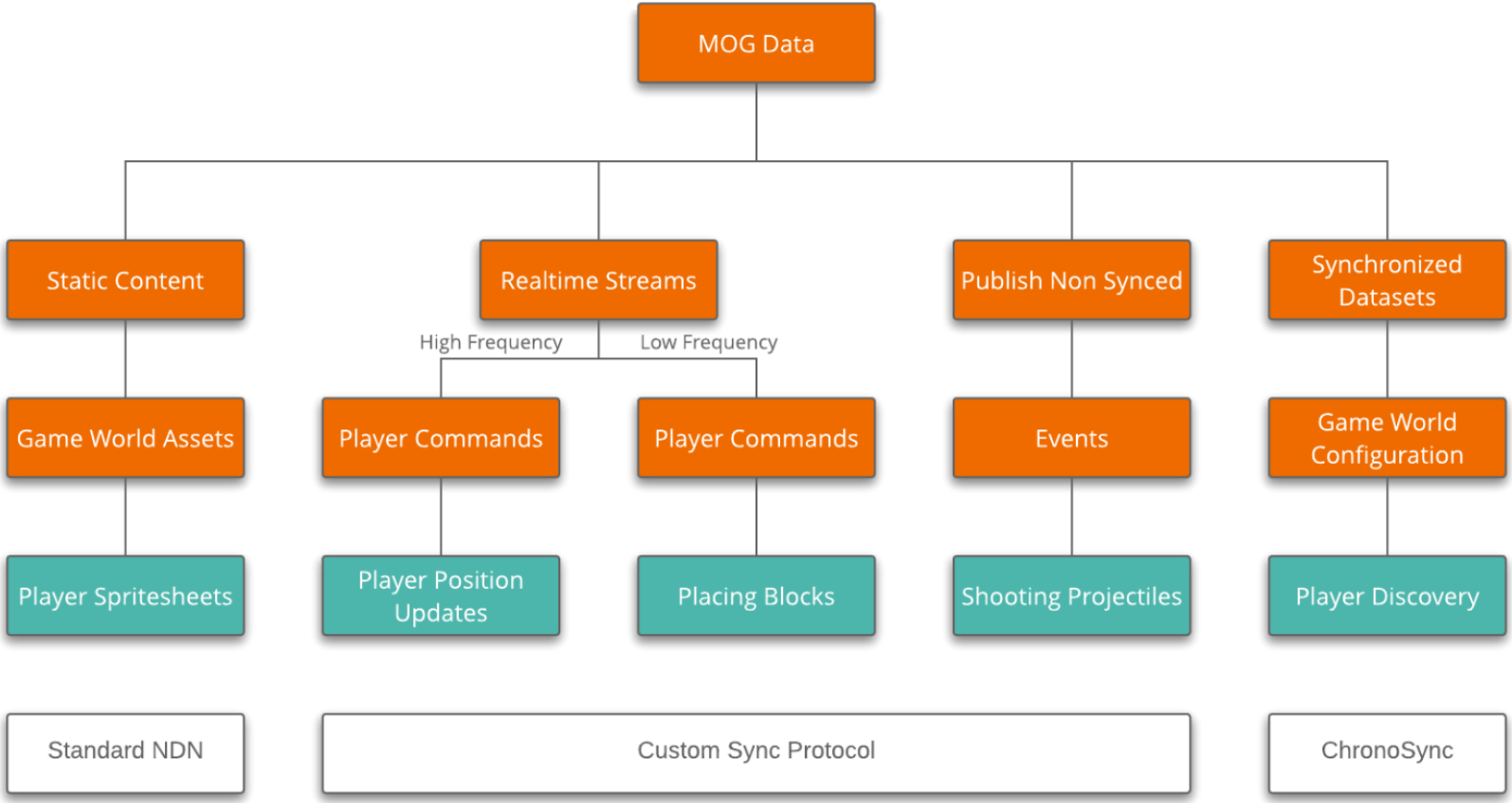
How can the benefits of NDN be exploited in a MOG context?

How can the different data types in MOGs be synced over NDN?

