Irmin~~(sule)~~

Amir Chaudhry, Jon Crowcroft, **Thomas Gazagnaire**
Anil Madhavapeddy, Richard Mortier[1]
David Scott[2], David Sheets and Gregory Tsipenyuk,

University of Cambridge, University of Nottingham[1], Citrix Systems[2]

SRG Seminar
Computer Lab, Cambridge
22/05/2014

Summary

Context: The Stack Contest

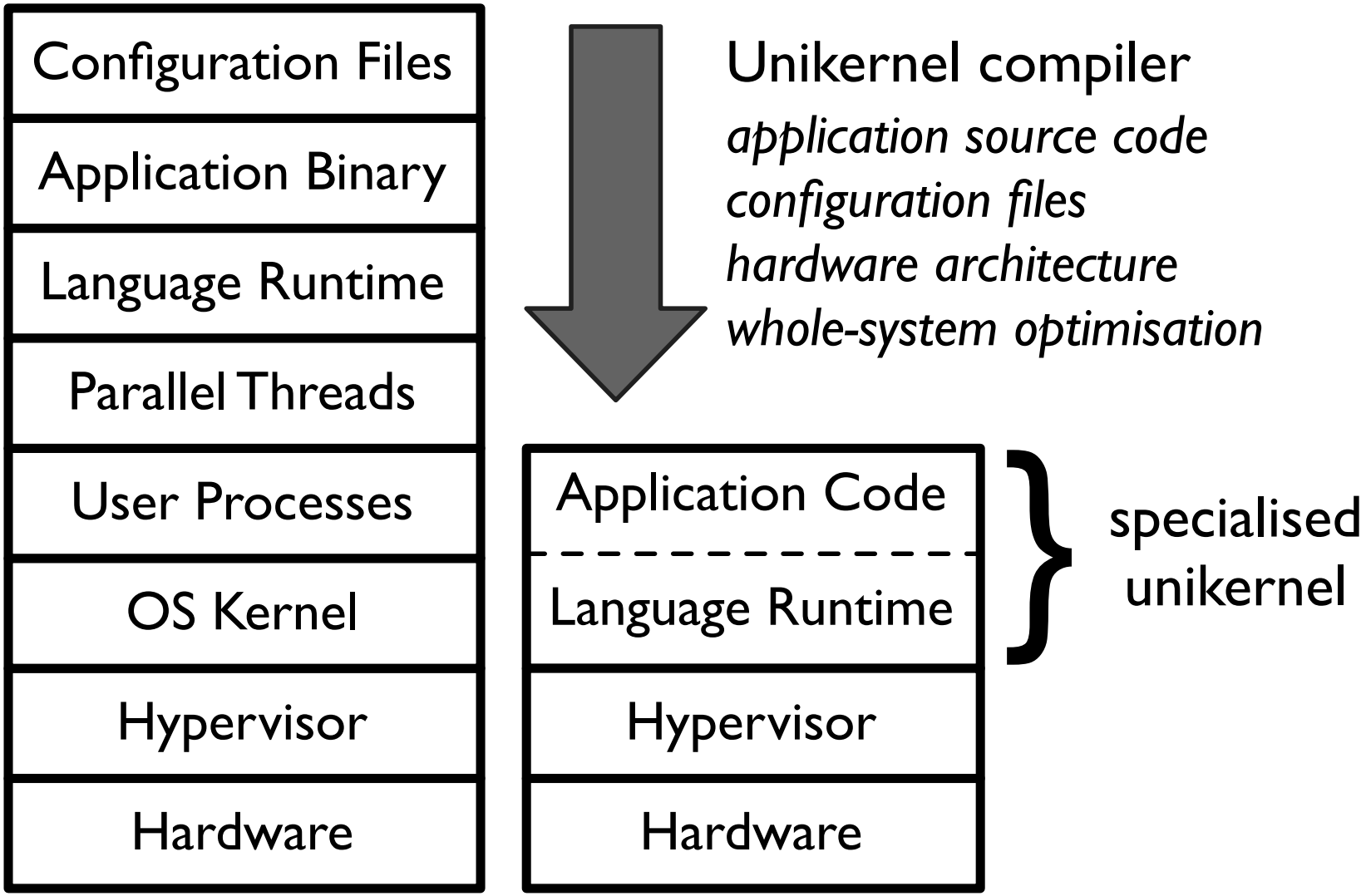
LAMP

- **L**inux
- **A**pache (or Nginx)
- **M**ySQL (or PostgreSQL)
- **P**HP (or Ruby-on-Rails)

MISO

- **M**irage
- **I**rmin
- **S**ignpost
- **O**Caml

Context: Mirage



Irmin

Distributed

Same design principle as Distributed Version Control Systems

- More Git (state based) than Darcs (operational based)

Irmin

Distributed

Same design principle as Distributed Version Control Systems

- More Git (state based) than Darcs (operational based)

Immutable

Like Git, the underlying store should never forgets

- Build mutability as an abstraction

Irmin

Distributed

Same design principle as Distributed Version Control Systems

- More Git (state based) than Darcs (operational based)

Immutable

Like Git, the underlying store should never forgets

- Build mutability as an abstraction

Large Scale

Unlike Git, creation of *partial* replicas is easy and fast

- Replicas should be both clients *and* servers (p2p)
- Replicas should work on partial fetches

Branch Consistency

- History metadata are also stored in the database

Branch Consistency

- History metadata are also stored in the database
- Every operation is relative to an history state
  `read(h, key)` $\rightsquigarrow$ `value`

Branch Consistency

- History metadata are also stored in the database
- Every operation is relative to an history state
  `read(h, key)` $\rightsquigarrow$ `value`
- Every update operation produces a new history state
  `update(h, key, value)` $\rightsquigarrow$ `h'`

