Reliability & Chubby

CSE 490H

Overview

- Writable / WritableComparable
- Reliability review
- Chubby + PAXOS

Datatypes in Hadoop

- Hadoop provides support for primitive datatypes
  - String → Text
  - Integer → IntWritable
  - Long → LongWritable
  - FloatWritable, DoubleWritable, ByteWritable, ArrayWritable…

The *Writable* Interface

```
interface Writable {
  public void readFields(DataInput in);
  public void write(DataOutput out);
}
```

Example: LongWritable

```
public class LongWritable implements
    WritableComparable {
  private long value;

  public void readFields(DataInput in)
      throws IOException {
    value = in.readLong();
  }

  public void write(DataOutput out)
      throws IOException {
    out.writeLong(value);
  }
}
```

WritableComparable

- Extends *Writable* so the data can be used as a key, not just a value

```
int compareTo(Object what)
int hashCode()
```

```
this.compareTo(x) == 0 =>
  x.hashCode() == this.hashCode()
```

A Composite Writable

```
class IntPairWritable implements Writable {
  private int fst;
  private int snd;

  public void readFields(DataInput in)
      throws IOException {
    fst = in.readInt();
    snd = in.readInt();
}

  public void write(DataOutput out)
      throws IOException {
    out.writeInt(fst);
    out.writeInt(snd);
  }
}
```

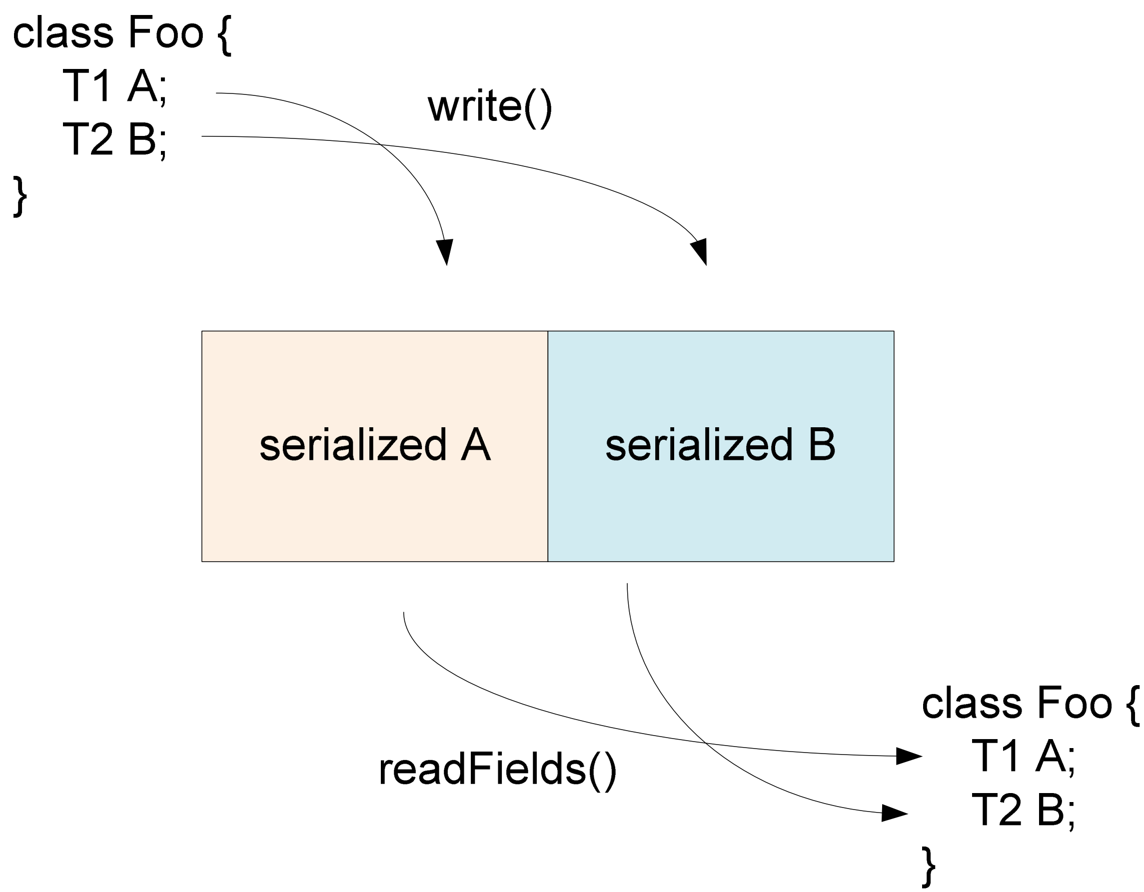
A Composite Writable (2)

```
class IntPairWritable implements Writable {
  private IntWritable fst;
  private IntWritable snd;

  public void readFields(DataInput in)
      throws IOException {
    fst.readFields(in);
    snd.readFields(in);
  }

  public void write(DataOutput out)
      throws IOException {
    fst.write(out);
    snd.write(out);
  }
}
```