Design

A Basic 2D Game




MOG Data
Static Content
Realtime Streams
Publish Non Synced
Synchronized Datasets
High Frequency
Low Frequency
Game World Assets
Player Commands
Player Commands
Events
Game World Configuration
Player Spritesheets
Player Position Updates
Placing Blocks
Shooting Projectiles
Player Discovery
Standard NDN
Custom Sync Protocol
ChronoSync

Custom Sync Protocol: Motivation

Limitations of existing protocols

- Require **two round trips** (one for sync data, one for actual data)
- Write access to dataset is synchronous
  - Concurrent writes cause major problems and require reconciliation

Key Insights

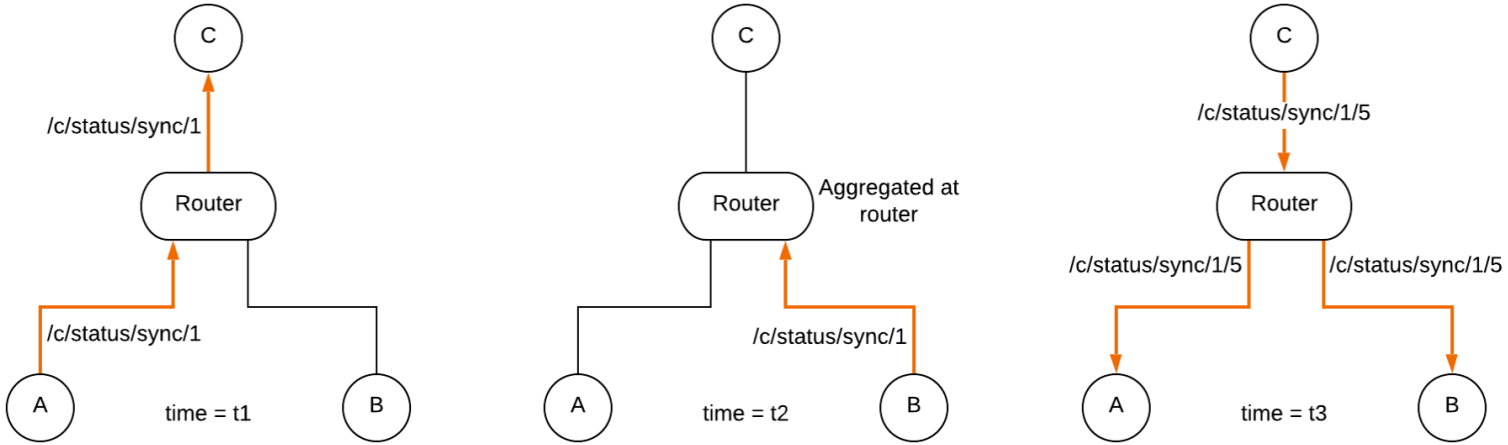
- MOG data is not inherently sequential
- Publishers need only publish the **newest** instance of their data

Custom Sync Protocol: Sync

- Interest Expression Name for Sync
  - `<game_prefix>/<game_id>/<player_name>/<type>/sync/<sn>`
  - **`/ndngame/0/player1/status/sync/1`**
- Data Reply Name
  - `<game_prefix>/<game_id>/<player_name>/<type>/sync/<sn>/<next_sn>`
  - **`/ndngame/0/player1/status/sync/1/50`**
- Key points
  - Publisher responds with **Data[49] not** Data[1]
  - Subscribers obtains next sequence number and will express for Data[50] next
  - Natural catch up mechanism
  - Exploits `CanBePrefix` field of NDN Interest packets

Custom Sync Protocol: Benefits

- Publishers only produce Data when there are **updates**
- Interest are **aggregated** at intermediate routers
- Data produced is **multicasted** back to subscribers