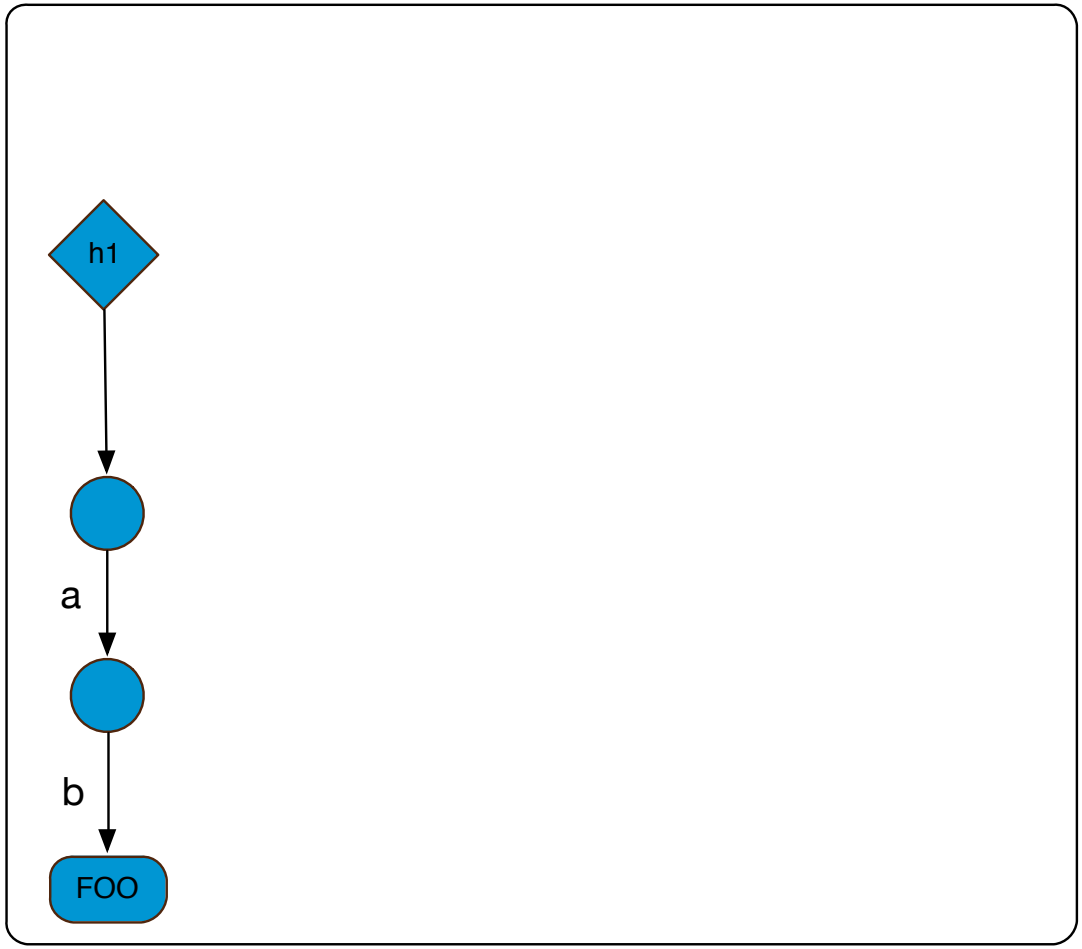
Branch Consistency

- History metadata are also stored in the database
- Every operation is relative to an history state
  `read(h, key)` $\rightsquigarrow$ `value`
- Every update operation produces a new history state
  `update(h, key, value)` $\rightsquigarrow$ `h'`
- All the replica are always consistent (by design)

Branch Consistency

- History metadata are also stored in the database
- Every operation is relative to an history state
  `read(h, key)` $\rightsquigarrow$ `value`
- Every update operation produces a new history state
  `update(h, key, value)` $\rightsquigarrow$ `h'`
- All the replica are always consistent (by design)
- Their union forms a global database

