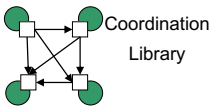
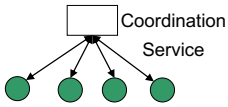
11. Coordination Services

Coordination

- Distributed systems need coordination
- How can it be obtained?



Ex: GCS, consensus



Ex: ZooKeeper, Chubby

- Coordination services are the present and probably the future: Google, Yahoo!, Microsoft, …

Coordination Services

- Distributed algorithms:
  - Hard to implement, not simple to use for coordination
  - (Typically) non scalable: how to run a consensus among hundreds of processes?
  - Programming model not clear…
- Coordination services:
  - Write and make dependable once, use always
  - Expertise in fault-tolerance and distributed systems programming needed for the coordination service programmers, not so much for the application programmers
  - A single service can be shared by many applications
    - Example: 90k+ clients communicate with a single Google's Chubby master (2 CPUs – info from 2006)

Coordination Services

- Provide concurrent objects
  - Programming model is shared memory, usually simpler than distributed (message-passing) algorithms
  - If wait-free, allow the implementation of wait-free objects
  - If universal (consensus number infinite), allow consensus between any number of processes
- Used in distributed systems so the object has to be emulated using distributed algorithms
  - Therefore, distributed algorithms are used after all…
  - Fault tolerance and security are requirements
  - Performance is a requirement (low latency, high throughput, predictability, etc.)

Coordination Services

- In summary, coordination services provide **high-available small storage**, **interface with synchronization power**, and **client failure detection**
- There are many coordination services available

System	Data Model	Sync. Primitive	Wait-free
Boxwood [44]	Key-Value store	Locks	No
Chubby [17]	(Small) File system	Locks	No
Sinfonia [6]	Key-Value store	Microtransactions	Yes
DepSpace [14]	Tuple space	cas/replace ops	Yes
ZooKeeper [31]	Hierar. of data nodes	Sequencers	Yes
etcd [3]	Hierar. of data nodes	Sequen./Atomic ops	Yes
LogCabin [5]	Hierar. of data nodes	Conditions	Yes

from T. Distler et al. Extensible Distributed Coordination. ACM EuroSys'15.

Chubby

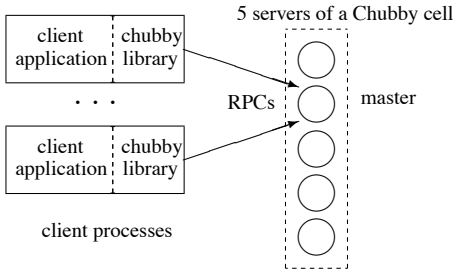
Mike Burrows "The Chubby lock service for loosely-coupled distributed systems", USENIX OSDI' 2006

Google' Chubby

- A *coarse-grained lock service*
  - Other distributed systems can use this to synchronize access to shared resources
  - Implements a file system interface, with directories, where small files can be stored and files can be locked
  - Provides also **coarse-grained advisory locks**:
    - Corse grained -> locks are held for hours or days
    - They are advisory in the sense that the application should implement/respect lock requirements (Chubby provides sequencers, a kind of ticket that can be used to prove a process have the lock)
- High availability, reliability
  - Employ the Paxos algorithm for replication
- Locks are not wait-free ☹

Chubby – use cases

- Intended for use by "loosely-coupled distributed systems", accessing it as client applications
  - Google File system (GFS): Elect a master
  - BigTable: master election, client discovery, table service locking
  - Well-known location to bootstrap larger systems
  - Partition workloads among many processes



DepSpace

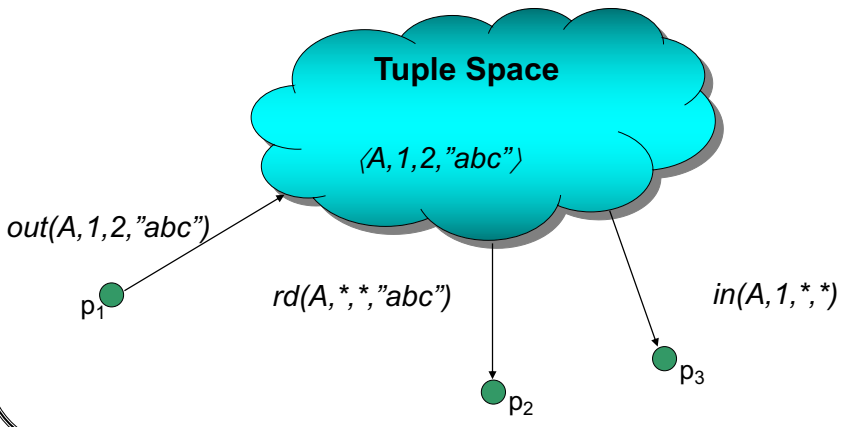
A. Bessani et al. “DepSpace: A Byzantine fault-tolerant coordination service”, ACM EuroSys’08

https://github.com/bft-smart/depspace

DepSpace: Dependable Tuple Space

- DepSpace implements a **dependable augmented tuple space** to be used in untrusted environments
- Tuple Spaces were introduced in the Linda coordination language
  - It introduced many ideas used in coordination services
- A tuple space is a shared memory object where generic data structures called tuples are kept:
  - Tuple access is content-based (“SQL like”);
  - It supports three non-blocking operations:
    - out(t): inserts tuple t in the space;
    - rdp(t’): reads a tuple that matches t’ from the space;
    - inp(t’): reads and remove a tuple that matches t’ from the space
- This is an object with consensus number 2

