

Lecture 10: Zookeeper – A Distributed Coordination Paradigm

CSCI4180 (Fall 2013)

Patrick P. C. Lee

Acknowledgment

➤ Slides are adapted from:

- Dr. T Y Wong's slides in 2012 spring course
- Zookeeper slides by Yahoo!
- Zookeeper tutorial in Eurosys

<https://cwiki.apache.org/ZOOKEEPER/eurosys/tutorial.data/>

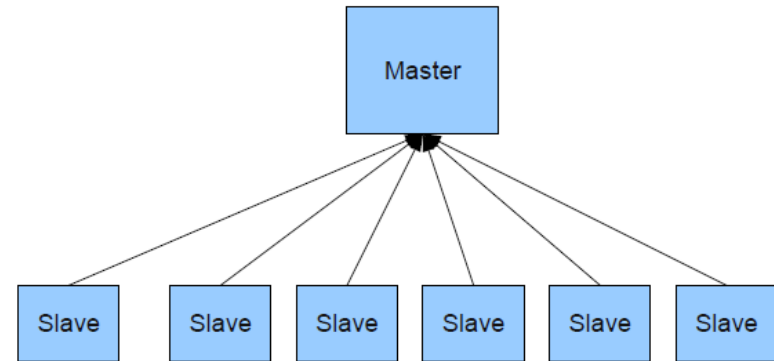
Outline

- Why Zookeeper?
- Zookeeper data model
- Zookeeper programming basics
- Zookeeper applications
 - Distributed lock service
 - Distributed barrier
 - Distributed producer-consumer queue
 - Distributed leader election

Classic Distributed Systems

➤ Work assignment

- Master assigns work
- Workers execute tasks assigned by master



➤ What happens if master crashes?

- Single point of failure
- No work is assigned → a new master is needed

➤ What happens if one worker crashes?

- Okay... but fails if dependencies exist
- Need to detect crashed workers

HBase: Recap

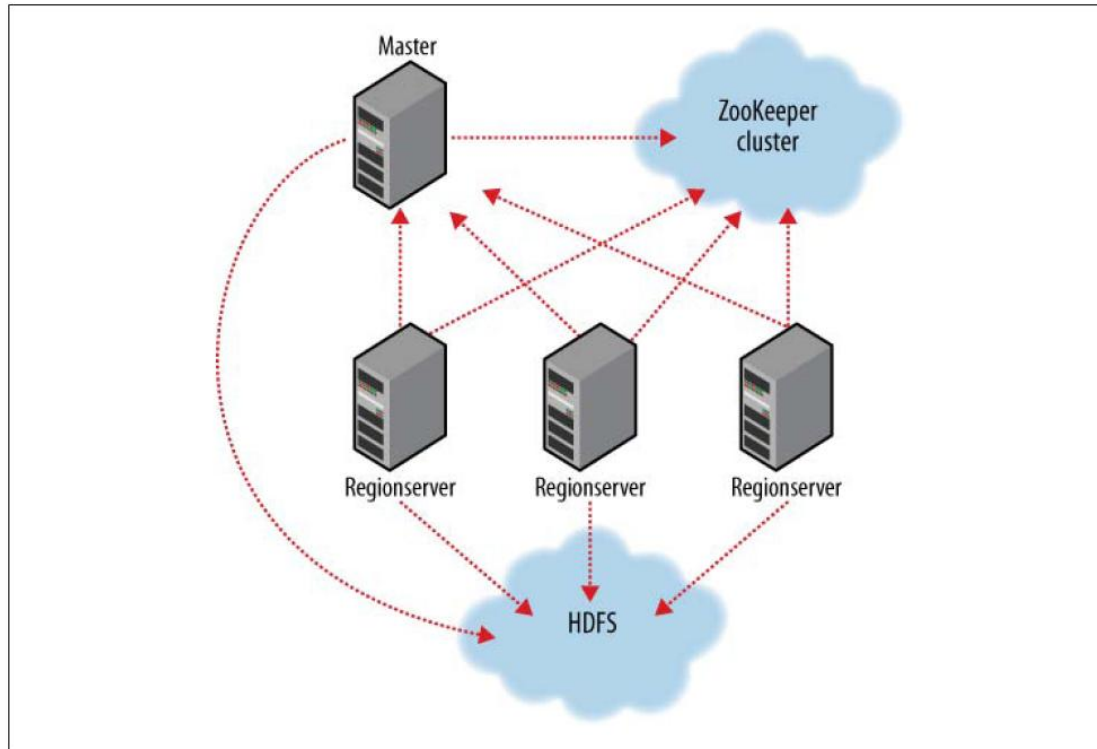


Figure 13-1. HBase cluster members

- HBase depends on **ZooKeeper** and by default it manages a ZooKeeper instance as the authority on cluster state.

HBase: Recap

- How Zookeeper helps HBase?
 - **Crash recovery:** assignment of regions is mediated via ZooKeeper in case participating servers crash mid-assignment.
 - **Location management:** HBase clients navigate the ZooKeeper hierarchy to learn cluster attributes such as server locations

What is Zookeeper?

- **Zookeeper** is a highly-available, high-performance coordination service:
 - *Key features:* scalable, distributed, configuration, consensus, group membership, leader election, naming
- It is much more than just a distributed lock server!!

What is Zookeeper?

