Towards a Unified Theory of Replication

Nalini Belaramani, Mike Dahlin, Lei Gao, Prince Mahajan, Amol Nayate, Jiandan Zheng
Department of Computer Sciences
University of Texas at Austin

December 4, 2007 Department of Computer Sciences, UT Austin

Why a Unified Theory of Replication?

(1) Better way to build replication systems

(2) Way to build better replication systems

Better Way to Build Replication Systems

Replication is a key building block
• Personal FS, enterprise FS, edge servers, distributed database, Grid FS, ...
• New applications, environments, workloads, technologies require new trade-offs
  • E.g., 14 OSDI/SOSP papers in last 10 years
Fundamental challenge
• There is no perfect system
  • CAP dilemma [Brewer01, Siegel92]
  • PC dilemma [Lipton and Sandberg 1988]
Problem
• Existing systems entangle policy with mechanism

Better Way to Build Replication Systems

Cleanly separate mechanism and policy
Integrate disparate theories/protocols

Policy
• Quorums, client-server, server replication, p2p, ...
  • E.g., Why can't I combine of Coda and Bayou?
Reduce costs to design/deploy new systems
• "Replication microkernel"
  Challenge: Implement each of the 14 OSDI/SOSP papers or a replication system for a new environment in <1000 LOC.

Outline

Case for a unified theory of replication
Mechanism
• PRACTI
• Evaluation
Policy
Conclusions
Challenge: PRACTI Replication

- Arbitrary consistency: Provides guarantees as prepared by application; don’t pay for unused guarantees.
- Partial replication: API may be exposed.
- Server replication: Any node can communicate with any other node.
- Topology independence: No assumptions about network topology.
- Object replication: Replicate any subset of data to any node.

Implementing PRACTI Replication

Core stores local state (mechanism):
- Reads request from checkpoint (or remote)
- Writes to log and checkpoint (and propagate)

Controller triggers communication (policy):
- Any node can communicate with any other node.

PRACTI Architecture

3 key ideas:
- Peer-to-peer data exchange
- Separate data vs. metadata paths
- Conservative summaries of unneeded invalidations

P2P Log Exchange [Petersen97]

Updates to log:
- Local checkpoint for random access
- Log exchange for updates:
  - TI: Pairwise exchange with any peer
  - AC: Prefix property, causal consistency, eventual consistency
  - PR: All nodes store all data, see all updates

Naive Addition of PR to Log Exchange

Full replication assumption deeply embedded in server replication (AC-TI) protocols:
- Naive addition of PR to protocol violates AC
- Hierarchy topology assumption deeply embedded in client/server (PR-AC) protocols
- Lack of consistency key to flexibility of object replication (PR-TI) protocols

Separate Data and Invalidation

Log exchange:
- Ordered streams of metadata (invalidations)
- All nodes see all invalidations (logically)
- Checkpoints track which objects are VALID

Send any bodies to any nodes at any times.
Issue: Reading Bodies

Mechanism: Block until data VALID
- VALID = body matches latest invalidation
  - Prefix property, causal consistency, eventual consistency
Policy: Controller’s choice
- Demand read miss
  - Target is policy choice: client/server, DHT directory, original writer, random, ...
  - Prefetch

Node A
Node C
Node B

Read bur
Prepush bur
Read bur

Imprecise Invalidations

- Partial replication of metadata
  - For objects you care about
    - Subscribe for precise invalidations
  - For the rest
    - Subscribe for imprecise invalidations
  - Imprecise invalidations act as “placeholders”
    - In log and checkpoint

Processing Imprecise Invalidations

Read /a/q
II(/a/*, 101, 103)
I(/a/b, 101)
I(/a/c, 103)

Splitting Invalidations/Bodies Helps

Step towards partial replication
- Nodes only see bodies of interest
  - Order of magnitude improvement!
- Nodes still see all invalidations
  - Limits scalability
    - E.g., Enterprise file system
      - Every departmental server sees all departments’ updates
      - Every palmtop sees every update by any node

Additional Details (See paper)

Efficient, continuous update exchange
- Incremental log exchange
Garbage collect logs
- Incremental checkpoint exchange
  - Use imprecise invalidation’s bookkeeping
Self-tuning replication
- Prefetch/pre-push bodies over low-priority network channel (TCP-Nice)
Continuous consistency (e.g., TACT)
- Pay for just the guarantees you need
Enforce minimum replication for availability
- Bound invalidations