Tuple Space Model



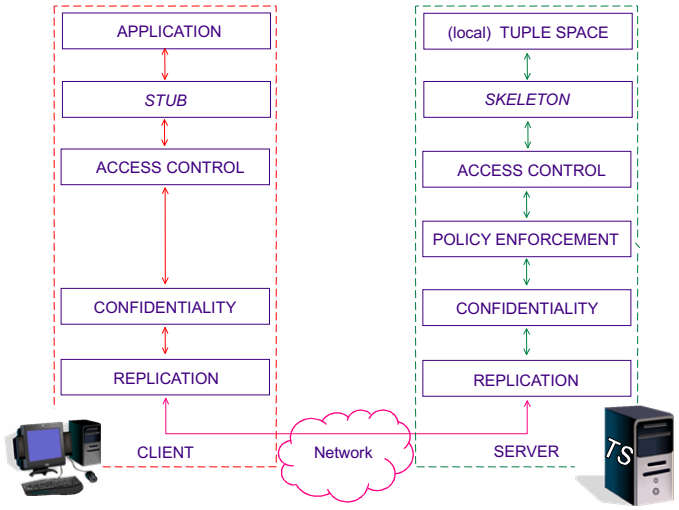
Augmented Tuple Spaces

- We need a tuple space with consensus number ∞ (universal object)
- For that we need a new operation:
  cas = Conditional Atomic Swap

  cas(t',t) ≡ *atomic(*if ¬ *rdp(t')* then *out(t))*
  If *t* is inserted, return *true*, otherwise return *false*.

- A tuple space that supports the *cas* operation is called an Augmented Tuple Space
- A **Policy-Enforced Augmented Tuple Space (PEATS)** also provides means to enforce access policies to the tuple space (and thus tolerate Byzantine failures)

DepSpace Architecture



Using DepSpace

- An example: Solving Byzantine consensus with DepSpace
  - Each process propose a value and all processes decide one of the proposed values
  - In general the solution requires complex protocols…
  - Not with DepSpace…

```
public ByzantineConsensus {
    public static Object propose(DepSpaceAccessor ts, int cid, Object v) {
        DepTuple tbar = DepTuple.createTuple("DEC",cid,null);
        if(ts.cas(tbar,DepTuple.createTuple("DEC",cid,v) ) ) {
            return v;
        }
        return tbar.getFields() [2];
    }
}
```

The Security Policy

cas(<"DEC",cid, x>,<"DEC", cid, y>)

can only be executed if *x is undefined*

ZooKeeper

P. Hunt et al. “ZooKeeper: Wait-free coordination for Internet-scale systems”, USENIX ATC 2010

http://zookeeper.apache.org

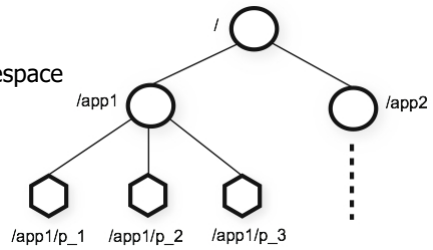
ZooKeeper (Apache/Yahoo!)

- A highly available, scalable, distributed, configuration, consensus, group membership, leader-election, naming, and coordination service
- Open source, widely used in many services
- Similar to Chubby (at high-level)
  - However, Zookeeper does not implement locks, but instead a set of wait-free primitives with consensus number ∞

Data model and namespace

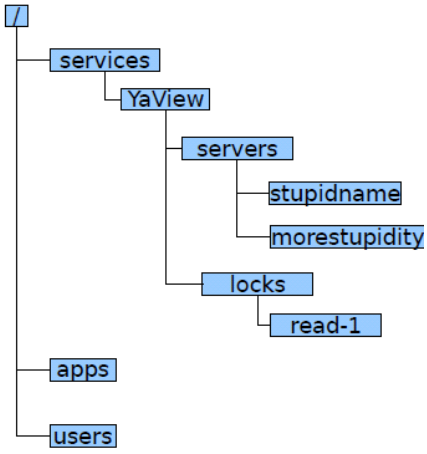
- The namespace provided by ZooKeeper is much like that of a file system
  - A name is a sequence of path elements separated by a slash
  - Every node in ZooKeeper's name space is identified by a path
  - Unlike standard file systems, each node (called **znode**) in a ZooKeeper namespace can have data associated