Marshalling Order Constraint



- readFields() and write() must operate in the same order

Subclassing is problematic

```
class AaronsData implements Writable { }
class TypeA extends AaronsData {
  int fieldA;
}

class TypeB extends AaronsData {
  float fieldB;
}
```

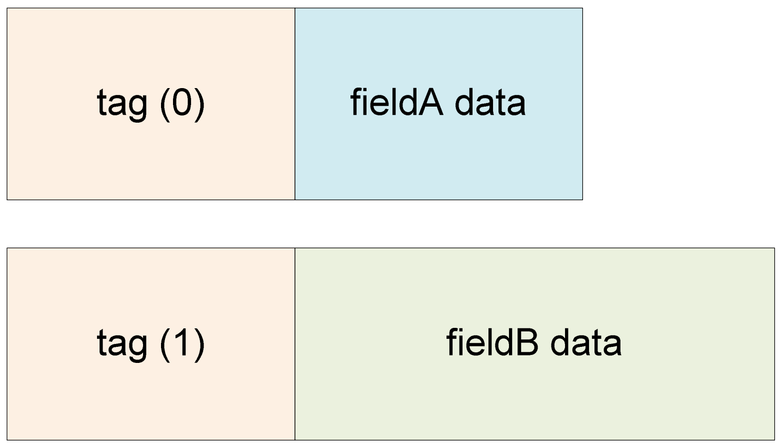
- Cannot do this with Hadoop!

Attempt 2…

```
class AaronsData implements Writable {
   int fieldA;
   float fieldB;
}
```

- But we only want to populate one field at a time; how do we determine which is the “real” field?

Looking at the Bytes



Tag-Discriminated Union

```
class AaronsData implements Writable {
   static final int TYPE_A = 0, TYPE_B = 1;
   int TAG;
   int fieldA;
   float fieldB;

   void readFields(DataInput in) {
     TAG = in.readInt();
     if (TAG == TYPE_A) { fieldA = in.readInt(); }
     else { fieldB = in.readFloat(); }
  }
}
```

Reliability

Reliability Demands

- Support partial failure
  - Total system must support graceful decline in application performance rather than a full halt

Reliability Demands

- Data Recoverability
  - If components fail, their workload must be picked up by still-functioning units

Reliability Demands

- Individual Recoverability
  - Nodes that fail and restart must be able to rejoin the group activity without a full group restart

Reliability Demands

- Consistency
  - Concurrent operations or partial internal failures should not cause externally visible nondeterminism

Reliability Demands

- Scalability
  - Adding increased load to a system should not cause outright failure, but a graceful decline
  - Increasing resources should support a proportional increase in load capacity

Reliability Demands

- Security
  - The entire system should be impervious to unauthorized access
  - Requires considering many more attack vectors than single-machine systems

Ken Arnold, CORBA designer:

"Failure is the defining difference between distributed and local programming"