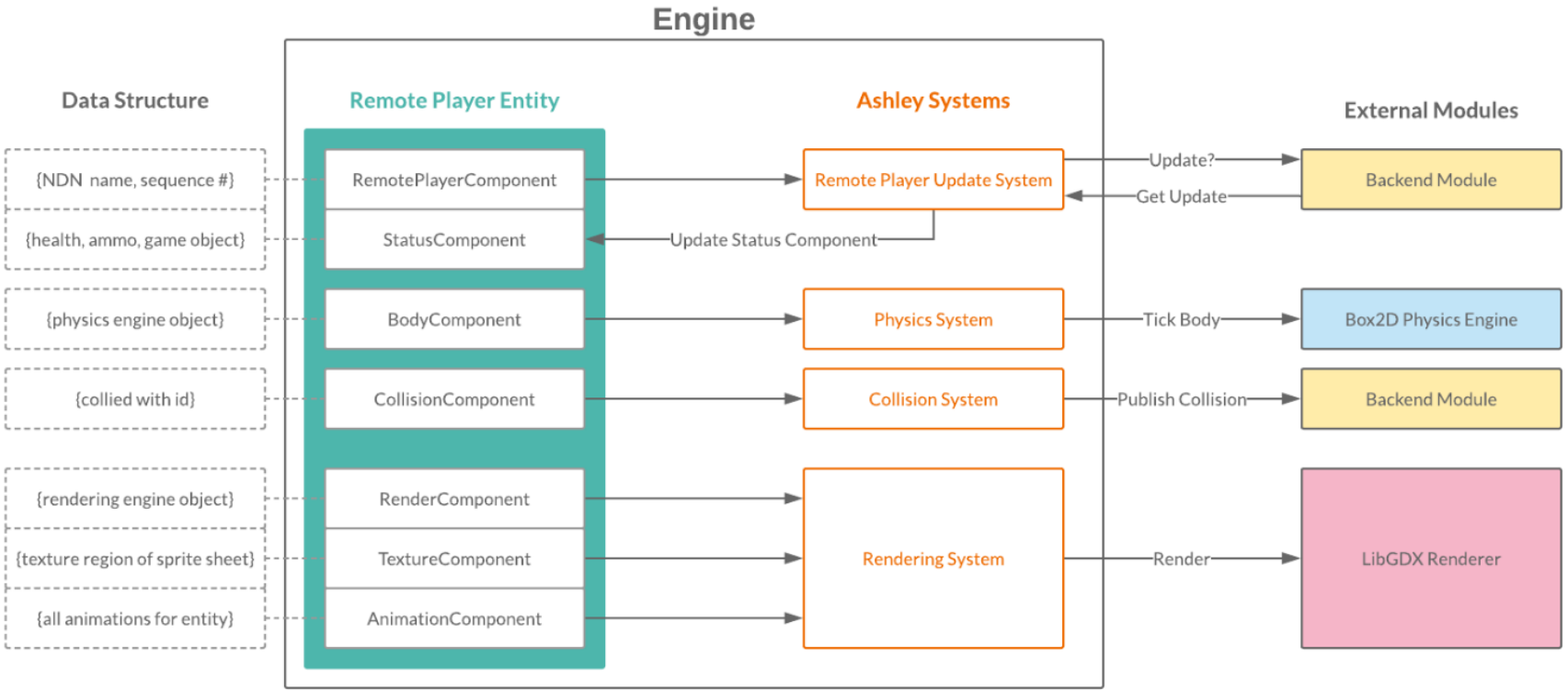


Implementation

Front End Implementation

- Built in Java using LibGDX
  - Java wrapper for OpenGL
  - Produces cross platform executables
  - Supports *headless* mode
- Uses Entity Management System (EMS) called Ashley
  - Components: Basic data structures
  - Entities: Contain a list of components
  - Systems: Performs operations on entities which contain certain components
  - Engines: Stores entities, invokes systems

Entity Representation for Remote Player



Backend Implementation

- Built in Java 8
  - NDN primitives using jNDN, NDN client library for Java
  - Google's Guice for Dependency Injection
  - Google's Protocol Buffers (Protobuf) for high performance serialization

- Backend is **entirely decoupled** from front end
  - Allows for plugging in of different backend modules
  - Build an IP based backend for direct comparison (future work)

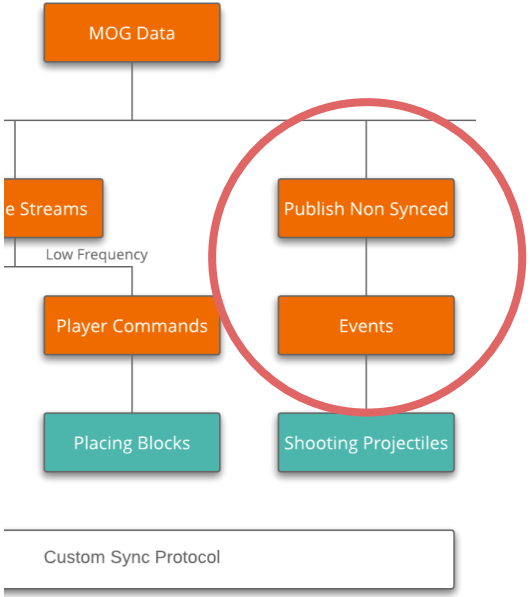
Optimizations: Publisher Side Update Throttling