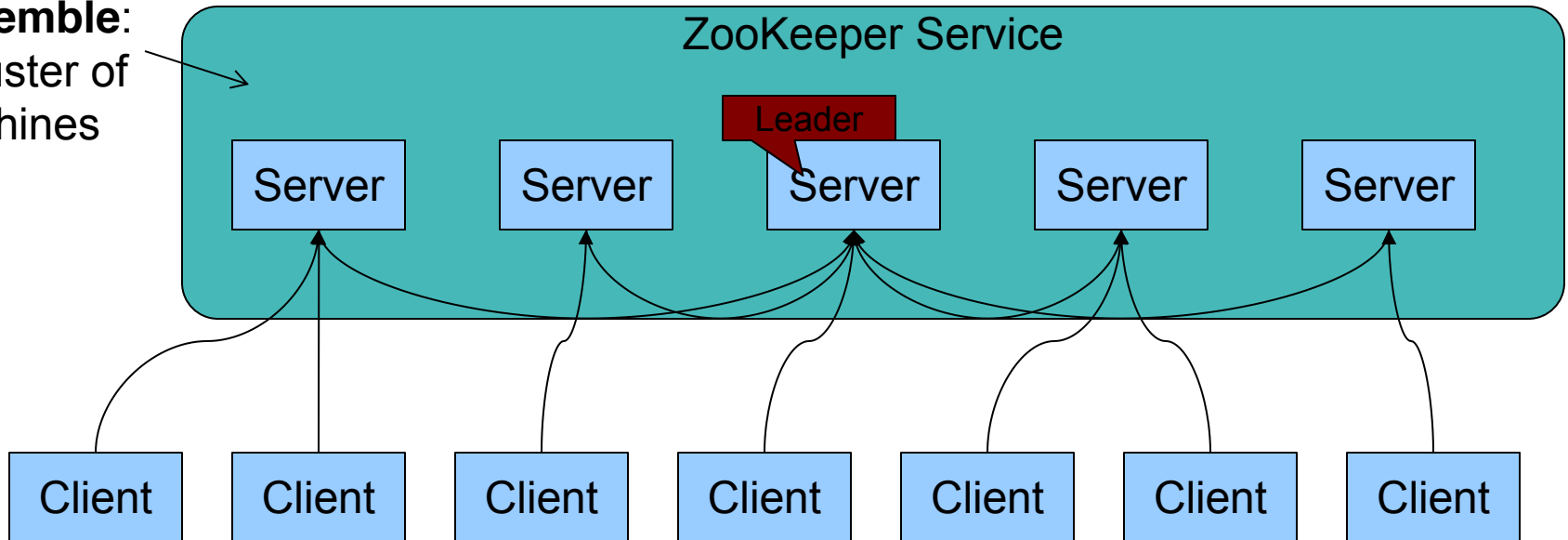
- File API without partial reads/writes
- No renames
- Ordered updates and strong persistence guarantees
- Conditional updates (version)
- Watches for data changes
- Ephemeral nodes
- Generated file names

Zookeeper Guarantees

- Clients will never detect old data.
- Clients will get notified of a change to data they are watching within a bounded period of time.
- All requests from a client will be processed in order.
- All results received by a client will be consistent with results received by all other clients.

Zookeeper Overview

Ensemble:
a cluster of
machines



- All servers store a copy of the data (in memory)
- A **leader** is elected at startup
- **Followers** service clients, all updates go through leader
- Update responses are sent when a majority of servers have persisted the change

Zookeeper Overview

- ZooKeeper uses a protocol called **Zab** that runs in two phases
 - Leader election:
 - Select a member called the *leader*, and have other machines be *followers*. This phase is finished once a majority (or *quorum*) of followers have synchronized their state with the leader.
 - Atomic broadcast:
 - All write requests are forwarded to the leader, which broadcasts the update to the followers. The leader commits the update if a majority agree
 - Consensus is designed to be **atomic**, so a change either succeeds or fails.

Zookeeper Overview

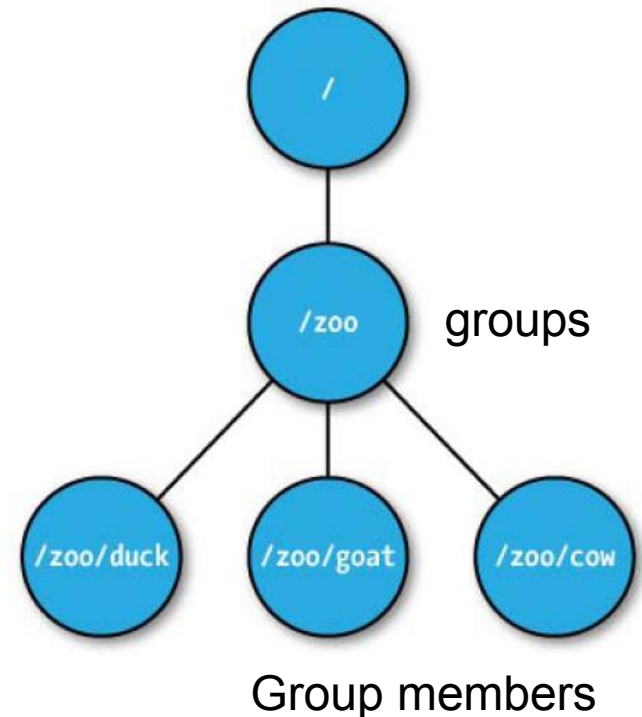
➤ Advanced notes:

- Zookeeper's Zab is different from Paxos, which is a group coordination service
- Zookeeper is different from Google Chubby Lock's service, as it is more lightweight

➤ Details: see Hunt's paper

Data Model

- A client can create a **znode**, read and write data to **znode** inside Zookeeper.
- Znodes are organized in a directory-like structure.
 - Call the intermediate nodes groups.
 - Call the leaf nodes group members.
 - Of course, it is not restricted to 3 levels (in the figure).