Component Failure

- Individual nodes simply stop

Data Failure

- Packets omitted by overtaxed router
- Or dropped by full receive-buffer in kernel
- Corrupt data retrieved from disk or net

Network Failure

- External & internal links can die
  - Some can be routed around in ring or mesh topology
  - Star topology may cause individual nodes to appear to halt
  - Tree topology may cause “split”
  - Messages may be sent multiple times or not at all or in corrupted form…

Timing Failure

- Temporal properties may be violated
  - Lack of “heartbeat” message may be interpreted as component halt
  - Clock skew between nodes may confuse version-aware data readers

Byzantine Failure

- Difficult-to-reason-about circumstances arise
  - Commands sent to foreign node are not confirmed: What can we reason about the state of the system?

Malicious Failure

- Malicious (or maybe naïve) operator injects invalid or harmful commands into system

Preparing for Failure

- Distributed systems must be robust to these failure conditions
- But there are lots of pitfalls…

The Eight Design Fallacies

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

-- Peter Deutsch and James Gosling, Sun Microsystems

Dealing With Component Failure

- Use heartbeats to monitor component availability
- “Buddy” or “Parent” node is aware of desired computation and can restart it elsewhere if needed
- Individual storage nodes should not be the sole owner of data
  - Pitfall: How do you keep replicas consistent?

Dealing With Data Failure

- Data should be check-summed and verified at several points
  - Never trust another machine to do your data validation!
- Sequence identifiers can be used to ensure commands, packets are not lost

Dealing With Network Failure

- Have well-defined split policy
  - Networks should routinely self-discover topology
  - Well-defined arbitration/leader election protocols determine authoritative components
    - Inactive components should gracefully clean up and wait for network rejoin

Dealing With Other Failures

- Individual application-specific problems can be difficult to envision
- Make as few assumptions about foreign machines as possible
- Design for security at each step

Chubby

What is it?

- A *coarse-grained lock service*
  - Other distributed systems can use this to synchronize access to shared resources
- Intended for use by “loosely-coupled distributed systems”

Design Goals

- High availability
- Reliability

- Anti-goals:
  - High performance
  - Throughput
  - Storage capacity

Intended Use Cases

- GFS: Elect a master
- BigTable: master election, client discovery, table service locking
- Well-known location to bootstrap larger systems
- Partition workloads
- Locks should be **coarse**: held for hours or days – build your own fast locks on top