Summary of Mechanism

Partial replication of bodies
- Separate invalidations and bodies
- Any node can receive any body at any time
- Local rules for coordinating incoming streams
Partial replication of invalidations
- Log exchange for consistency, prefix property
- Imprecise invalidations summarize missing info
- Careful bookkeeping to track, recover missing info
→ PRACTI!
Policy choices
- Where to place bodies (PR)
- What consistency to enforce (AC)
- With whom to communicate (TI)

Efficient invalidation exchange
→ Order of magnitude improvement!
Outline

Case for a unified theory of replication
Mechanism
- PRACTI
- Evaluation
Policy
Conclusions

Methodology

Prototype benchmarking
- Prototype: Java + BerkeleyDB

How to evaluate “Unified theory”?
- Compare with
  - AC-TI server replication (e.g., Bayou, TACT)
  - PR-AC client-server (e.g., Coda, NFS)
  - PR-TI object replication (e.g., Ficus, Pangea)
- Key questions
  - Can system match their performance?
  - Does system provide significant advantages?
  - What extra cost do you pay for flexibility?

Partial Replication: PRACTI v. Server Repl

Order of magnitude improvements
- Both separate inval v. body AND imprecise inval
- Storage requirements see similar improvements

Topology Independence: PRACTI v. Client/server

Synchronization Time

- Client-server (e.g., Coda)
  - Limited by network to server - Not an attractive solution
- Server replication (e.g., Bayou)
  - Limited by fraction of shared data - Not a feasible solution
- PRACTI:
  - Up to order of magnitude better - Does what you want!

What is the cost of flexibility?

PRACTI subsumes server replication, client-server, and object replication

What costs do you pay?
- Server replication (e.g., Bayou)
  - None?
- Object replication (e.g., Ficus)
  - Explicit encoding of “missing” invalidations
    - v. weaker consistency semantics
- Client-server (e.g., Coda)
  - Explicit encoding of “missing” invalidations
    - v. implicit in client-server contract
Cost of Consistency

Read latency, availability
- Tunable consistency (Yu and Vahdat)
- Reads block until consistency constraints satisfied
- Pay for what you need
- No additional read latency or availability overhead

Write bandwidth
- Distribute enough information to order writes
- When needed by future reads
- Imprecise invalidations
  - Transmit full information to support anticipated reads
  - Transmits summaries for the rest
    - Placeholders for future, unanticipated reads
    - Placeholders for other nodes that receive writes from my log
  - How expensive are the imprecise invalidations?

Mechanism Summary

PRACTI subsumes existing approaches
- Client-server/hierarchy
- Server replication
- Object replication
Additional flexibility improves performance v. consistency trade-offs

Imprecise invalidations save v. all-precise
Imprecise invalidations cost v. coherence only
- Worst case 2:1 (messages)
- Locality reduces cost

Outline

Case for a unified theory of replication Mechanism
- PRACTI
- Evaluation
Policy
Conclusions

"Better Way"?

Separate Mechanism v. Policy
Goal: Common core mechanism
- "Replication microkernel"
- Vision:
  - Implement replication system for new environment in <1000 lines of policy code
  - Queue sound of scared graduate students running for the door

Two Assertions

(1) PRACTI
Async mes and declarative routing
sound mechanisms
storage nirvana?
Core v. Controller

**Core: Mechanism**
- Safety: Any message can be processed at any time
- Asynchronous message passing style
- Requests are only hints

**Controller: Policy**
- Liveness: Trigger messages between nodes

---

Example: Read Miss in Client-Server Controller

It seems like we have the right mechanisms
- Students still worried when I say “Build 14 systems…”

---

PRACTI Mechanisms → Policy ~ Topology

1. From whom should a node subscribe to receive invalidations?
   - Which precise and which precise?
   - Which invalidations from which partner?
   **Bayou**: Subscribe to * from all neighbors

2. From whom should a node subscribe to prefetch bodies?
   **Bayou**: Subscribe to * from all neighbors

3. Where should a node send a demand read miss request?
   **Bayou**: N/A

---

Overlog

A declarative language for specifying overlay routing based on datalog [Loo et al.]

A relational data model
- Tables
- Rules
  - Fired by combination of tuples
  - At the end of a rule, one tuple is generated.