Data Model

- Think of Zookeeper as a “file system”
 - Hierarchical namespace: easy for management
- Znodes are not designed for general data storage. Instead, znodes map to **abstractions** of the client application, e.g., metadata
- However, znodes can store data
 - A maximum of 1MB by default

Data Model

- Zookeeper doesn't use handles to access znodes
 - Use absolute paths (e.g., /zoo/cow) directly
 - Saves overhead of calling open() or close()
- Data access is **atomic**
 - A client reading the data stored at a znode will never receive only some of the data; either the data will be delivered in its entirety, or the read will fail.

Data Model: Types of znodes

➤ Persistent (or regular) znodes

- Clients manipulate persistent znodes by creating and deleting them explicitly
- Usually represent groups, or **intermediate nodes**

➤ Ephemeral znodes

- Clients create such znodes, and they either delete them explicitly, or let the system remove them automatically when the **session** that creates them terminates (deliberately or due to a failure).
- Sessions have an associated timeout. Zookeeper considers a client faulty if it does not receive anything from its session for more than that timeout.
- Usually represent the existence of an client, i.e., **leaf node**

Data Model: Watches

- **Watches** allow clients to get notifications when a znode changes
 - A client may wait on the creation of a znode. If the znode is created, the watch is triggered
- Watchers are triggered **only once**
 - Clients need to reregister the watch to receive multiple notifications

Data Model: znode creation

➤ PERSISTENT & EPHEMERAL

- The naming of such znodes has to be **specific** and **unique**.
- If a client with a duplicated pathname is joining a group, it will be rejected.

➤ PERSISTENT_SEQUENTIAL & EPHEMERAL_SEQUENTIAL

- The naming of such znodes contains a client-defined prefix and the ***monotonic increasing, numerical postfix***: generated by the Zookeeper service.
- E.g., creating a sequential znode using the name “/zoo/fubar”, the resulting name will be: “/zoo/fubar001”
- Another client using the **same name** “/zoo/fubar” to create a sequential znode **will not be rejected**, and the resulting name will be “/zoo/fubar002”.

Client APIs

- **create(path, data, flags):**
 - Creates a znode with pathname; flag specifies the type of znode
- **delete(path, version):**
 - Deletes the znode path if that znode is at the expected version
- **exists(path, watch):**
 - Returns true if the znode with pathname exists
- **getData(path, watch) / setData(path, data, version)**
 - Reads/writes data in the znode
- **getChildren(path, watch):**
 - Returns the set of names of the children of a znode
- **sync():**
 - Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to

Installing Zookeeper

➤ Install zookeeper-3.4.3

- Set the path to “/usr/local/zookeeper-3.4.3/bin”
- Create zoo.cfg in “/usr/local/zookeeper-3.4.3/conf”

```
tickTime=2000
dataDir=/app/zookeeper
clientPort=2181
```

➤ Start the Zookeeper service

```
[hduser@localhost zookeeper-hduser]$ zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper-3.4.3/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[hduser@localhost zookeeper-hduser]$ echo ruok | nc localhost 2181
imok[hduser@localhost zookeeper-hduser]$
```

Compiling Zookeeper Programs

➤ Compiling

```
export CLASSPATH=/usr/local/zookeeper-3.4.3/lib/*:/usr/local/zookeeper-3.4.3/*:.  
javac [Program Name]
```

➤ Run

```
export CLASSPATH=/usr/local/zookeeper-3.4.3/lib/*:/usr/local/zookeeper-3.4.3/*:.  
java [Class name] [Args]
```