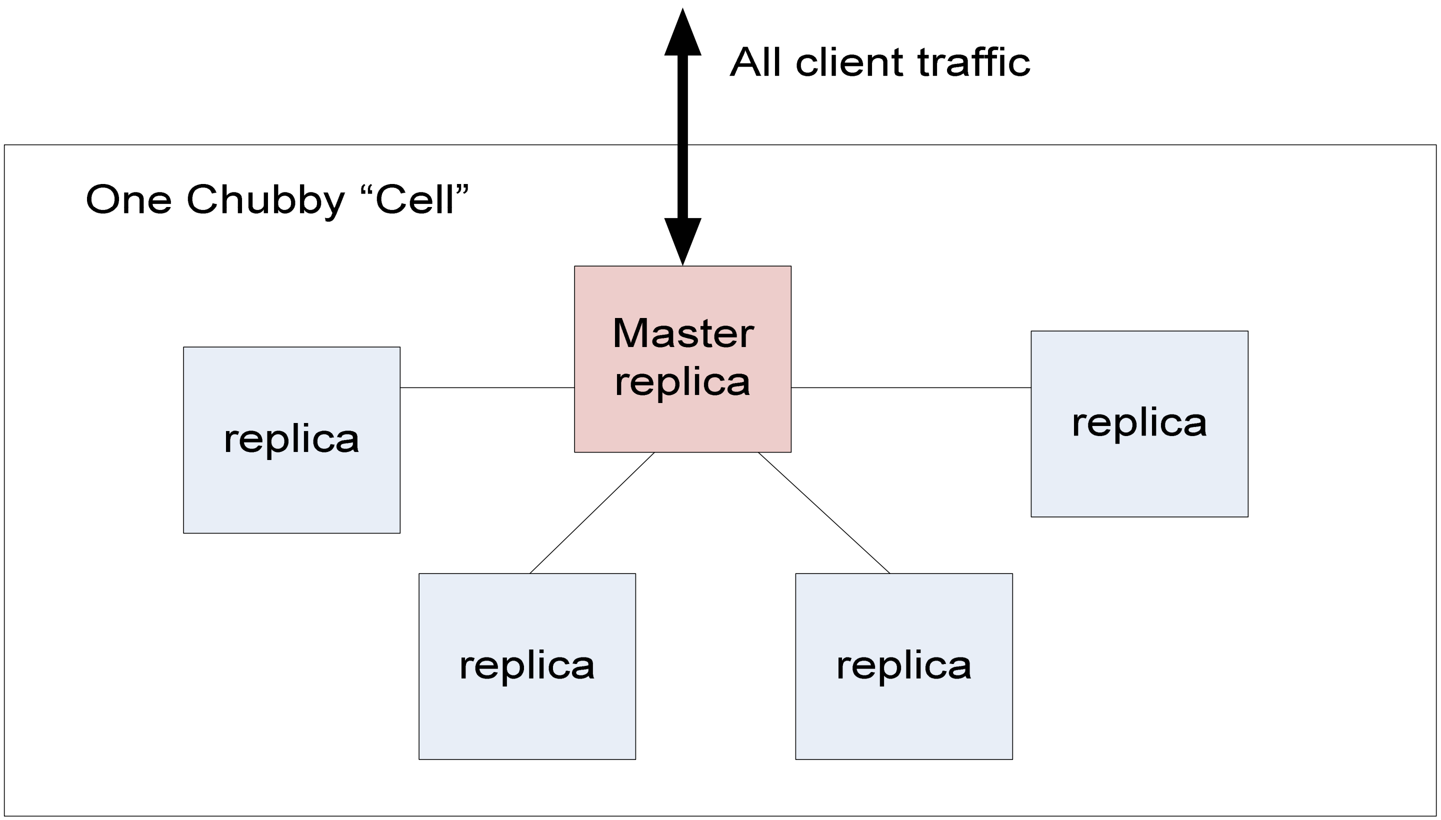
External Interface

- Presents a simple distributed file system
- Clients can open/close/read/write files
  - Reads and writes are *whole-file*
  - Also supports *advisory* reader/writer locks
  - Clients can register for notification of file update

Files == Locks?

- "Files" are just *handles to information*
- These handles can have several *attributes*
  - The contents of the file is one (primary) attribute
  - As is the owner of the file, permissions, date modified, etc
  - Can also have an attribute indicating whether the file is locked or not.

Topology


All client traffic
One Chubby "Cell"
Master
replica
replica
replica
replica
replica

Master election

- Master election is simple: all replicas try to acquire a write lock on designated file. The one who gets the lock is the master.
  - Master can then write its address to file; other replicas can read this file to discover the chosen master name.
  - Chubby doubles as a *name service*

Distributed Consensus

- Chubby cell is usually 5 replicas
  - 3 must be alive for cell to be viable
- How do replicas in Chubby agree on their own master, official lock values?
  - PAXOS algorithm

PAXOS

- Paxos is a family of algorithms (by Leslie Lamport) designed to provide *distributed consensus* in a **network** of several **processors**.

Processor Assumptions

- Operate at arbitrary speed
- Independent, random failures
- Procs with stable storage may rejoin protocol after failure
- Do not lie, collude, or attempt to maliciously subvert the protocol

Network Assumptions

- All processors can communicate with (“see”) one another
- Messages are sent asynchronously and may take arbitrarily long to deliver
- Order of messages is not guaranteed: they may be lost, reordered, or duplicated
- Messages, if delivered, are not corrupted in the process

A Fault Tolerant Memory of Facts

- Paxos provides a memory for individual “facts” in the network.
- A **fact** is a binding from a variable to a value.
- Paxos between 2F+1 processors is reliable and can make progress if up to F of them fail.