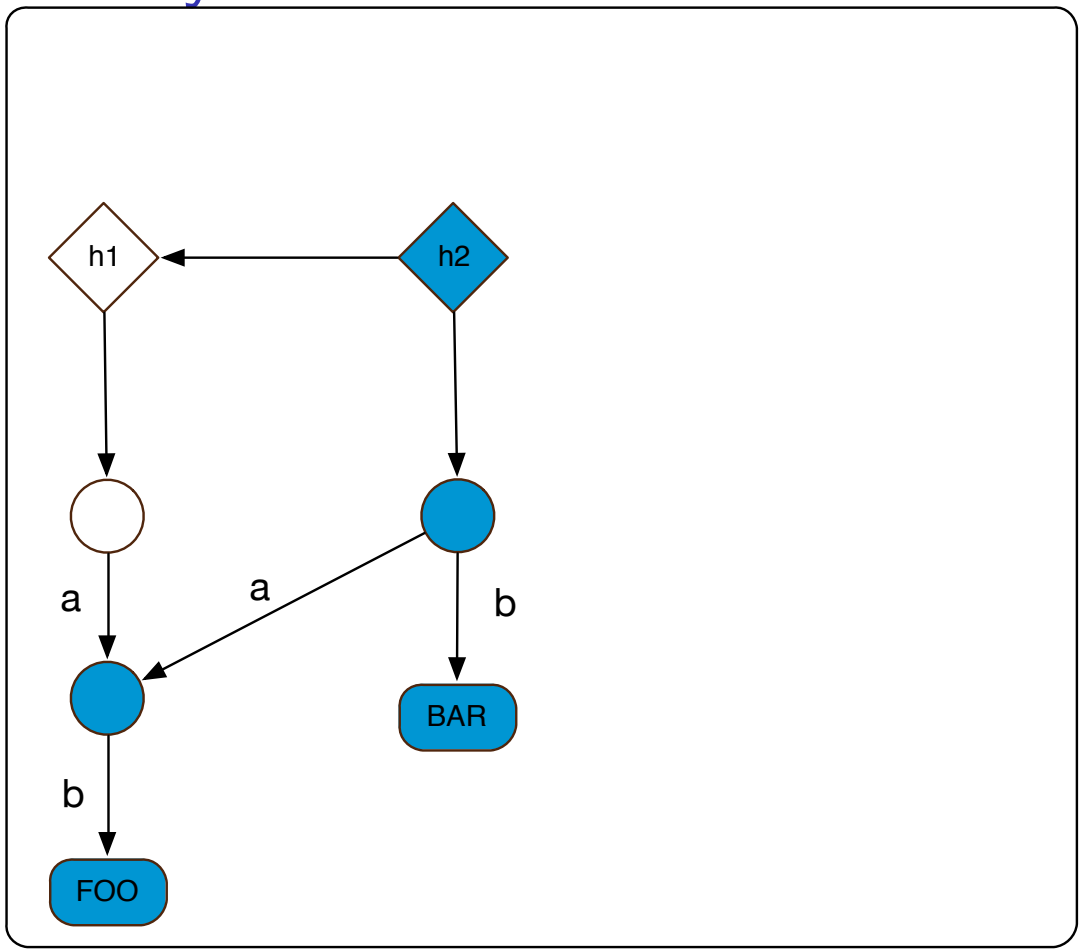
Branch Consistency



Process 1

- $\texttt{update}(\bot, \texttt{a/b}, \text{“FOO”}) \rightsquigarrow h_1$

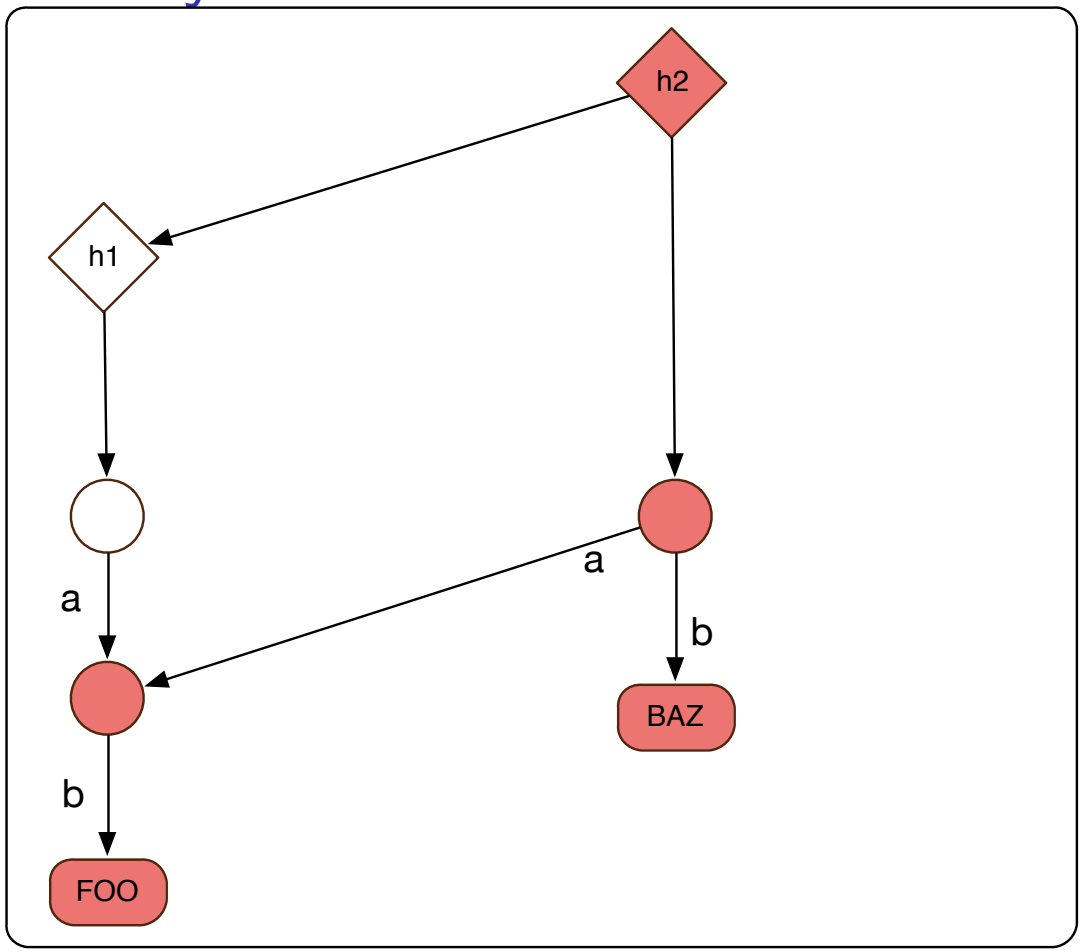
Branch Consistency



Process 1

- $\texttt{update}(\bot, \texttt{a/b}, \text{“FOO”}) \rightsquigarrow h_1$
- $\texttt{update}(h_1, \texttt{b}\quad, \text{“BAR”}) \rightsquigarrow h_2$