ZooKeeper's Hierarchical Namespace



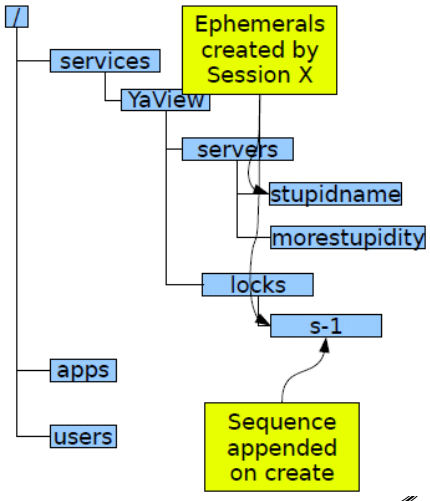
Data model

- Nodes (znodes) maintain a hierarchical structure
  - Includes version numbers for data changes, ACL changes, and timestamps
- Data is read and written in its entirety
  - Each znode has an ACL
- Clients can set watches on znodes
  - When a znode is changed, a packet is sent to the client and the watch is removed (long poll)



Data model

- Ephemeral node
  - Will be deleted when the session that created it times out or is explicitly deleted
- Order
  - Each update is stamped with a number that reflects the order of all ZooKeeper transactions



ZooKeeper Guarantees

- Sequential Consistency
  - Updates from a client will be applied in the order that they were sent
- Atomicity
  - Updates either succeed or fail. No partial results
- Single System Image
  - A client will see the same view of the service regardless of the server that it connects to
- Reliability
  - Once an update has been applied, it will persist from that time forward until a client overwrites the update
- Timeliness
  - Clients view of the system is guaranteed to be up-to-date within a certain time bound

Zookeeper Recipes

- Zookeeper is a coordination service → provides a number of concurrent objects
- The objective is to use it to coordinate processes, i.e., to implement other objects/shared memory algorithms
  - Processes fail only by crash
- Some examples…
  - As before, the pseudo-code presented is for each process

Consensus

```
consensus(id, value)
   create("/consensus/"+id+"/decision", value,…)
   if error then
        return getData("/consensus/"+id+"/decision", null)
        //null → don't put watch
   else
        return value
```

- Works because "create" atomically tests if there exists a znode with the same name before creating the node (if non-existent)

Membership

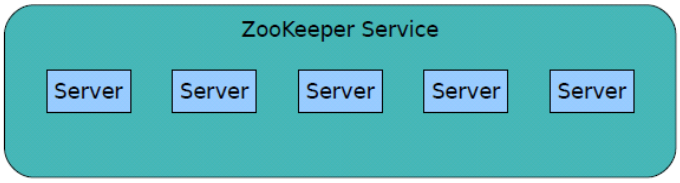
```
join(groupid)      //group id is name with full path
  create(groupid+"/"+hostname, anyvalue,  EPHEMERAL)
                   //automatically removed when crash

leave(groupid)
  delete(groupid+"/"+hostname)

getview(groupid)
  getChildren(groupid)
```


Leader Election

```
leader()
   getData("/servers/leader",true)// true sets watch
   if successful, the writer of  "/servers/leader" is the
       new leader; exit
   create("/servers/leader", hostname, EPHEMERAL)
   if successful, I am the leader; return
   restart procedure
```

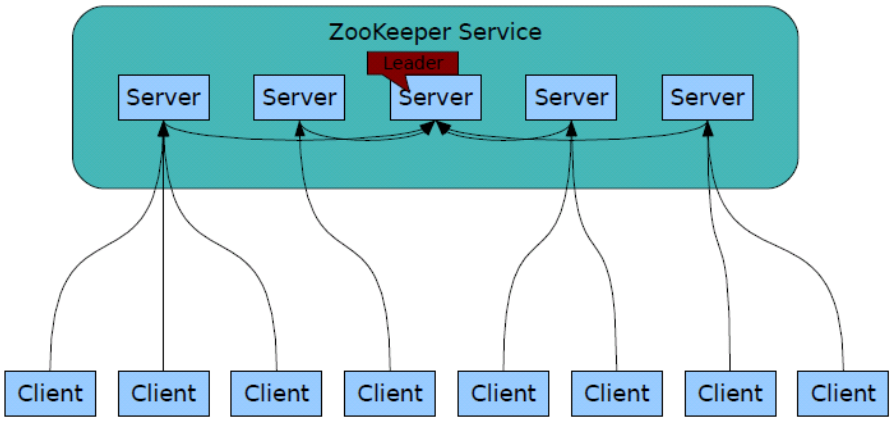
If a watch is triggered for ".../servers/leader", followers will restart the leader election process

Zookeeper implementation



- Replication is based on Zab, a Paxos-like algorithm that ensures updates are applied in FIFO total order
- All servers store a copy of the data on memory and disk (implements durable state machine replication)
- A leader is elected at startup
- Followers service clients, all updates go through the leader (it is a kind of semi-active replication)

ZooKeeper Servers



Client-server communication

- Clients only connect to a single server
  - The client maintains a TCP connection
  - This connection is used to send requests, get responses, get watch events, and send heartbeats
- When the client first connects to the service
  - The server sets up a session for the client
- When server breaks the TCP connection
  - Client connects to a different server
  - Session is reestablished by the new server