Zookeeper Programming

➤ **Zookeeper object:**

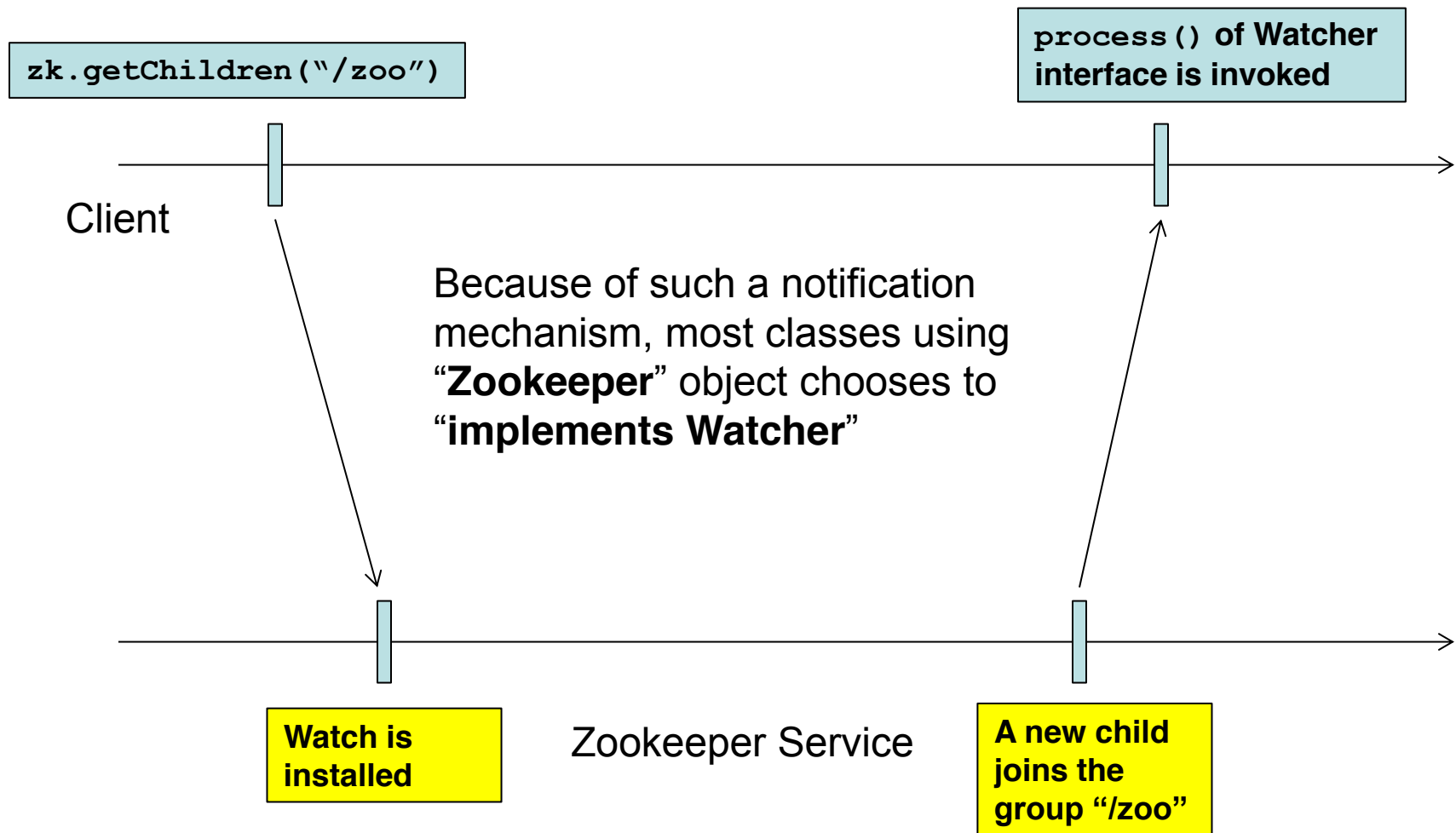
- The main object controlling every aspect of Zookeeper
- Instantiate this object when connecting to Zookeeper.
- Using the object methods, you can:
 - create/delete znodes
 - get and set znode data
 - query a znode's child list

➤ **Watcher interface:**

- It is the Zookeeper-specific event. It calls back when:
 - the concerned node is created/deleted.
 - the concerned node has its list of children changed.

➤ See demos: CreateGroup.java, DeleteGroup.java, JoinGroup.java, WatchGroup.java

Watcher Interface



Watcher Interface

➤ Watch creation

- There are 3 methods that install watches
 - `Zookeeper.exists()`
 - `Zookeeper.getChildren()`
 - `Zookeeper.getData()`