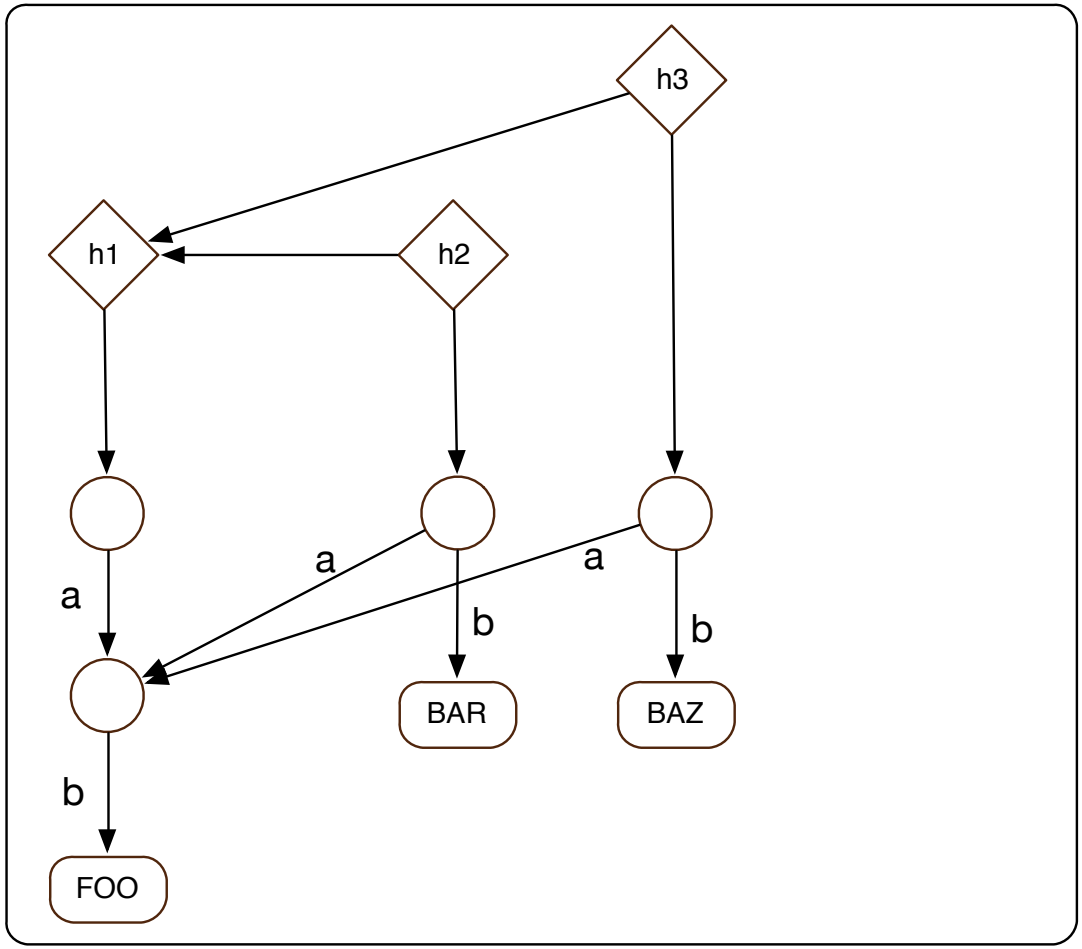
Branch Consistency



Process 2

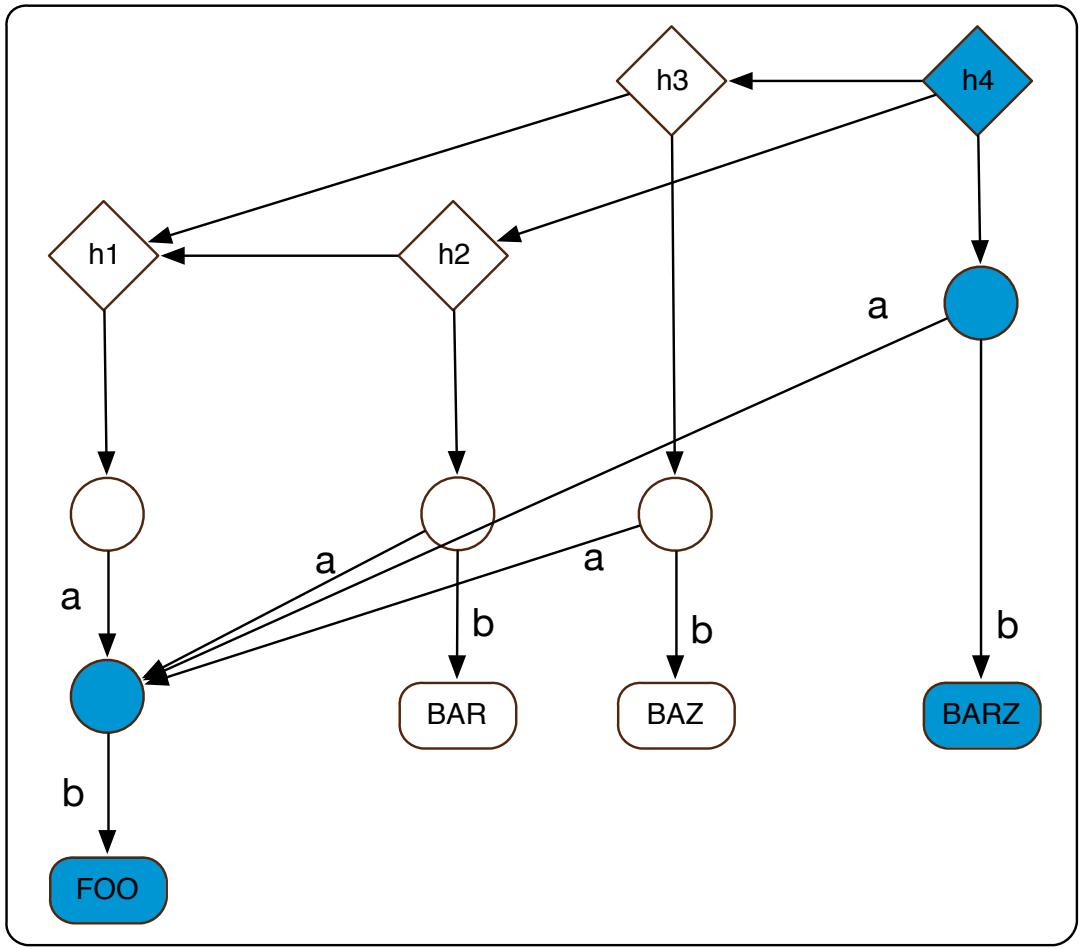
- $\texttt{update}(\bot, \texttt{a/b}, \text{“FOO”}) \rightsquigarrow h_1$
- $\texttt{update}(h_1, \texttt{b}\quad, \text{“BAZ”}) \rightsquigarrow h_3$

Branch Consistency



Global Graph

Branch Consistency



Explicit Reconciliation

- $\texttt{update}(h_2 + h_3, \texttt{b}, \text{“BARZ”}) \rightsquigarrow h_4$

Branch Consistency

- **Every replica is a branch pointer in the global database**

Branch Consistency

- **Every replica is a branch pointer in the global database**
- All operations are relative to a branch pointer
  - They might modify the pointer

Branch Consistency

- **Every replica is a branch pointer in the global database**
- All operations are relative to a branch pointer
  - They might modify the pointer
- Branch pointers are the only mutable part of the system
  - They do not need to be global
  - The *current* branch is implicit

Library Database

Complex Storage Policies

"All replicas should eventually be in sync".

"My phone should sync to my personal cloud only when wifi is available".

"Data stored in the cloud should be encrypted".

Library Database

Heterogeneous Devices

- cloud
- smart-phones / tablets
- IoT (Internet of Things)
- ...

Heterogeneous Scheduling

- Encryption
- Data locality
- Paths of data migration
- ...

Library Database

No Unique Solution

Library Database

No Unique Solution

Library Database

- Base policies and combinators available as libraries
- An application is a composition of policies
- Heterogeneous backends
  - Git
  - in-memory
  - HTTP
  - Distributed Hash-table (DHT)
  - convergent encryption store (ExoStore)
  - ...

Library Database

```
module Git = IrminGit.Make
       (IrminKey.SHA1)
       (IrminContents.String)
       (IrminReference.String)

module Store = (val Git.create ~bare:true
                               ~kind:‘Disk
                               ~root:path ())
```

Library Database

```
let main () =
  Store.create ()                    >>= fun t1 ->
  Store.update t1 ["a";"b"] "FOO"    >>= fun () ->
  Store.branch t1 "process2"         >>= fun t2 ->
  Store.update t1 ["b"]     "BAR"    >>= fun () ->
  Store.update t2 ["b"]     "BAZ"    >>= fun () ->
  Store.merge  t2 ~into:t1           >>= function
  | `Ok ()       -> return_unit
  | `Conflict _ ->
    Store.update t2 ["b"]  "BARZ"    >>= fun () ->
    Store.update t1 ["b"]  "BARZ"    >>= fun () ->
    Store.merge_exn t2 ~into:t1
```

The (only?) slide to remember

The (only?) slide to remember

Irmin “secret ingredients”

1. **Branch Consistency**
2. **Library Database**

Summary

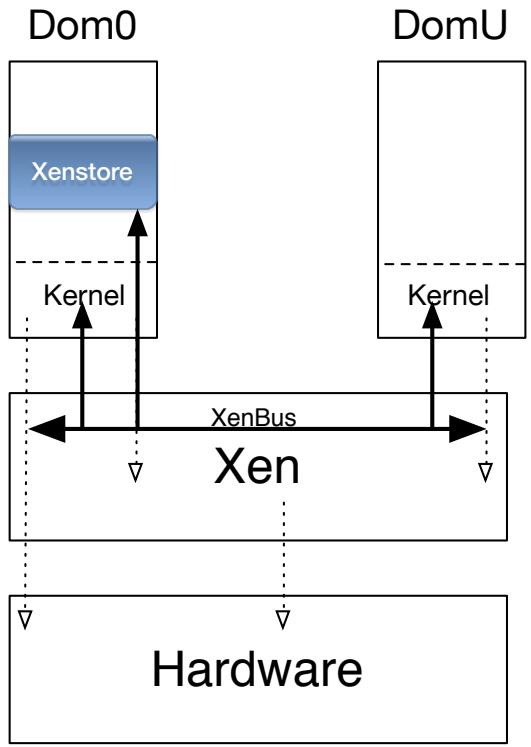
Use-Cases

Irmin co-development

- Active cycle between use-cases and Irmin dev
- Find the right abstractions
  - Be easy enough to use and to integrate
- Provide the right performance
  - Should work on real systems

New use-cases are welcome!