Roles

- Proposer – An agent that proposes a fact
- Leader – the authoritative proposer
- Acceptor – holds agreed-upon facts in its memory
- Learner – May retrieve a fact from the system

Safety Guarantees

- Nontriviality: Only *proposed* values can be learned
- Consistency: Only at most one value can be learned
- Liveness: If at least one value V has been proposed, eventually any learner L will get *some* value

Key Idea

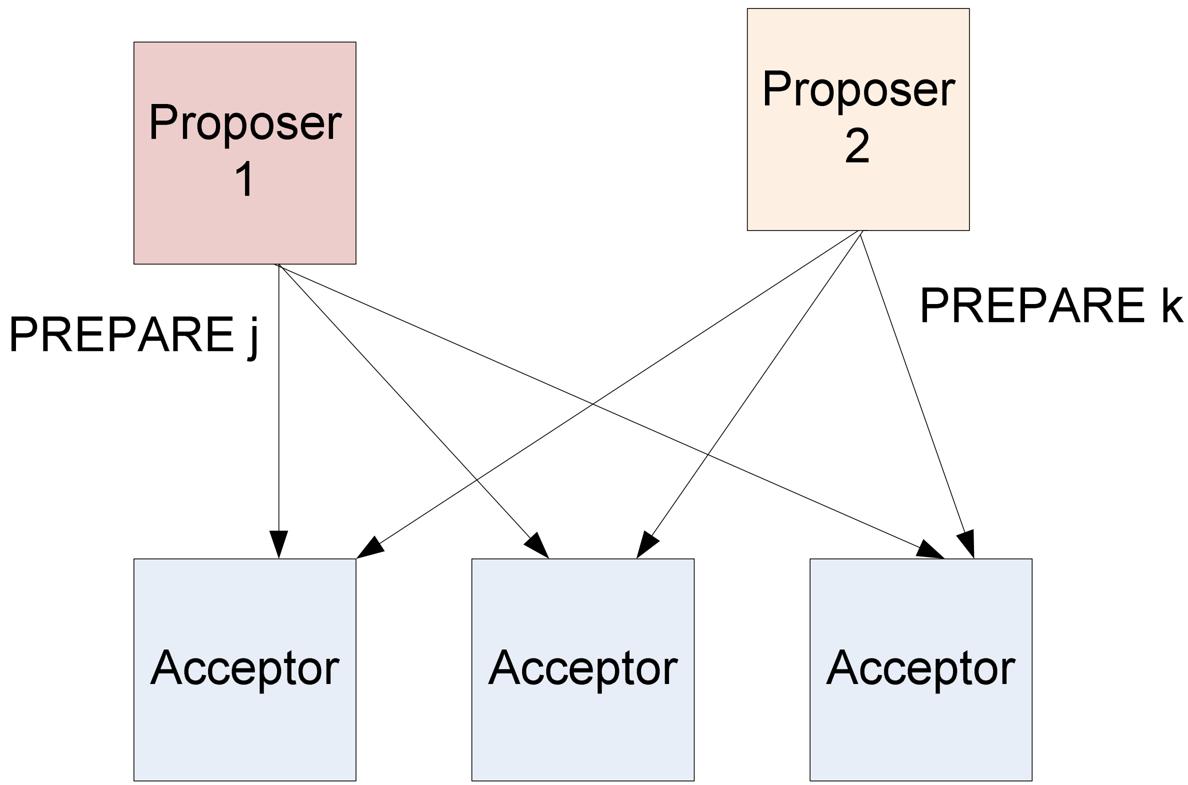
- Acceptors do not act unilaterally. For a fact to be learned, a **quorum** of acceptors must agree upon the fact
- A quorum is any majority of acceptors
- Given acceptors {A, B, C, D}, Q = {{A, B, C}, {A, B, D}, {B, C, D}, {A, C, D}}

Basic Paxos

- Determines the authoritative value for a single variable

- Several proposers offer a value V_n to set the variable to.
- The system converges on a single agreed-upon V to be the fact.