- Subscribers use extrapolation
  - Publishers can slow down their publish rates
- Publishers use dead reckoning
  - Keep a small cache `{nextSN → {PlayerStatus, Timestamp}}`
  - Represents data we sent when we specified the next sequence # to be `nextSN`
- Each subscriber has an outstanding sync interest containing a SN
- Engine publishes update if:
  1. Has no cache entry for `nextSN`
  2. Local player's velocity has changed*
  3. The **estimate** of the subscriber's position for the local player exceeds a threshold value

* The current implementation's physics engine does not use accelerations meaning changes in velocity are changes in direction

Sequence Numbered Cache

- Use case
  - Want to send all events that subscriber has not yet seen
  - **NDN has no information about source of Interest!**
  - But the application has sequence numbers
- Essentially want a multimap like data structure
  - Map of {SN → [new events since that SN]}
  - This is a nice API but would be a very inefficient implementation
- Array based, circular data structure



Testing

Evaluation

Metrics

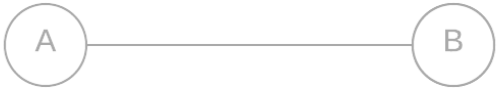
- Round trip time
- Network traffic
  - Subscriber Interest rates
  - Publisher Interest rates
- PlayerStatus deltas
- Scalability (# nodes)

Parameters

- Caching
  - `FreshnessPeriod*`
- Topology
- Forwarding strategy
- Publisher throttling
- Area of Interest Management
- Game Mechanics

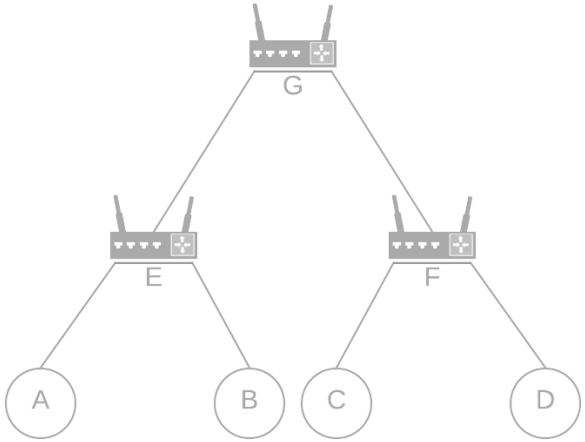
* Defines hop-by-hop content store lifetime - **not total packet lifetime**

Topologies

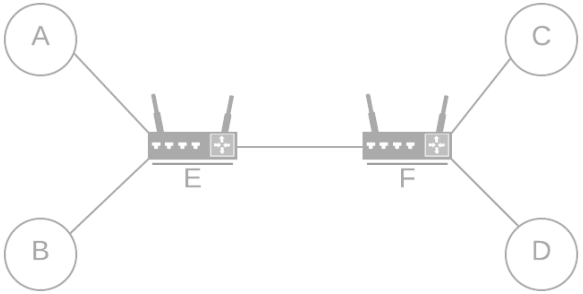


2 Game Nodes - Linear

4 Game Nodes - Square



4 Game Nodes - Tree



4 Game Nodes - Dumbbell

Docker

- Basic Docker image containing NFD, NLSR, game and startup script
- Docker overlay network to connect all docker nodes together
  - Allows creation of UDP tunnels between NFDs running on the nodes
- Deploy locally using `docker-compose`
- Deploy across cluster using Docker swarms and stacks

- Python scripts
  - Define topologies in code
  - Builds NLSR config file for each node - **this defines the NDN topology**
  - Builds docker compose file for given topology