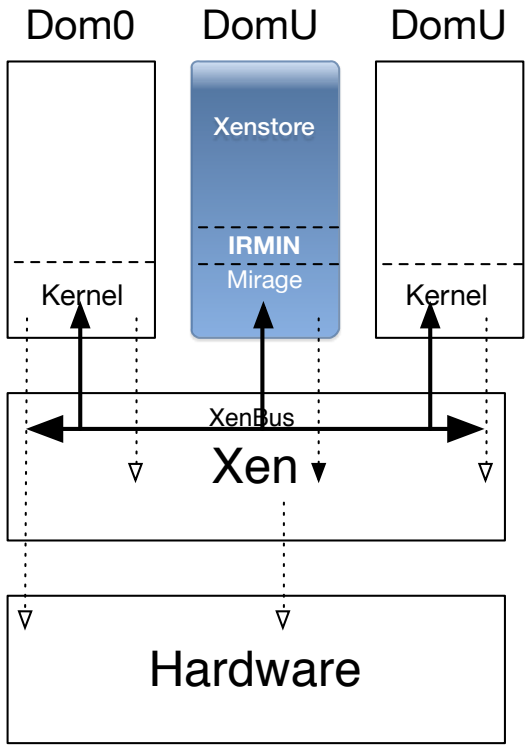
Xenstore [David Scott]



Legacy Xenstore

- Efficient, single-host, in-memory database
- Fast inter-process communication channels ($\sim$ 30 per VM)
- Critical to the proper functioning of hosts running *Xen*

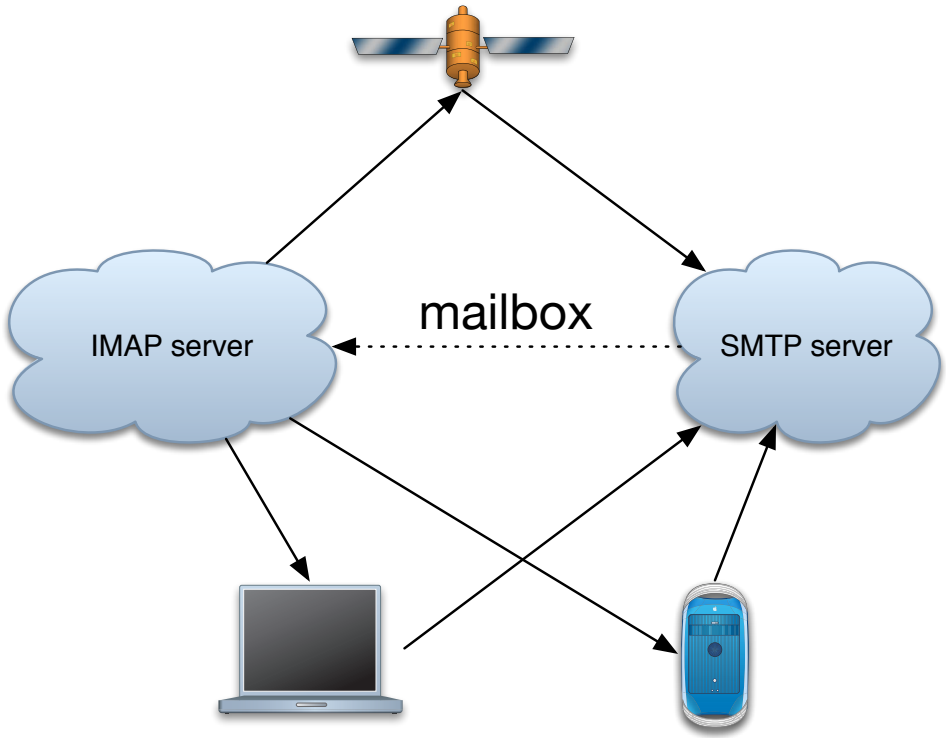
Xenstore [David Scott]



Using Irmin

- Fault-tolerance
- Full diagnostic event tracing system
- RPC tree for general inter-process communication

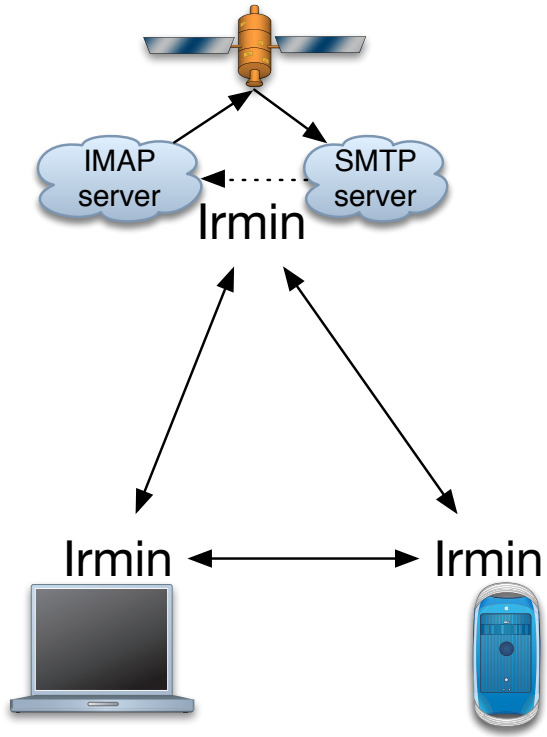
IMAPlet [Gregory Tsipenyuk]



Legacy IMAP

- Custom protocol to synchronize mailboxes
- Separate protocol to receive and send emails (SMTP)

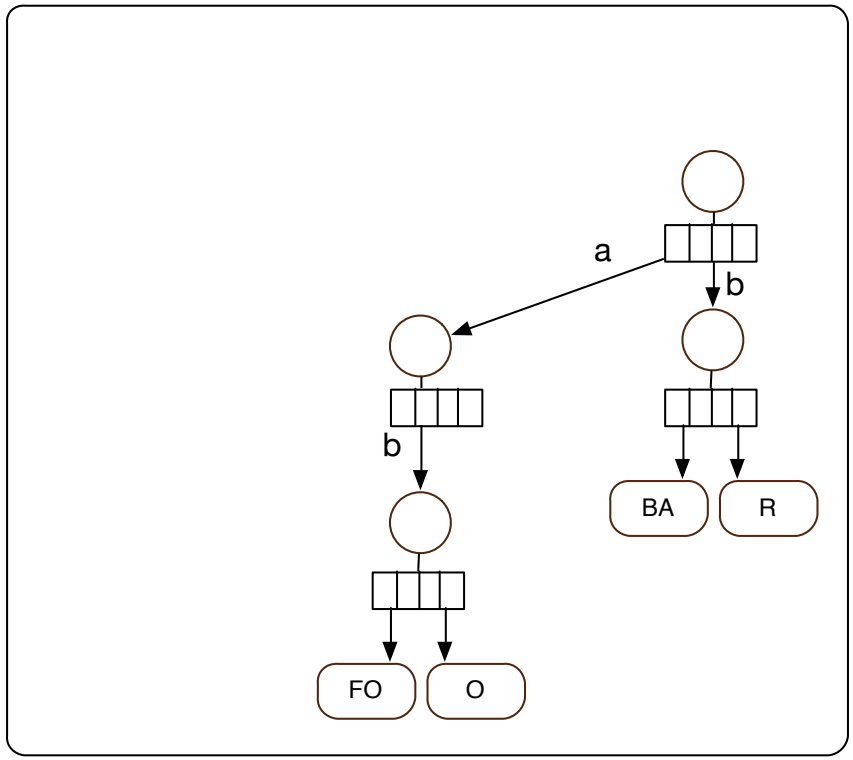
IMAPlet [Gregory Tsipenyuk]