**Basic Syntax for Rules**

<Action> :- <Condition1><Condition2>…<ConditionN>

---

PRACTI and Overlog

Overlog to represent policy
- PRACTI generates events for P2
- “Output tables” trigger PRACTI actions from P2
  - E.g., SubscribeUpdatedFrom@X(X, Partner, Volume)

**Bayou: Server-Server Anti-Entropy**
- A Server will retrieve updates when it is connected to another server
- Inputs:
  - Neighbor@X(X, S) Managed by overlay network
  - WantVolume@X(X, V) The volumes I am interested in
- Rule:
  - Volume@X(X, V) :- WantVolume@X(X, V)
  - SubscribeUpdatedFrom@X(X, S, V) :- Neighbor@X(X, S), Volume@X(X, V)

---

Coda: Client/Server Callbacks

Coda: Client Server Callbacks
- Inputs:
  - Server@X(Me, S)
  - LocalReadMiss@X(Me, File ID)
  - AddHoard@X(Me, File ID)
- Rules:
  - HoardList@X(X, Fid) :- AddHoard@X(X, Fid)
  - SubscribeUpdatedFrom@X(X, S, Fid) :- HoardList@X(X, Fid), Server@X(X, S)
  - DemandRead@X(X, Fid, S) :-
    - LocalReadMiss@X(X, Fid), Server@X(X, S)
  - SubscribeInvalFrom@X(X, S, Fid) :- LocalReadMiss@X(X, Fid), Server@X(X, S)
“Better way”?

Grand challenge/eventual goal
• Build 14 SOSP/OSDI systems in <1000 LOC
• Build novel systems with useful properties

Status
• Several controllers built
  • Bayou, simplified client-server,
    PLFS for WAN experiments
• Several others in progress
  • Enterprise FS, Personal FS,
    Edge server for replicated TPC-W
• P2 controller framework in progress

Experience:
• Positive – easy to build wide range of systems
• Declarative “replication policy as routing” promising

Case Study: PlanetLabFS

Goal: Simplify running experiments
• Track current locations of files via DHT
• Cooperative caching
  • Flood initial data, programs from server to clients
  • Direct transfer of updates among clients
• Future: Self-tuning prefetching

Experience
• SDIMS + PRACTI → Easy to build this
  • <2 weeks
  • Caveat: Not a “real” system yet

PLFS Evaluation

Benchmark
• Phase 1 Disseminate:
  • Disseminate 10MB from server to all clients
• Phase 2 Process:
  • 10x pairwise exchange 1MB between random clients
• Phase 3 Post-Process:
  • Gather 1MB from each client to server

PlanetLabFS

• 3x-5x v. client-server (dissemination)
• 2.4x-9x v. server replication (process, post-process)
• 1.5x v. cooperative caching (process)
• TBD: Add self-tuning prefetching

Outline

Case for a unified theory of replication
Mechanism
• PRACTI
• Evaluation
Policy
Conclusions
• Towards a unified theory and practice

PRACTI Design Summary

Result: Subsume many existing mechanisms
• Client/server: Coda, NFS, AFS, ...
• Server replication: Bayou, TACT
• Object replication: Ficus, Pangea, ...

Key ideas
(1) Separate data from metadata
  • Separate streams for invalidations and bodies
  • Key idea: Synchronize these streams
(2) Summarize unneeded metadata
  • Imprecise invalidations
    • Key idea: Track “precise” and “imprecise” data
(3) Separate mechanism from policy
  • Core: PRACTI mechanisms, Controller: Policy
  • Key idea: Async message passing style
**Evaluation Summary**

Way to build better file systems
- Partial replication of data (10x)
- Partial replication of metadata (10x)
- Topology independence (10x)
- Minimal consistency cost
- Additional benefits (see paper)
  - Self-tuning replication of bodies
  - Incremental checkpoint transfer

Better way to build file system
- Preliminary experience is positive

---

**Towards" a Unified Theory**

- Improve productivity?
  - Does PRACTI reduce development costs by 10x?
  - Can we support 14 OSDI/SOSP papers in <1000 LOC each?
- Unify the literature?
  - Can we support quorums, client-server, server replication, p2p, ... on the same substrate?
  - How do various consistency paradigms relate?
    - FIFO, causal, sequential, linearizable, etc.
    - Reads follow writes, monotonic reads, etc.
    - Safe, regular, atomic, etc.
- What are the "core mechanisms" for security?
- General API?
  - Can we support FS, tuple store, and DB on same substrate?
  - Other environments?
  - Can we unify other "large scale" replication systems (e.g., cluster)?