➤ Recall that the watch is installed and is triggered once only

- Need repetitive calls to one of the above 3 methods to install the watch again

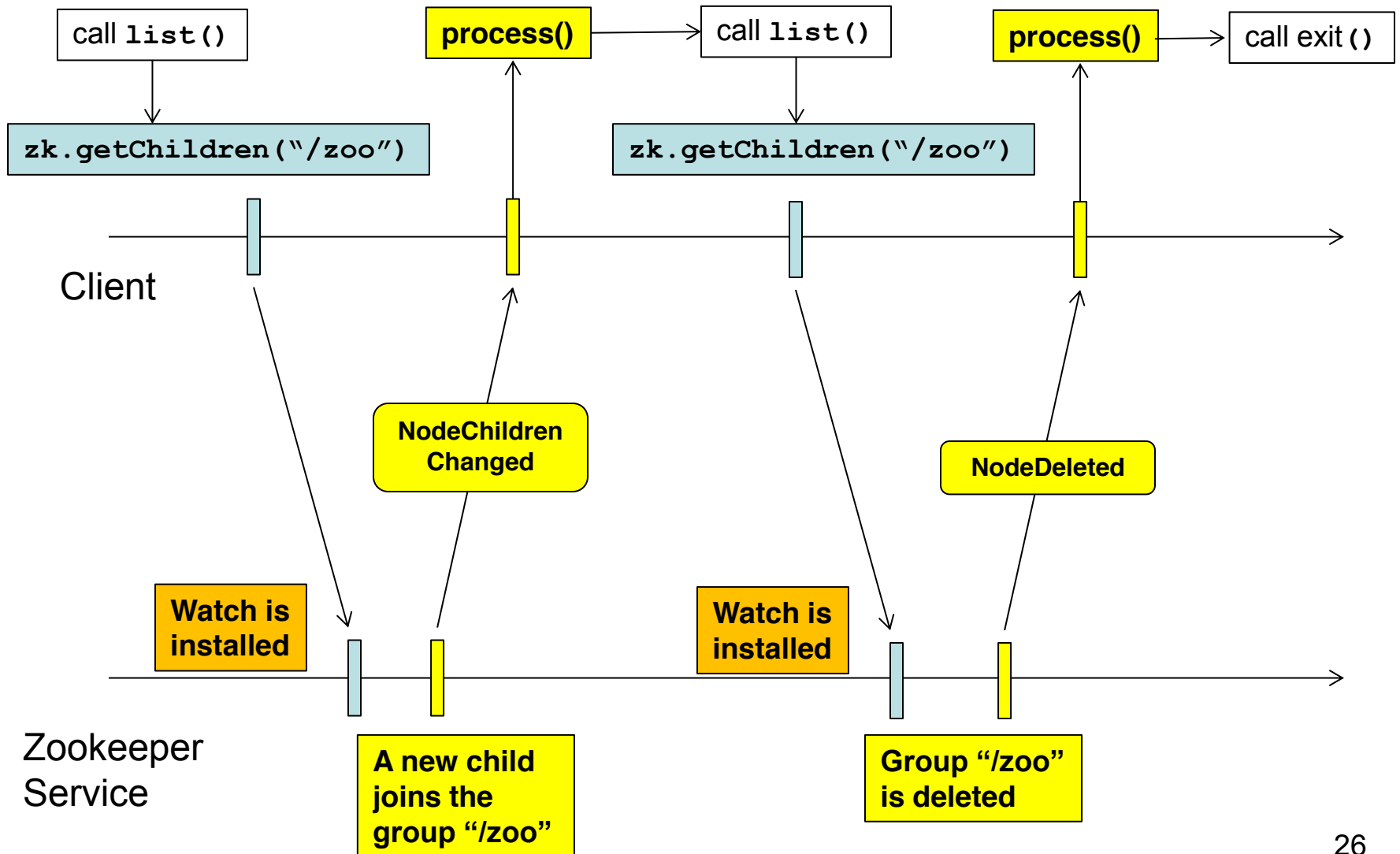
Watch Triggers

Table 14-2. Watch creation operations and their corresponding triggers

Watch creation	Watch trigger			
	create		delete	setData
	znode	child	znode	child
exists	NodeCreated		NodeDeleted	
getData			NodeDeleted	
getChildren	NodeChildren Changed		NodeDeleted NodeChildren Changed	
			NodeChildren Changed	

➤ See `Watcher.java`

Watch Triggers



Distributed Lock Service

➤ Lock and critical section:

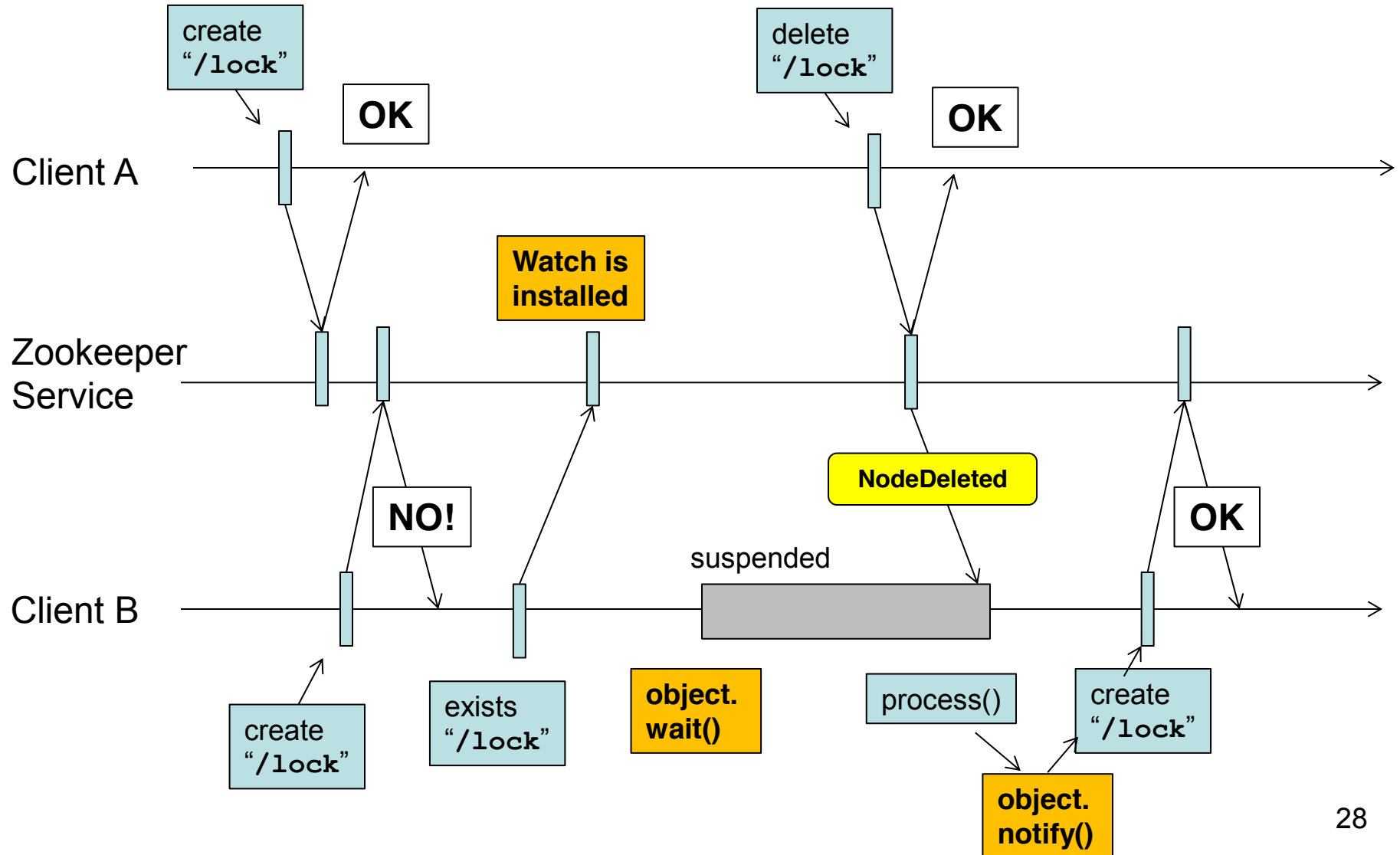
- When I'm the first one who takes the lock, others should wait until I leave.
- Others will then race in order to get the lock.