Using Irmin

- Can run in the browser
- Use Irmin to synchronize mailboxes and send emails
- History and revert come for free

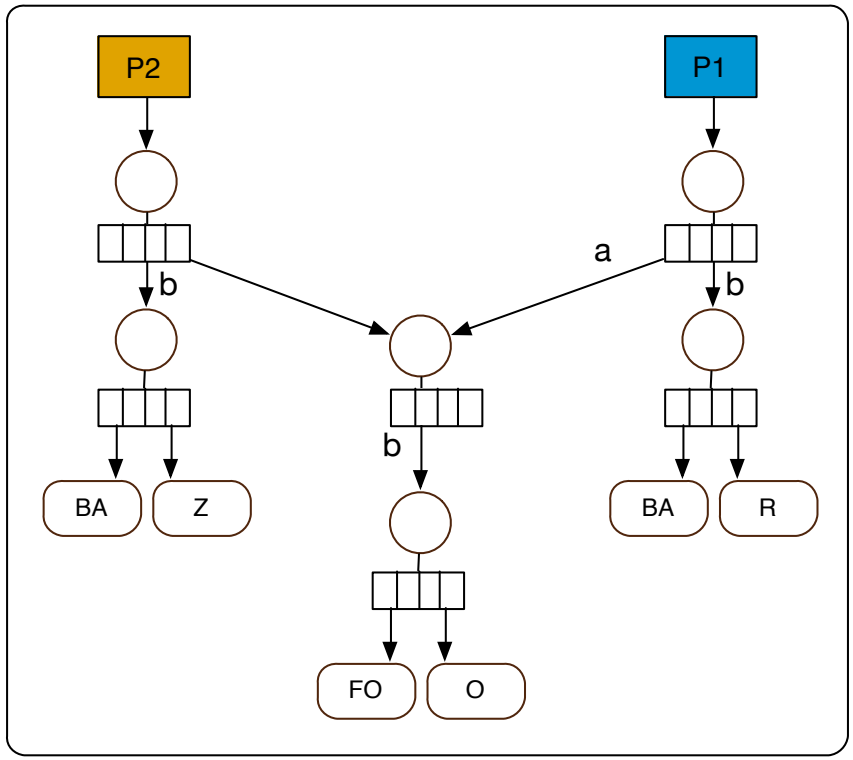
eFS [David Sheets]



Legacy Filesystem

- Prefix-trees with bounded nodes (inodes and memory pages)
- Concurrent semantics not very well defined
- Most implementations use locks

eFS [David Sheets]



Using Irmin

- Every pair PID/fd has a branch
- Explicit syncing points
- Structured files with custom merge policies

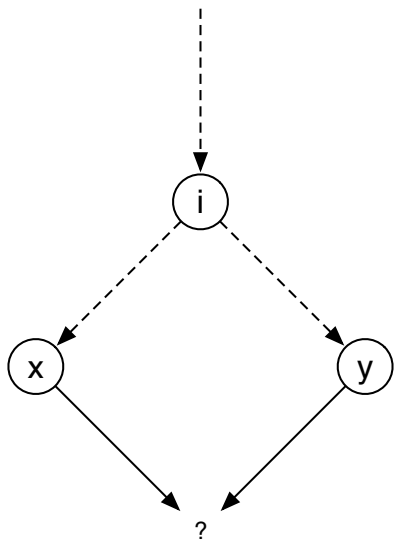
Summary

Mergeable Contents

Application-specific data-structures

User-provided merge function

- 3-way merge



Mergeable Contents

Distributed Counters

- type: *integers*
- merge function:
  $f(i, x, y) = i + (x - i) + (y - i)$

Mergeable Contents

Distributed Counters

- type: *integers*
- merge function:
  $f(i, x, y) = i + (x - i) + (y - i)$

Distributed Sets

- type: polymorphic *sets*
- merge function:
  $f(I, X, Y) = I \cup (X \setminus I) \cup (Y \setminus I) \setminus (I \setminus X) \setminus (I \setminus Y))$