---

**Conclusion**

Build your next large-scale replication system using PRACTI
- A better way to build replication systems
- A way to build better replication systems

Details on my web page


---

**Overlog Example**

**Tables**
- neighbor(NAddr, NId)
- successor(NAddr, SuccId, SAddr)

**Example Rule:**
```
response@Req(Q, K, SAddr) :-
lookup@NAddr(NAddr, Req, K)
node@NAddr(NAddr, NId)
successor@NAddr(NAddr, SuccId, SAddr)
K in (NId, SuccId)
```

The above rule tries to find the node responsible for a given K in a ring topology

**Pseudocode:**
```
n.lookup(k)
  if k in (n, n.successor):
    return n.successor.address
```

---

**Self-Tuning Updates**

---

**Overlog Example**

**Tables**
- node(NAddr, NId)
- successor(NAddr, SuccId, SAddr)

**Example Rule:**
```
response@Req(Q, K, SAddr) :-
lookup@NAddr(NAddr, Req, K)
node@NAddr(NAddr, NId)
successor@NAddr(NAddr, SuccId, SAddr)
K in (NId, SuccId)
```

The above rule tries to find the node responsible for a given K in a ring topology

**Pseudocode:**
```
n.lookup(k)
  if k in (n, n.successor):
    return n.successor.address
```
Issue: Synchronization of Separate Streams

Retrieved body may be newer than metadata
→ Violate causality
→ Solution: Buffer body until apply associated inval

Palmtop/Laptop Sync Time

Synchronize palmtop to laptop
• Client-server: Limited by network to server
• Bayou: Limited by fraction of shared data (1%)

Self-Tuning Prefetch

Prefetch aggressively when BW available
• Conservative when resources scarce

Towards Unified Replication III

Quorums
• Client-server a special case of quorums
• Server-replication a special case of quorums
• Quorum literature and terminology disjoint from Client-server, server-replication
• Status
  • Initial work: "Volume Quorums"
  • Future: Clean integration to "Unified Theory"
    - Some hooks in PRACTI, but not quite there...

Security
• Different environments make different trade-offs
• Is there a core set of mechanisms disjoint from policy?
• Draw on secure log exchange literature?

Real Motivation: Bayou v. Coda

<table>
<thead>
<tr>
<th></th>
<th>Coda</th>
<th>Bayou</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology</td>
<td>Client-server</td>
<td>Arbitrary</td>
</tr>
<tr>
<td>Disconnected operation</td>
<td>Local disk</td>
<td>Local disk</td>
</tr>
<tr>
<td>Disconnected updates</td>
<td>Send redo log</td>
<td>Send redo log</td>
</tr>
<tr>
<td>Disconnected reads</td>
<td>Hoard key data</td>
<td>Replicate all data</td>
</tr>
<tr>
<td>Consistency</td>
<td>Causal, eventual</td>
<td>Causal, eventual</td>
</tr>
<tr>
<td>Resolve conflicts</td>
<td>App-specific rules, manual</td>
<td>App-specific rules</td>
</tr>
</tbody>
</table>

Want combination
• Should be straightforward...
“Towards” a Unified Theory

Not there yet
• Today: PRACTI
• Unify large part of design space
  • Client-server (e.g., NFS, Coda, AFS)
  • Server replication (e.g., Bayou, TACT)
  • Object replication (e.g., Ficus, Pangea)
• Future work to incorporate
  • Quorums, general model of security,
    DHT-based P2P, content-keyed identifiers, ...

Practical benefits
• Simplify teaching
  • Does it?
• Simplify development and deployment
  • E.g., PLFS prototype in 2 weeks
• Improved trade-offs
  • Order of magnitude improvements available

Step 1: Separate Data and Metadata

Node A
- Node B
  - Node C
- Node D

Metadata: Log invalidations
Data: Store update bodies in checkpoint

Log exchange:
• Send invalidations separate from bodies
Synchronization of separate streams:
• Don’t apply body until apply associated inval

Example: Bayou Controller

Subscriptions
• Precise invalidations, bodies
  • For all n in neighbors subscribe to "/*" from n
No-Op Events
• Local read miss - Never occurs
• Inval/Body arrives - No action needed
• Cache replacement policy - Full replication
Housekeeping
• Garbage collect log if exceed threshold

2 Questions
1. Why are there so many file systems
   for large-scale systems?
   AFS [TOCS88]  Bayou [SOSP95, SOSP97]
   Coda [SOSP92]  Bullet [SOSP03]
   Rover [SOSP95]  LBFS [SOSP01]
   Weak Connectivity [SOSP95]  CFs [SOSP01]
   Blue [OSDI04]  Pangea [OSDI02]
   Iris [OSDI02]  SFS [OSDI00]
   USITS/NSDI, HPDC, ICDCS, Supercomputing, ...
2. Why are none of them what I want?
2 Questions
- Why are there so many file systems for large-scale systems?
- Why are none of them quite what I want?