➤ Mapping to Zookeeper:

- When a client is the first one who creates **an EPHEMERAL znode**, that means “*client takes the lock*”.
- Other clients would fail to create such a znode. They **attach a Watch to the created znode**.
- When the first client dies, all waiting clients will be triggered (**EventType.NodeDeleted**).
- They can race for the lock by creating the same **EPHEMERAL znode**.

➤ See DistLock.java

Distributed Lock Service



Distributed Lock Service

- **Herd effect:** what if there are hundreds of clients waiting for the lock?
 - Many notifications are sent.
 - Many clients awake and rush for the lock together.
 - Only one of them gets the lock.
 - The situation happens again and again...
- How to avoid?
 - Using a set of “**EPHEMERAL_SEQUENTIAL**” clients!

Distributed Lock Service

- We give up the property that a released lock will be randomly held by another client.
 - New property: the client with the (numerically) **smallest pathname** will hold the lock.
- Using **exists()**, a client will only watch the deletion of the client that is just smaller than its pathname.



Distributed Lock Service

➤ Pseudo-code of simple lock without herd effect:

- Lock()

```
1  n = create(l + "/lock-", EPHMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for watch event
6  goto 2
```

- Unlock()

```
1  delete(n)
```

Distributed Lock Service

➤ Pseudo-code of read/write lock:

- Write lock

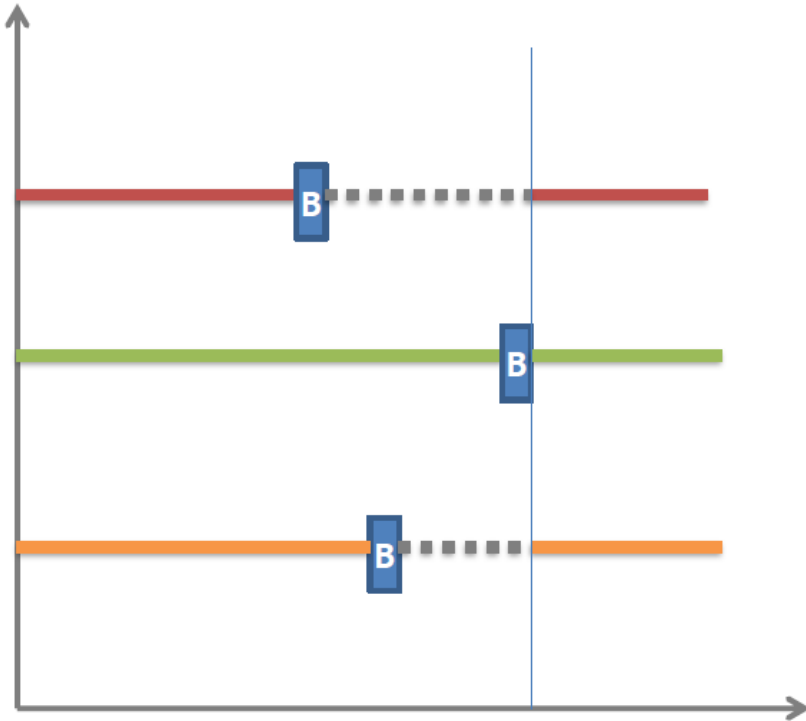
```
1  n = create(l + "/write-", EPHMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

- Read lock

```
1  n = create(l + "/read-", EPHMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 3
```

Any herd effect?

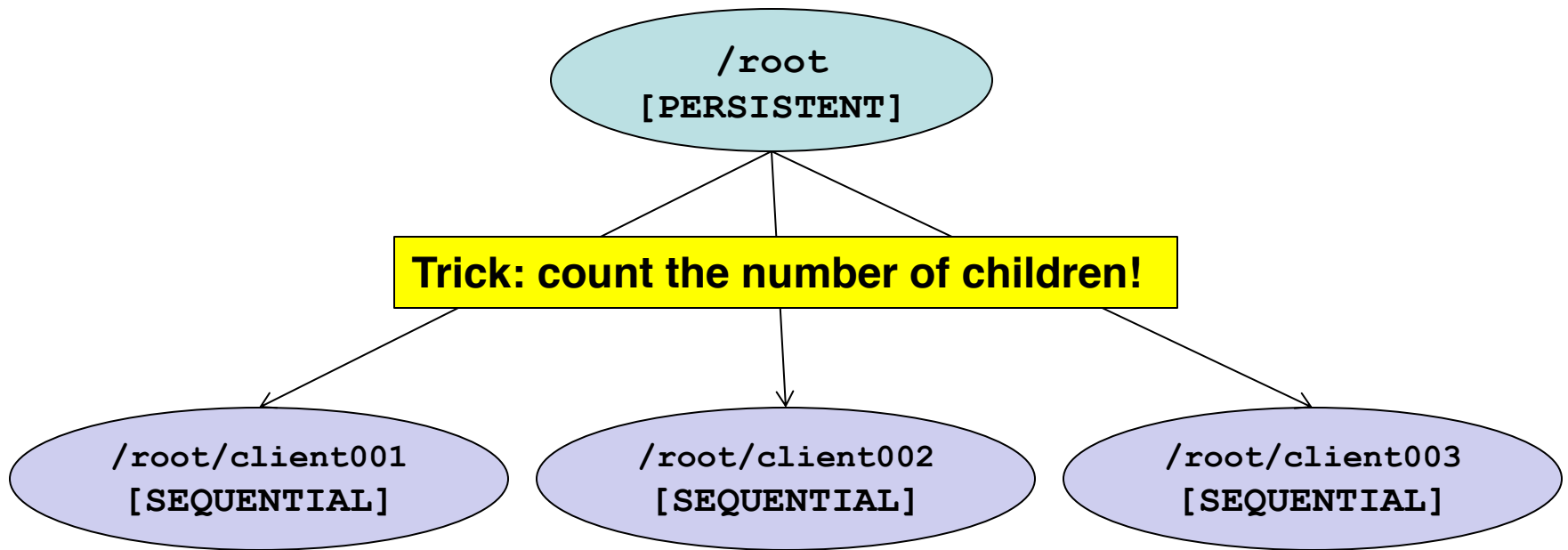
Distributed Barrier



- **Barrier** is a common programming construct in **multi-threading** programming.
- It aims to **synchronize progress** among threads.
- A barrier API usually asks for the number of threads that should wait, e.g., 3 in the example.
- For 3 threads have invoked the barrier call, they will be unblocked and progress together.

Distributed Barrier

- How to realize using Zookeeper?
 - Use “**EPHEMERAL_SEQUENTIAL**” clients



Distributed Barrier

- How to realize using Zookeeper?
 - Use “**EPHEMERAL_SEQUENTIAL**” clients

Client:

```
while (true) {  
    zk.getChildren(...)  
    if (# of children < threshold)  
        object.wait()  
    else  
        break  
}  
  
// ...
```

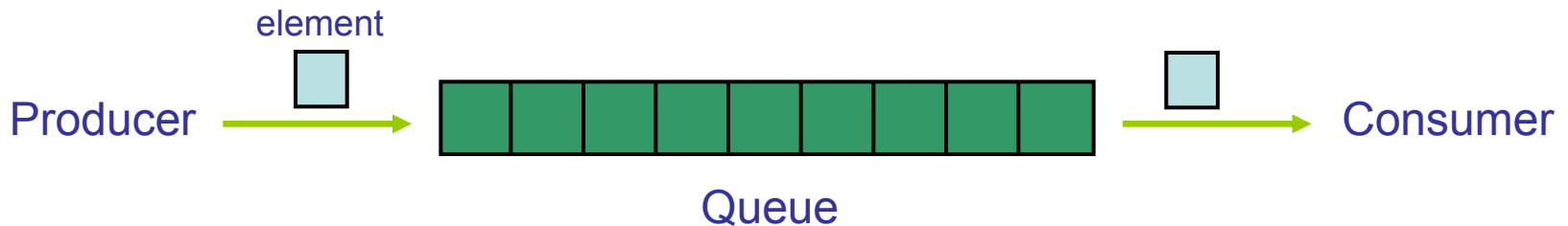
process()

object.notify()

*Triggered when
of children
changes*

Distributed Producer-Consumer Queue

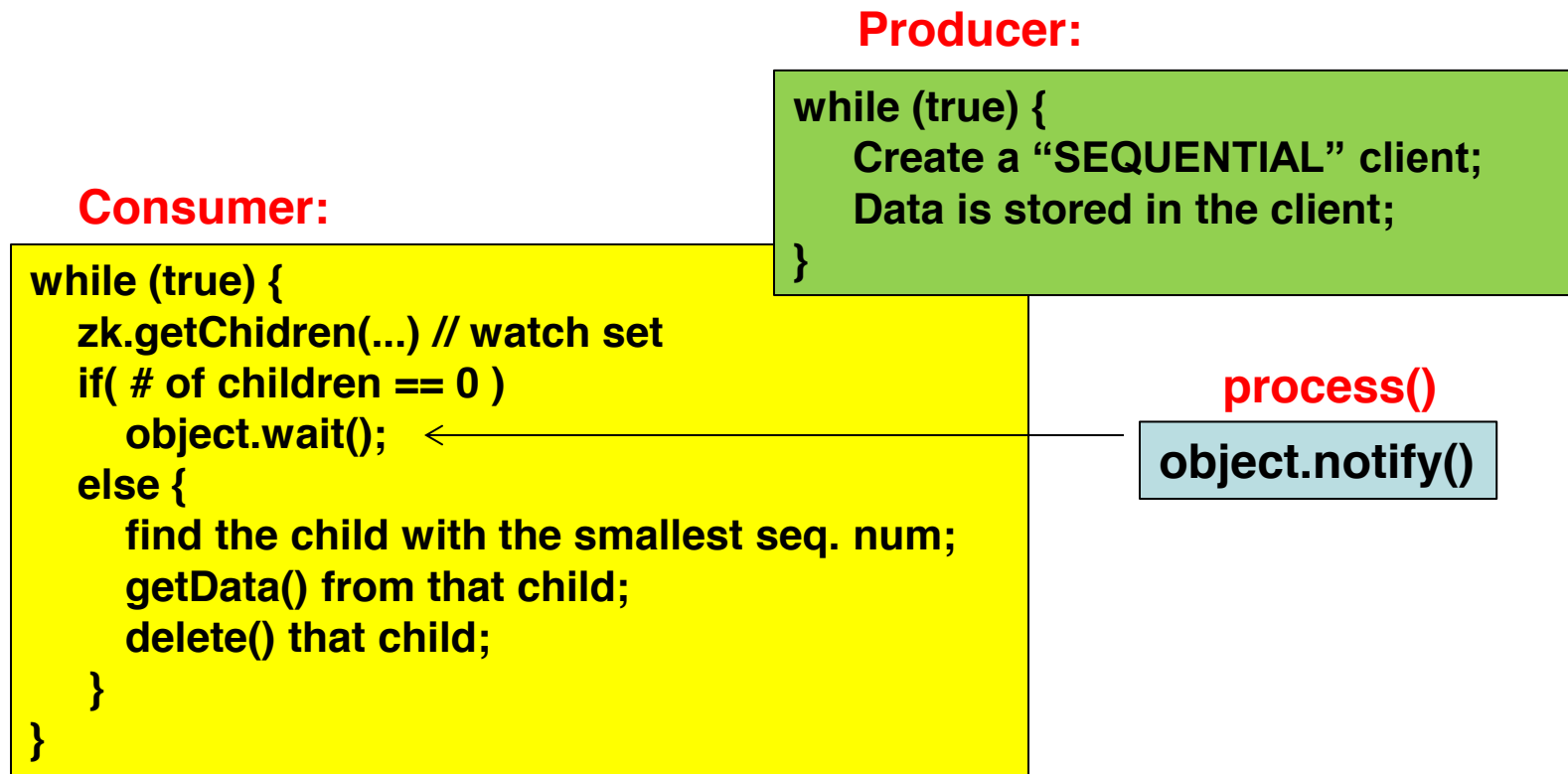
- Requirements of a producer-consumer queue
 - Producer inserts elements when buffer is not full
 - Consumer extracts elements when buffer is not empty
 - First-in-first-out (FIFO): inserted elements and extracted elements in the same order



- Here, we consider an **unbounded** queue
 - Queue is never full

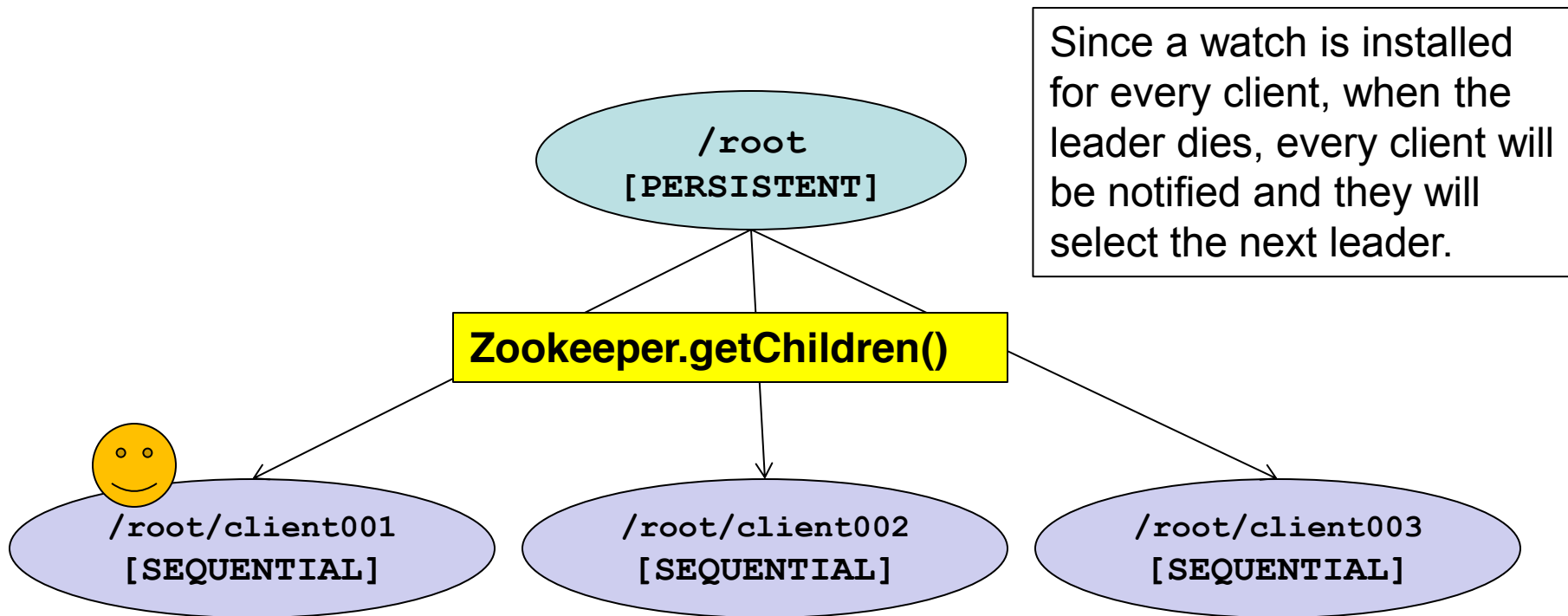
Distributed Producer-Consumer Queue

- Unbounded producer-consumer queue:



Distributed Leader Election

- We pick the client with the smallest sequential number to be the leader



Sublties: Configuration Service

- Goal: a highly-available service that stores key-value pairs.
 - Treat this as a highly-available, robust file system.
- •API calls:
 - Zookeeper object:
 - `getData(path, watcher);`
 - `setData(path, byte_array, -1);`

Use Cases in Yahoo!

➤ Fetching service

- Master coordinates page fetching process
- Master provides fetchers with configuration
- Fetchers inform master their health and status
- Zookeeper is used to manage configuration and elect masters

➤ Katta

- Distributed indexer
- Zookeeper is used to track group membership, elect master, and manages configuration

➤ Yahoo! Message Broker

- A distributed publish-subscribe service that manages different topics of messages
- Zookeeper is used to manage distribution of topics, dealing with failures of machines, and control system operations

Use Cases in Yahoo!

➤ Summary:

- » Leader Election
- » Group Membership
- » Work Queues
- » Configuration Management
- » Cluster Management
- » Load Balancing
- » Sharding (database partitioning)

Conclusions

- Zookeeper is a generic platform for distributed computation
 - You can view it as a programming model like MapReduce
- Zookeeper is easy to learn and has a great potential!