Mergeable Contents

Distributed Counters

- type: *integers*
- merge function:
  $f(i, x, y) = i + (x - i) + (y - i)$

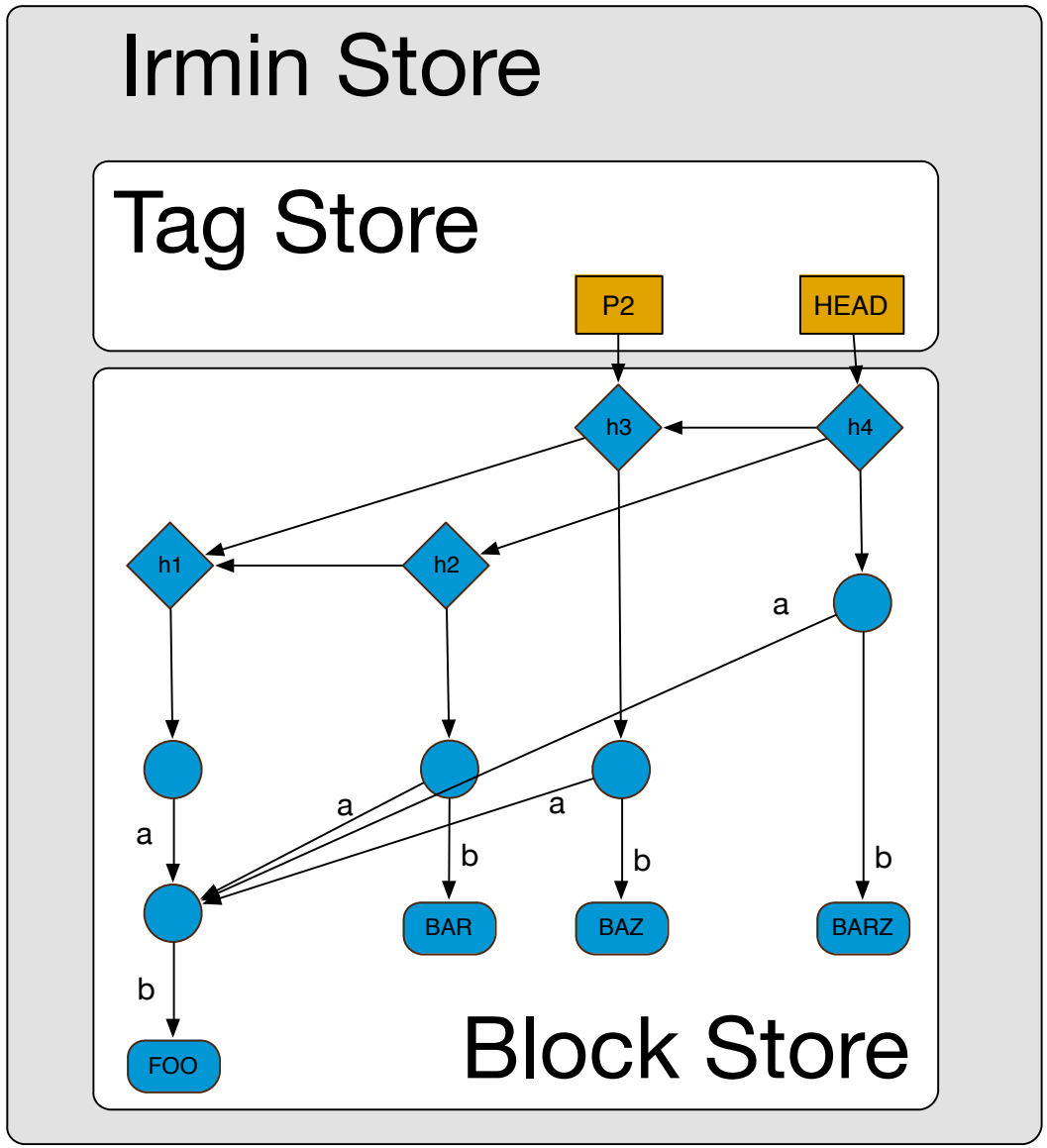
Distributed Sets

- type: polymorphic *sets*
- merge function:
  $f(I, X, Y) = I \cup (X \setminus I) \cup (Y \setminus I) \setminus (I \setminus X) \setminus (I \setminus Y))$

More Complex Datastructures [Benjamin Farinier]

- Distributed queues / stacks
- Ropes, Blame-trees, (ie. fast strings merging)

Block Store


Irmin Store
Tag Store
P2
HEAD
h3
h4
h1
h2
a
a
a
a
b
b
b
b
BAR
BAZ
BARZ
b
FOO
Block Store

Block Store

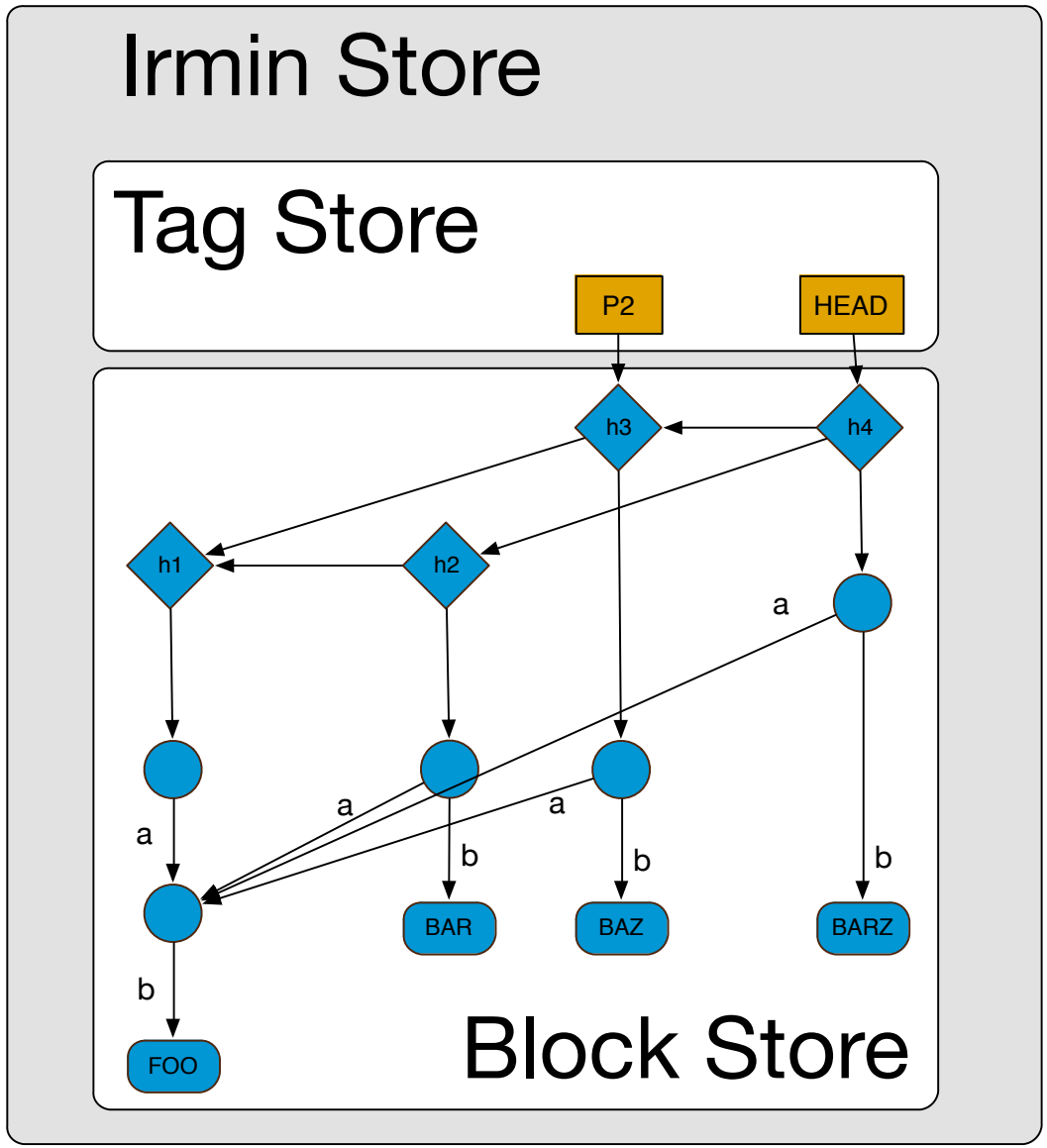
- **Append-only** key / value store
- Potentially very large (unbounded growth)
- *Values* are sequence of bytes
  - Raw blobs
  - Serialized structured values
  - Serialized history metadata
- *Keys* are computed deterministically from the values (`SHAxx`)
- Very simple interface: easy to create new backends
  - Git
  - In-memory
  - Raw block devices
  - HTTP
  - Encrypted