Thesis of project
- Answer: Because current "point solution" systems entangle mechanism with policy
- Fix via unified replication architecture that captures fundamental mechanisms

Ideal: Unified Theory of Replication

Subsume
- Client/server
- Server replication
- P2P
- Quorums
- Object replication

Practical benefits
1) Simplify teaching
   • A few core concepts v. 20 case studies
   • Why are Coda and Bayou different systems?
2) Simplify development and deployment
   • Policy layer over core mechanisms
     - Replication "toolkit" or "microkernel"
   • Vision: Implement any system on slide 1 w/ <1000 LOC
3) Improved trade-offs
   • Continuum of options v. "point solutions"
   • Order of magnitude improvements available

Implementing Imprecise Invalidations

Imprecise invalidation
- Prevent (standard) reads to objSet
- Until missing invalidations received
- Bookkeeping: Each interest set maintains
  • \( pVV \) - last precise version vector
  • \( lpVV \) - currentVV
- Guarantee read consistency if \( lpVV = currentVV \)

Make \(/a/*\) precise
- \( \text{Update } currentVV \)
- Must receive all precise invalidations to \( /a/* \) from \( lpVV_{/a} \) to \( currentVV \)
- Rely on causal consistency of incoming streams
  → No gaps from \( \text{stream.prev} \) to \( \text{stream.next} \)

Imprecise Invalidations Summary

Imprecise invalidation
- Summarize multiple writes
- Placeholder in log
  - Ensures causal consistency of reads and writes
To process imprecise invalidation \( I \)
- Update \( currentVV \)
- Do not update \( lpVV \) for interest sets that overlap \( I \)
  • Or that are already imprecise
Read of imprecise interest set \( IS \) blocks
- Until \( IS \) is precise again
  • Receive all precise invalidations from \( IS /lpVV \) to \( currentVV \)
  • Use causal order of incoming streams
  • Ensure "no gaps" between incoming invalidations
  • Reason about when OK to update \( lpVV \)

Towards a Unified Theory and Practice

I. Universal mechanism
II. New policies
Towards Unified Replication I: Mechanism

PRACTI = "Replication Microkernel"?

- Subsume server replication
- Subsume object replication
- Almost subsume client/server
  - TBD: Adaptive callbacks, leases
- TBD: Quorums, universal security mechanisms, ...

**Goal:**
- Toolkit for building replication systems
- Each FS on slide 1 as <1000-line "policy layer"

Towards Unified Replication II: Policy

New Policies
- Given better mechanisms, can we build better policy trade-offs than past systems?
- Can we build a self-tuning "universal policy"?
  - Specify high level consistency, performance, availability, bandwidth goals/constraints

Case for a Unified Theory of Replication*

Current systems entangle mechanism with policy
- E.g., Coda v. Bayou
- 14 OSDI/SOSP papers in 10 years
  - New environment → new trade-offs → new mechanisms
  - Not clear new systems dominate old ones (or that 14 is "enough")

Current literature fragmented
- Client-server v. quorums v. server replication v. p2p v. ...
- E.g., Coda and Bayou each have separate server-replication and client-server caching protocols

**Impact**
- Systems narrowly tailored for specific environments
- Significant effort to develop system for new environment

* Scope: "Large scale" replication
  - WAN, mobile, enterprise, etc.
  - File systems, tuple stores, databases, distributed objects, …

Outline

- Case for a unified theory of replication
- PRACTI: A first step
- Evaluation
- Future directions

Vision: Replication Microkernel/Toolkit

Universal Policy

- Grand Challenges:
  - Each large-scale FS from OSDI/SOSP 1990-2005
    - <1000-line "policy layer"
  - "Universal policy" - self-tuning replication
    - Control replication to meet high level goals
      - e.g., "Minimize response time and maximize availability while providing causal consistency and less than 1 minute staleness to all replicas while using less than 2x demand-read traffic."