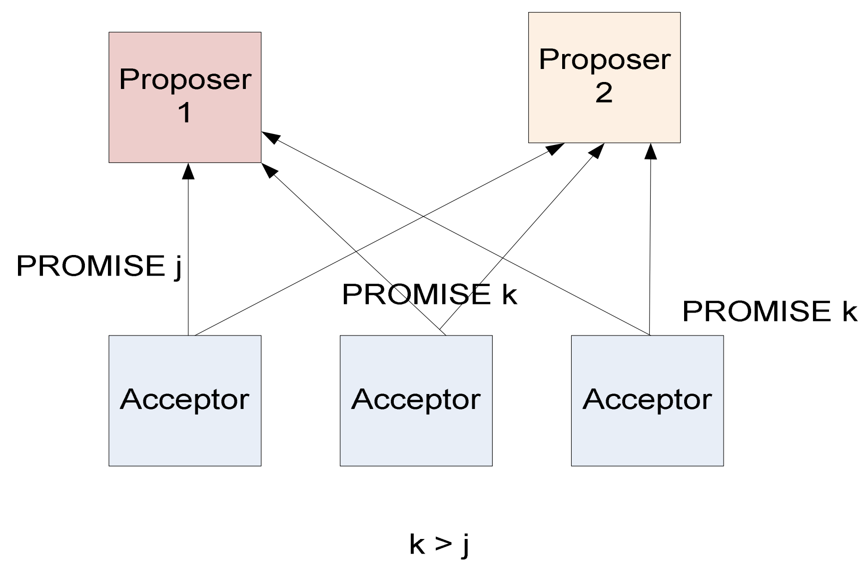
Step 1: Prepare



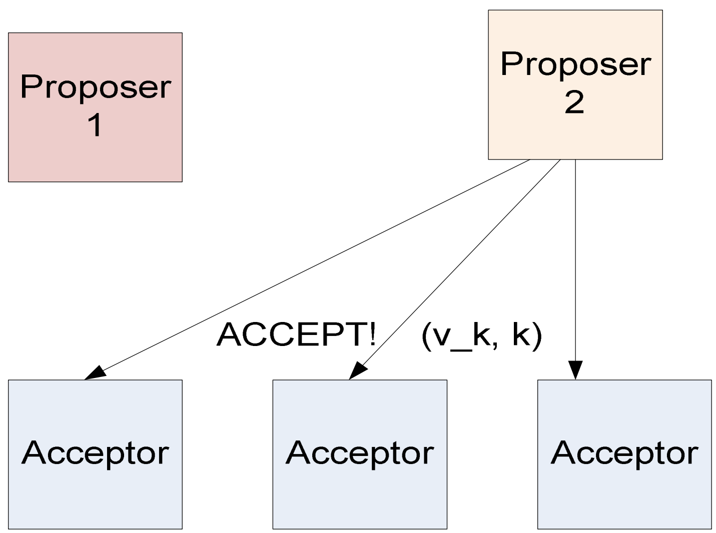
k > j

Step 2: Promise

- PROMISE x – Acceptor will accept proposals only numbered *x* or higher
- Proposer 1 is *ineligible* because a quorum has voted for a higher number than *j*