/NLSR/square/nodeA-nlsr.conf

```
general {
  network /com
  site /stefanolupo
  router /%C1.Router/routerA
}
; Adjacent routers
neighbors {
  neighbor  {
    name com/stefanolupo/%C1.Router/routerB
    face-uri  udp4://nodeb.ndngame.com
  }
  neighbor  {
    name com/stefanolupo/%C1.Router/routerD
    face-uri  udp4://noded.ndngame.com
  }
}
; Prefixes we are responsible for
advertising {
   prefix /ndngame/0/discovery/nodeA
   prefix /ndngame/0/nodeA/status
   ;... etc
}
```

topoplogy-builder.py

```
# Build square topology
nodeA = Node("A")
nodeB = Node("B")
nodeC = Node("C")
nodeD = Node("D")

linkNodes(nodeA, nodeB)
linkNodes(nodeB, nodeC)
linkNodes(nodeC, nodeD)
linkNodes(nodeD, nodeA)

topology = "square"
nodes = [nodeA,nodeB,nodeC,nodeD]

NlsrBuilder(topology)
   .buildNlsrFiles(nodes)

ComposeBuilder(topology)
   .buildComposeFile(nodes)
```

docker-compose.yml

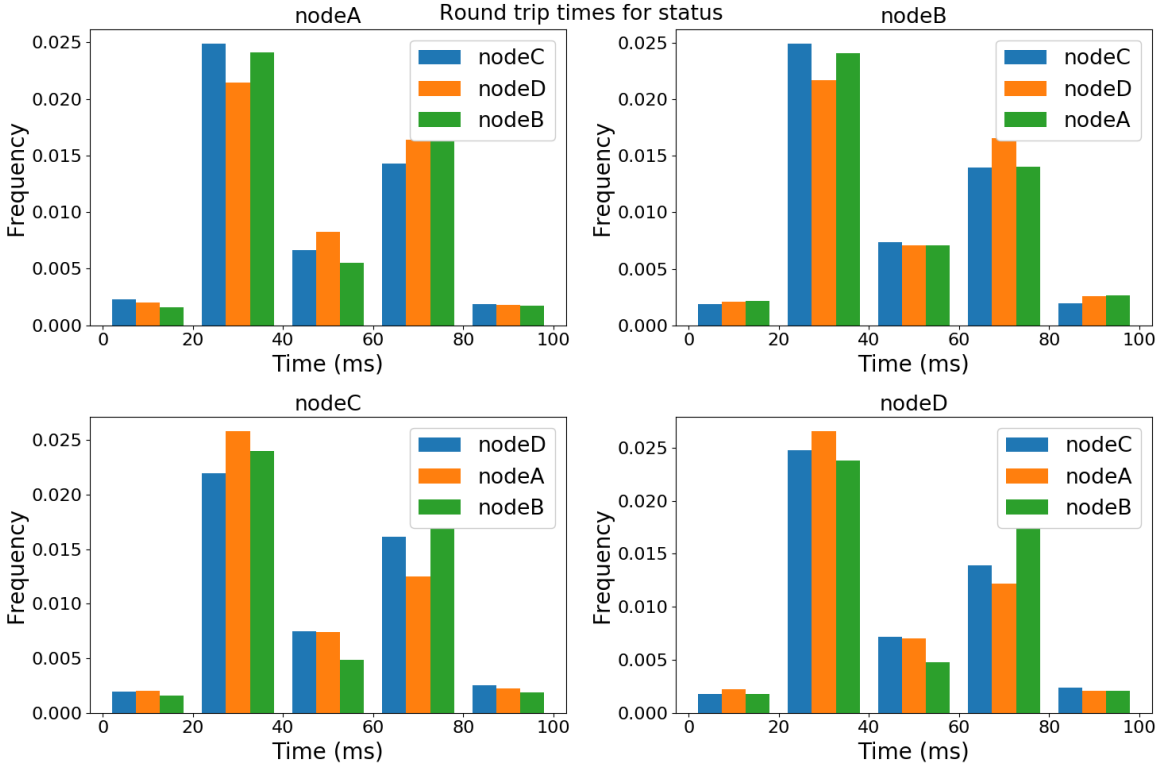
```
A:
 image: stefanolupo/ndn:ndn-node
 environment:
 - NLSR_CONFIG=/NLSR/square/nodeA-nlsr.conf
 - GAME=java -jar game.jar -a -hl nodeA
 - METRICS_DIR=/metrics
 hostname: nodeA
 networks:
   private-ndn-overlay:
    aliases:
    - nodea.ndngame.com


# Similar for B, C and D

networks:
 private-ndn-overlay:
   driver: overlay
```