Cost of Consistency (PRACTI v. PR-TI)

Tunable consistency
- Causal, causal + TACT, sequential, linearizable
- Consistent or coherent
  - Consistency: Order writes across all objects
  - Coherence: Order writes to individual objects

PRACTI benefits
- Semantics specified on per-read, per-write basis
  - What information must a read or write wait for to complete?
    - No unnecessary read delay or write delay
  - Separation of invalidations from bodies
    - Minimize delay (hence inconsistency)
Improved Consistency Trade-Offs

<table>
<thead>
<tr>
<th>Bayou</th>
<th>TACT-Aggressive</th>
<th>PRACTI</th>
</tr>
</thead>
<tbody>
<tr>
<td>How</td>
<td>Batch</td>
<td>Incremental</td>
</tr>
<tr>
<td>When</td>
<td>Periodic</td>
<td>Frequent</td>
</tr>
<tr>
<td>Bodies</td>
<td>All</td>
<td>Continuous</td>
</tr>
</tbody>
</table>

- **Average Unavailability**
  - 0.0001
  - 0.001
  - 0.01
  - 0.1
  - 1

- **Available Bandwidth/Write Bandwidth**
  - 0
  - 0.5
  - 1
  - 1.5
  - 2

- **Periodic (500s)**
  - TACT-Aggressive
  - PRACTI-Demand
  - PRACTI-Prefetch

Cost of Consistency v. Coherence

Suppose I care about subset of data
- /A/* but not /B/*, /C/*, or /D/*

**PRACTI**
- Precise invalidations for /A/*
- Imprecise invalidations for the rest

Imprecise invalidations: “Placeholders”
- Allow future reads/writes to be consistently ordered with writes to /B/*, /C/*, /D/* if desired
- System that only guarantees coherence and never provides option of consistency could omit imprecise invalidations
- Worst case: Each precise invalidation paired with imprecise invalidation summarizing writes on which it depends
- How much overhead do these imprecise invalidations impose on nodes that don’t use them?

Controller Interface

- Notified of key events
  - Stream begin/end
  - Invalidation arrival
  - Body arrival
  - Local read miss
  - ...

- Directs communication among cores
  - Subscribe to inval or body stream
  - Request demand read body

- Local housekeeping
  - Log garbage collection
  - Cache replacement

Example: Client-Server Controller

- Subscriptions
  - Precise invalidations
    - For all f in <cached files> subscribe to f from server
  - Bodies
    - For all h in <hoard list> subscribe to h from server

- Local read miss on file f
  - if f is imprecise
    - request metadata + body from server
  - else /* f is precise but invalid */
    - request body from server
  - (read blocks until f is precise and valid)

- Point of interest perhaps only to me
  - Client/server crash recovery really natural/elegant

Example: EnterpriseFS Controller

- Support thousands of devices
  - Handful of big, geographically distributed servers
  - Many desktops, laptops, palmtops, etc.

- Read miss
  - Use DHT to find nearest copy of data

- Replication policy
  - DHT tracks file popularity
    - Self-tuning prefetch important updates to where they are/will be needed
    - Enforce minimum replication degree for reliability and availability

- Details TBD...

Step 2: Imprecise Invalidations

- Nodes subscribe for
  - Precise invalidations for interest sets
  - Imprecise invalidations for other data

- Precise invalidation
  - Metadata for one write
    - <object ID, accept stamp>

- Imprecise invalidation
  - Summary of multiple writes
    - <objectSet, [start]*, [end]*>
    - “One or more objects in objectSet were modified between start and end”
Processing an Imprecise Invalidation

**Update log**
- “Placeholder” for missing information

**Update checkpoint**
- Treat everything in `objectSet` as invalid

**Bookkeeping details (see paper)**
- Group data into `Interest Sets`
- Efficiently track which `Interest Sets` are missing invalidations ("imprecise")
- Block reads to `imprecise` `Interest Sets`
- Make interest set `precise` when missing invalidations applied

Architecture

- **Policy**
- **Application**
- **P2 System**
- **Runtime**
- **Local Interface**
- **Core**

Policy:
- Overlog rules to make routing decisions
- P2 System
- Interprets the overlog rules and sets up data flows

Runtime:
- Middleman between Practi Controller and overlog
- Carries out actions specified by the policy and informs the policy of events

Local Interface:
- Enforces consistency semantics