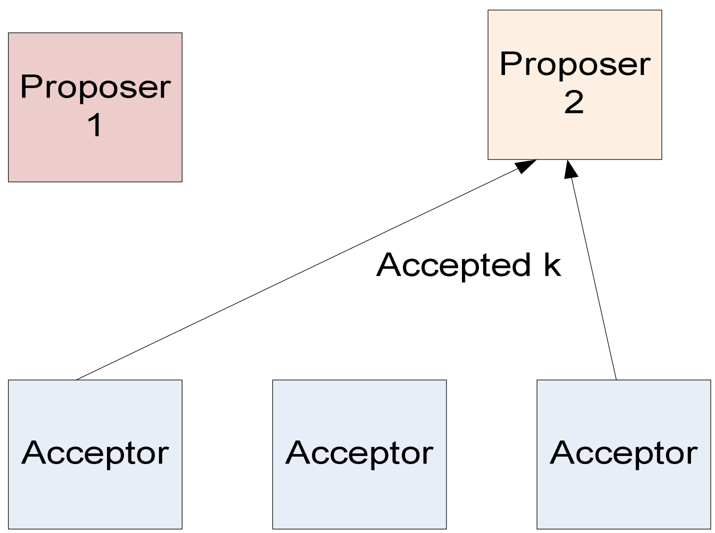


Step 3: Accept!



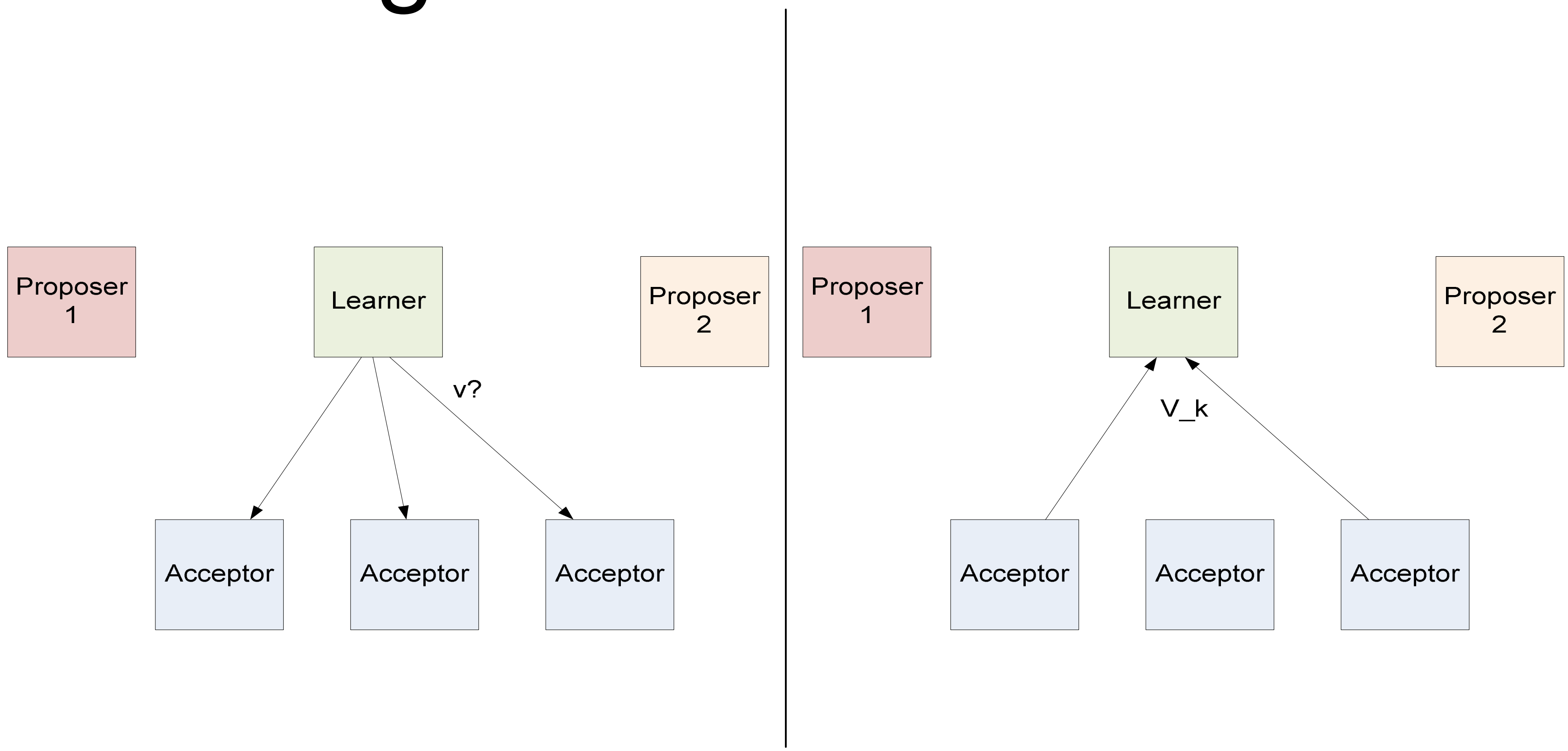
Proposer 1 is disqualified; Proposer 2 offers a value