Results

Round Trip Times

- ~30ms update rate
- Dumbbell topology
- No caching
- No publish rate tricks

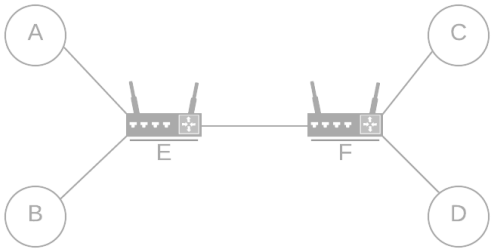
RTT are hugely dependent on publisher update rates



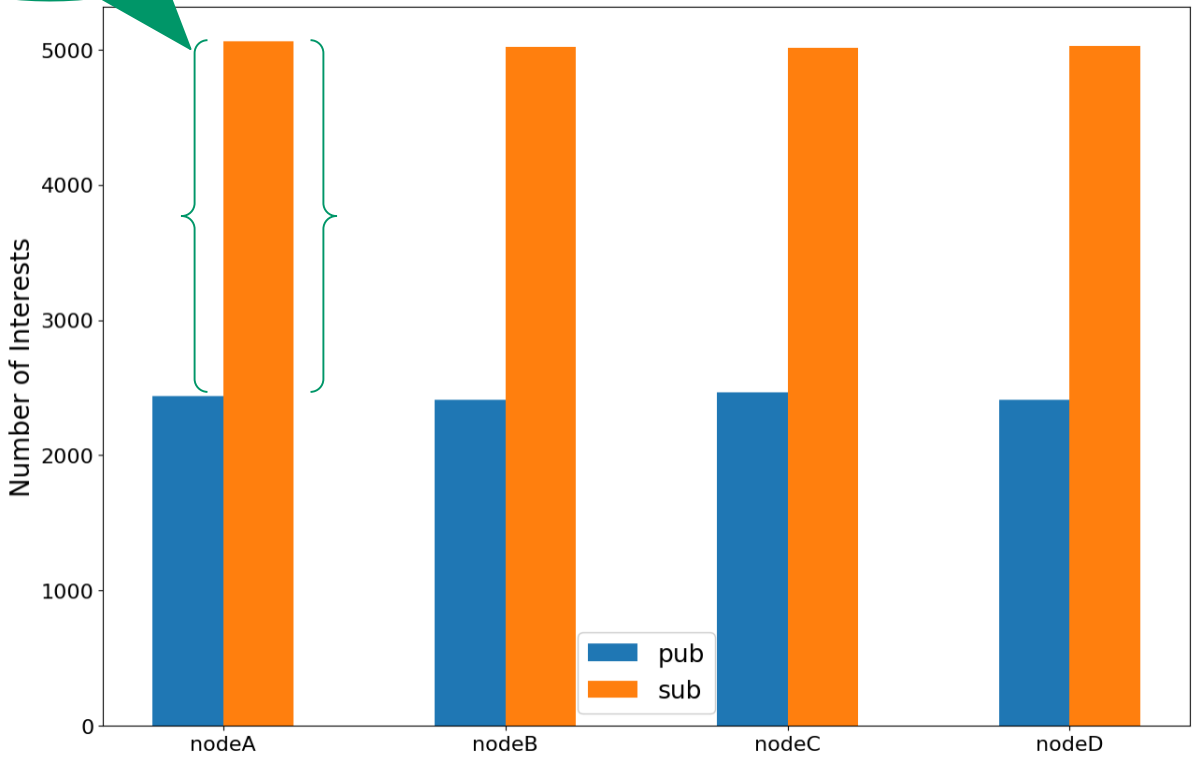
Interest Aggregation

Packets would be required in IP

- # Interests **towards** publisher
