Practical Consensus
Consensus == agreement
Consensus = agreement

1. meta data for protocol? (config files)
2. who is leader?
3. who has a lock?
4. what is stored in data?
5. what is in a snapshot?

writing consensus code is hard!!! you don't want every team to rewrite this code.
Third, a lock-based interface is more familiar to our programmers. Both the replicated state machine of Paxos and the critical sections associated with exclusive locks can provide the programmer with the illusion of sequential programming. However, many programmers have come across locks before, and think they know to use them. Ironically, such programmers are usually wrong, especially when they use locks in a distributed system; few consider the effects of independent machine failures on locks in a system with asynchronous communications. Nevertheless, the apparent familiarity of locks overcomes a hurdle in persuading programmers to use a reliable mechanism for distributed decision making.

suggests that an event notification mechanism would be useful to avoid polling.

- Even if clients need not poll files periodically, many will; this is a consequence of supporting many developers. Thus, caching of files is desirable.
- Our developers are confused by non-intuitive caching semantics, so we prefer consistent caching.
- To avoid both financial loss and jail time, we provide security mechanisms, including access control.
General Analysis of consensus algs.

**Termination**: nodes in protocol converge on a decision

Termination = leader is elected

**Agreement**: the decision is the same across node

all nodes agree on who the leader

**Integrity**: if all nodes propose the same value then, then the decision is that same value

the leader is the node with the most votes -> majority
<table>
<thead>
<tr>
<th>Converge on a decision</th>
<th>Decision is the same across all nodes</th>
<th>Decision is the same as that proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAFT Election</td>
<td>3-party agreement</td>
<td>3-party agreement</td>
</tr>
<tr>
<td>Chandy-Upfal protocol</td>
<td>Terminalization</td>
<td>Integrity</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Termination</th>
<th>Agreement</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Randomized timeout</td>
<td>(1) Majority (2) (\geq 2 + 1)</td>
<td>(1) Each node can only vote once</td>
</tr>
<tr>
<td>(1) No markers are dropped</td>
<td>(1) Each node's snapshot is complete after markers received from other node</td>
<td>(1) Reordering of markers</td>
</tr>
<tr>
<td>Converge on a decision</td>
<td>Decision is the same across all nodes</td>
<td>Decision is as that proposed</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td><strong>Termination</strong></td>
<td>Agreement</td>
<td>Integrity</td>
</tr>
<tr>
<td>Raft</td>
<td>RAFT Election</td>
<td></td>
</tr>
<tr>
<td>Leader Election</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chandy Lomport</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to provide consensus?

As a library? Or as a service?

1. Need to restructure your code into state machines to use library

2. If deployed to a cluster, consensus is running on 1k servers (501 servers)
How to provide consensus?

As a library? Or as a service?

- Need to restructure your code into state machines to use library
- Your code library -> compile -> Binary
- Deploy on lots of servers
- Cluster running your app
  - Need to change your code to make RPC calls
  - Managed by another team

1. Decoupling of app from consensus
   a. You don't need to code state machines
   b. Majority is not a function of app deployment but of service size
How to provide consensus?

As a library?

Or as a service?

developer need to manage perf & scale

must write code as state machines to use library

large quorum (=> half) apps

Trust that service handles scale

easier to write apps simple to interact with via syscalls

smaller quorum ¼/2 of nodes in service (e.g. 3 of 5)
<table>
<thead>
<tr>
<th>Consensus Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
</tr>
<tr>
<td>etcd</td>
</tr>
<tr>
<td>chubby</td>
</tr>
<tr>
<td>ZooKeeper</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>Protocol</td>
</tr>
<tr>
<td>Raft</td>
</tr>
<tr>
<td>Paxos</td>
</tr>
<tr>
<td>ZAB</td>
</tr>
</tbody>
</table>
Zookeeper (a consensus service)

client library to interface with service provided

Each file is a znode

API is a Unix filesystem abstract.
Zookeeper

client library to interface with service. Provided caching

API is a Unix filesystem abstract. Each file is a znode.

getting a lock = create a file

leader election is the same

when leader crashes, the file is deleted by Zookeeper

whoever creates the file gets the lock
Lock = trying to create a file
& since only one node can
create a file ⇒ only one node
can get the lock

no lock

has lock

leader election = all nodes try create
file ⇒ whoever wins
is the leader
Consensus Service API \(=\) UNIX API

\[
\text{Open}(\text{path}, \text{data, acl, flags})
\]

Sequential file

- Name of the file
- Metadata (e.g. config)

Ephemeral file

- Zookeeper deletes the file when the client disappears

Permanent

- Your app needs to delete the file
while (true) {
    lock = open("/lock"");
    if (lock != err)
        // got the lock do stuff
    else {
        sleep(10)
    }
}

Busy waiting = sleep check & sleep pattern

* wastes CPU on nodes
* increases load on service

Solution: add a notification and alert nodes when lock is free
no notifications

sleep until notified

loop of wake-up/catch

no loop to check & sleep
getData(path, watch)
exists(path, watch)
getChildren(path, watch)

send notification when file changes
Native lock Implementation

```python
while (true) {
    lock = open("/lock...")
    if (lock != err) {
        Do stuff
        exit loop
    } else {
        sleep (10)
    }
}
```

```python
while (true) {
    lock = open("/lock...")
    if (lock != err) {
        Do stuff
        exit loop
    } else wait (lock)
}
```

only check when file is deleted

---

No more retries so less load

but still unnecessary load because only one will win

- Original: 
  - Open
  - Retry
  - Open
  - Create

- Modified: 
  - Open
  - Deleted
  - Open
  - Create

- New Service
  - Deleted
  - Open
  - Create
Native lock Implementation

Solution: Give each node a sequential filename & only notify one at a time.
while (true) {
    lock fn
}

for (lock Kock) {
    wait
    do stuff
    exit
    loop Model
}

else or you wait (lock_{-1})

if (lock = open "lock") {
    for (lock Kock) {
        wait
        do stuff
        exit
        loop Model
    }
}

else (lock = open "lock") {
    while (true) {
        wait (lock)
    }
}

Say file
Consensus Service API == UNIX API

Open( path, data, acl, flags)

Sequential file
- if the file exists, then create a file with a post-pended seq num
- open("/lock")
- if "lock" already exists then add seq num

Ephemeral file
- file is automatically deleted when node fails

Permanent
- client explicitly deletes

paths data, acl, flags

* file: if the file exists, then create a file with a post-pended seq num
* ephemeral file: file is automatically deleted when node fails
* permanent: client explicitly deletes
After Deployment of Service

most requests (>90%) are reads!!

Perf = stale / incorrect data
Perf optimization (client caching)

- may increase latency when stale data is being modified
- may work with some stale data
delete (path, version)

set-data (path, data, version)

Used to check that cache is valid. Request will be rejected if versions don't match.
Perf Optimization (read from followers)

- writes
- reads

Spread load for reads but compromises "total ordering"

will get stale reads
Optimizations for read-heavy workloads

* Client-side caching
* Reads from followers

These perf optimizations incur a penalty
Practical Consensus

Analysis of a consensus protocol (TAI)
Motivation for a consensus service (library Vespa service)
Discussion of Zookeeper
- file types
- read optimization