Step 4: Accepted



A quorum has accepted value v_k; it is now a fact

Learning values



If a learner interrogates the system, a quorum will respond with fact V_k

Basic Paxos…

- Proposer 1 is free to try again with a proposal number > k; can take over leadership and write in a new authoritative value
  - Official fact will change “atomically” on all acceptors from perspective of learners
  - If a leader dies mid-negotiation, value just drops, another leader tries with higher proposal

More Paxos Algorithms

- Not whole story
- MultiPaxos: steps 1—2 done once, 3—4 repeated multiple times by same leader
- Also: cheap Paxos, fast Paxos, generalized Paxos, Byzantine Paxos…

Paxos in Chubby

- Replicas in a cell initially use Paxos to establish the leader.
- Majority of replicas must agree
- Replicas promise not to try to elect new master for at least a few seconds ("master lease")
- Master lease is periodically renewed

Client Updates

- All client updates go through master
- Master updates official database; sends copy of update to replicas
  - Majority of replicas must acknowledge receipt of update before master writes its own value
- Clients find master through DNS
  - Contacting replica causes redirect to master

Chubby File System

- Looks like simple UNIX FS: /ls/foo/wombat
  - All filenames start with ‘/ls’ (“lockservice”)
  - Second component is cell (“foo”)
  - Rest of the path is anything you want
- No inter-directory move operation
- Permissions use ACLs, non-inherited
- No symlinks/hardlinks

Files

- Files have version numbers attached
- Opening a file receives handle to file
  - Clients cache all file data including file-not-found
  - Locks are *advisory* – not required to open file

Why Not Mandatory Locks?

- Locks represent client-controlled resources; how can Chubby enforce this?
- Mandatory locks imply shutting down client apps entirely to do debugging
  - Shutting down distributed applications much trickier than in single-machine case

Callbacks

- Master notifies clients if files modified, created, deleted, lock status changes
- Push-style notifications decrease bandwidth from constant polling

Cache Consistency

- Clients cache all file content
- Must send respond to Keep-Alive message from server at frequent interval
- KA messages include invalidation requests
  - Responding to KA implies acknowledgement of cache invalidation
- Modification only continues after all caches invalidated or KA time out

Client Sessions

- **Sessions** maintained between client and server
  - Keep-alive messages required to maintain session every few seconds
- If session is lost, server releases any client-held handles.
- What if master is late with next keep-alive?
  - Client has its own (longer) timeout to detect server failure

Master Failure

- If client does not hear back about keep-alive in *local lease timeout*, session is **in jeopardy**
  - Clear local cache
  - Wait for “grace period” (about 45 seconds)
  - Continue attempt to contact master
- Successful attempt => ok; jeopardy over
- Failed attempt => session assumed lost

Master Failure (2)

- If replicas lose contact with master, they wait for grace period (shorter: 4—6 secs)
- On timeout, hold new election

Reliability

- Started out using replicated Berkeley DB
- Now uses custom write-thru logging DB
- Entire database periodically sent to GFS
  - In a different data center
- Chubby replicas span multiple racks

Scalability

- 90K+ clients communicate with a single Chubby master (2 CPUs)
- System increases lease times from 12 sec up to 60 secs under heavy load
- Clients cache virtually everything
- Data is small – all held in RAM (as well as disk)

Conclusion

- Simple protocols win again
- Piggybacking data on Keep-alive is a simple, reliable coherency protocol