- # Interests **seen** by publisher

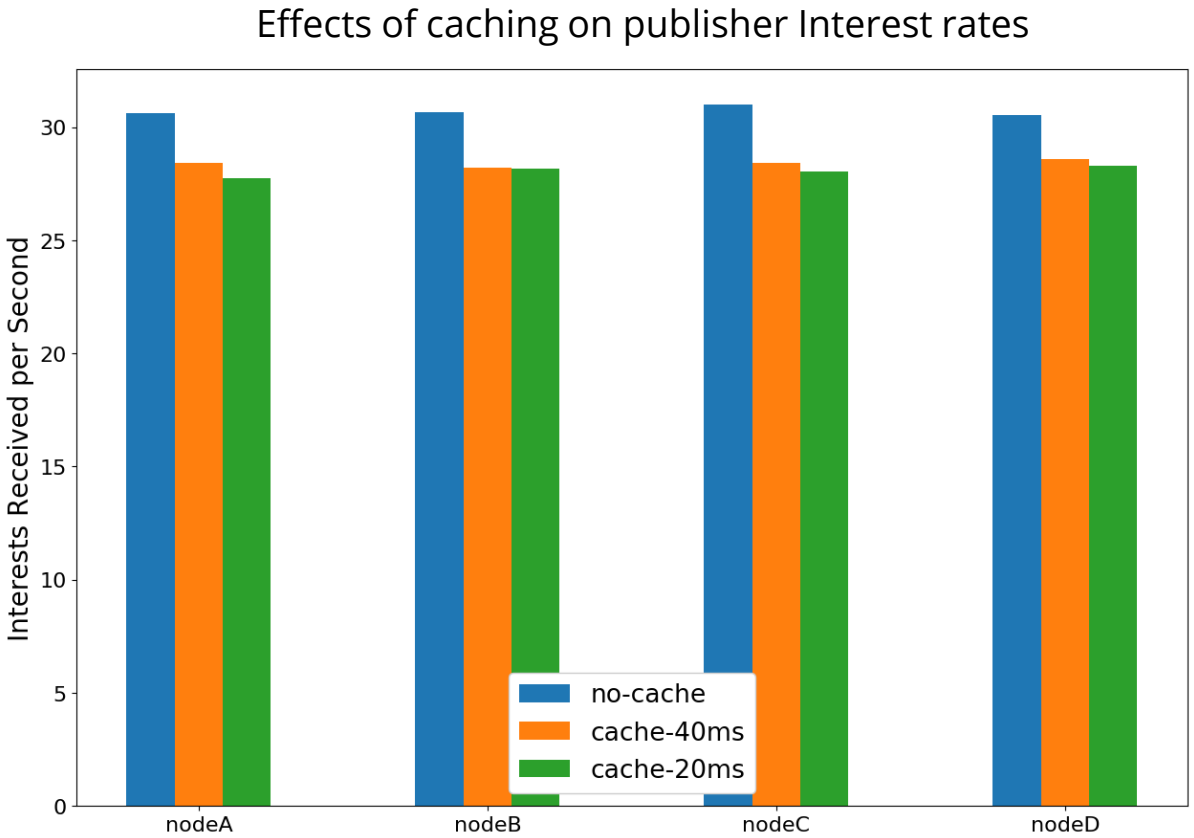


Effects of Interest aggregation on publisher Interest rates



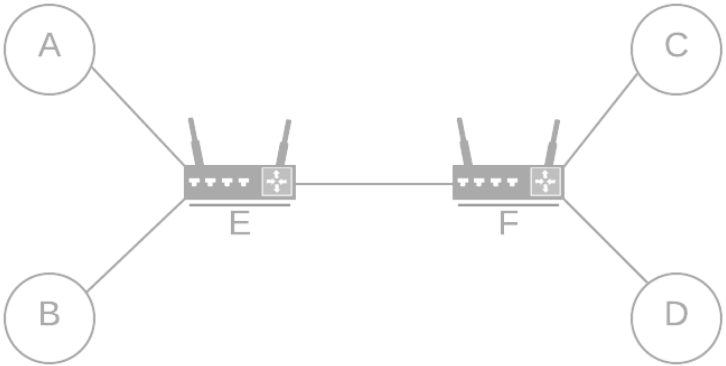
Effects of Caching

- Caching helps during bad periods
- Node can fall slightly behind
- Interest will have already been satisfied
- Receives cached copy