Block Store

Replication

- Fast hash set reconciliation for keys synchronization
  - data is immutable: **no conflict!**
  - Bloom Filters [Magnus Skjegstad]
- Lazy exchange of values
- Merge create new values
  - conflict resolution local to the replica
  - use custom merge functions
  - need to avoid stacking merge commits (associativity helps)
- Support for partial replicas
  - Context resolution of names [David Sheets]
  - Use direct mapping to backends (Git submodules)

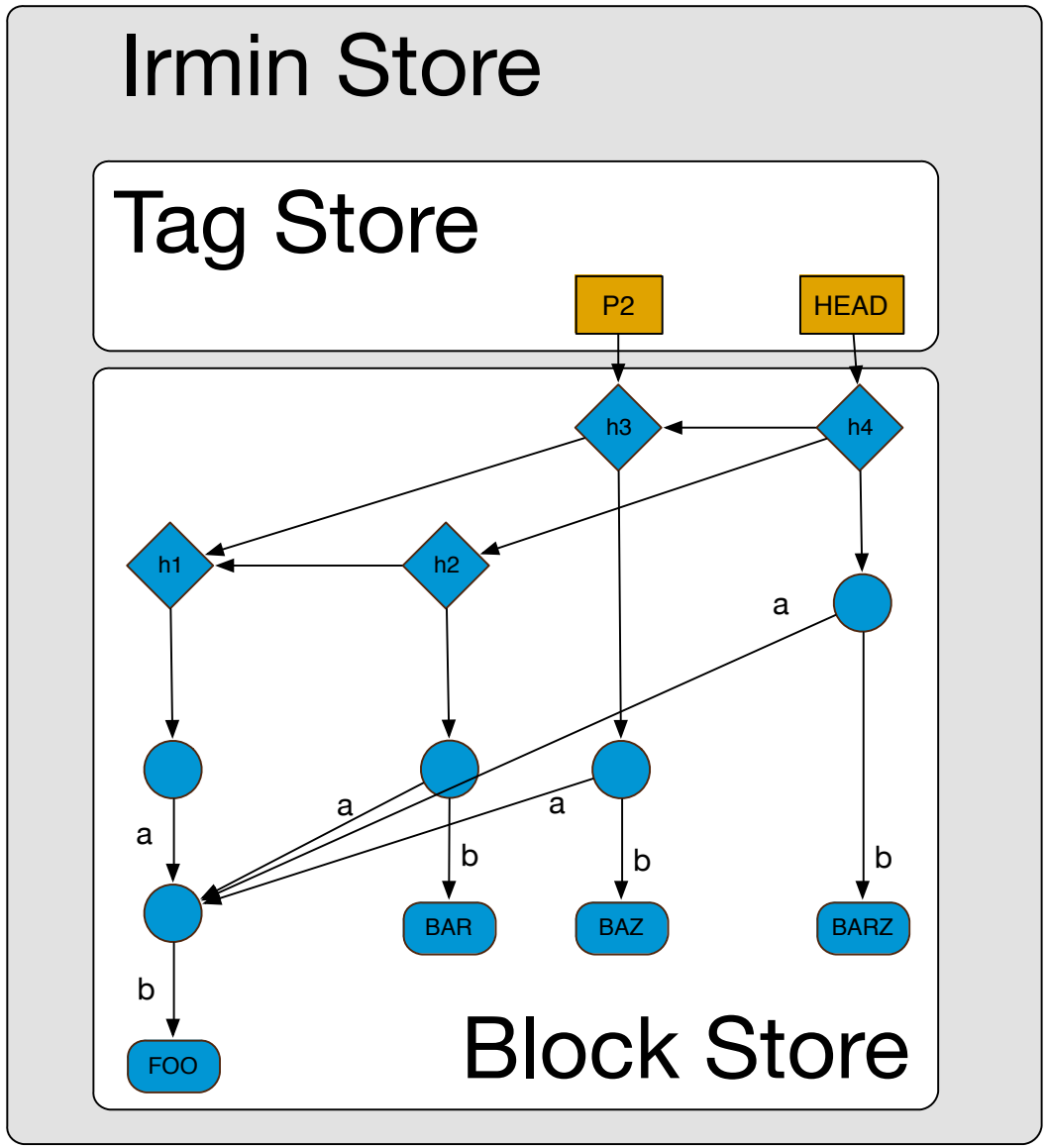
Tag Store


Irmin Store
Tag Store
P2
HEAD
h3
h4
h1
h2
a
a
a
a
b
b
b
b
BAR
BAZ
BARZ
b
FOO
Block Store

Tag store

- **Mutable** key / value store
- Small: usually one key (Git's `HEAD`)
- *Keys* are branch names
- *Values* are pointers to history state
- Should support event notification
- Very simple interface: easy to create new backends
  - Git
  - In-memory
  - DNS
- Do not need to be replicated!

Irmin Store


Irmin Store
Tag Store
P2
HEAD
h3
h4
h1
h2
a
a
a
a
b
b
b
b
BAR
BAZ
BARZ
FOO
Block Store

Irmin Store

Generated Mutable Store

- **Mutable** and structured store
- Automatically **generated** from:
  - Mergeable contents
  - Implementations for the block and tag stores
- Potentially very large
- *Keys* are tree paths
- *Values* are user-defined (need a merge function)

Irmin Store

Content-Addressable Store

- Keep a complete history of updates
  - Keep track of provenance
  - Snapshot / revert come for free
- Support for encryption
- Validation of synchronization consistency (Merkle Tree)

Future Work

Managing Unbounded Growth

Buy more disks

- commodity storage steadily becomes more and more inexpensive

Compression

- Store the most recent object uncompressed
- Store older objects as reverse diffs (reverse VHD)

Managing Unbounded Growth

Garbage Collection

- The block store is a distributed heap
- Tags points to the GC roots
- Usual (distributed) GC algorithms
- Need to keep track of global tag usage

Managing Unbounded Growth

Garbage Collection

- The block store is a distributed heap
- Tags points to the GC roots
- Usual (distributed) GC algorithms
- Need to keep track of global tag usage

Sparse History (Rebase)

- Each replica maintain two branches
  - A complete but bounded history: “the last 100 commit”
  - A sparse history: “A commit every hour”
- Regular squash-rebasing from the bounded to the sparse history
- Only synchronize the sparse histories
- Each replica runs its local GC

And More!

More backends, more mergeable contents

Benchmarks

Interfacing with non-OCaml code

- External API
  - Higher-level REST API (JSON over HTTP)
  - Binary protocols (using Google's Protocol buffer ?)
- Library
  - Ctypes description to generate automatic bindings to other languages

Questions ?

Try it!

Resources

`https://github.com/mirage/irmin`
`https://github.com/mirage/irmin/wiki/Getting-Started`

Use-Cases

- Xenstore: `https://github.com/djs55/ocaml-xenstore`
- IMAPlet: `https://github.com/ocamllabs/imaplet`
- eFS: `https://github.com/dsheets/efs`