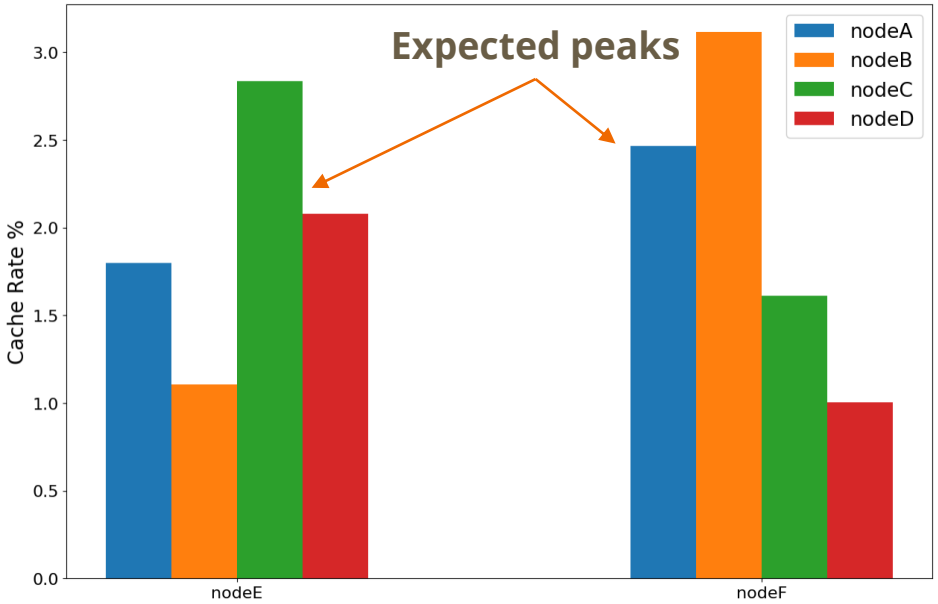


Where caching occurs

- Caching occurs at **intermediate routers** (in dumbbell topology)
- Cache entries **live longer** at intermediate routers
  - Hop by hop `freshnessPeriod`

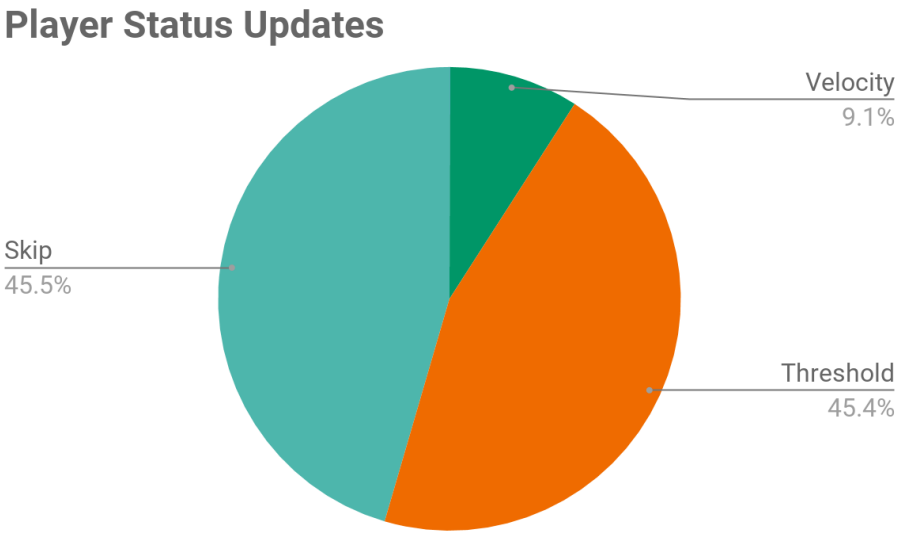


Router cache rates by node



Publisher Update Throttling

- ~0% cache miss with cache size of 50
- ~45% of updates skippable
- Tolerance of 0.5 GWU



Conclusion

Conclusions

- NDN can be very beneficial in P2P settings
  - Interest aggregation and native multicast
  - Caching to a smaller degree
- NDN is still in a proof of concept phase
  - Lots of hidden implementation details making protocol development tricky
- P2P games are much easier to scale
  - However they are very open to cheating

Thank you!