**ZooKeeper—Introduction**

ZooKeeper is a coordination service of Apache that helps manage the activities of distributed applications.

It is highly scalable.

It also has an open source library of recipes for distributed systems, such as, leader selection, exclusive locks, book keeper, and so on. These recipes facilitate building relations between distributed processes and applications.

One of the main features of ZooKeeper is that it helps handle partial failures in distributed systems

**Distributed Applications**

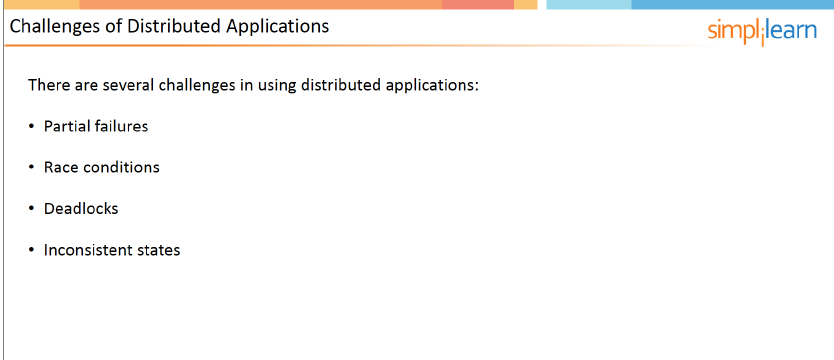
Before you learn about ZooKeeper, let us understand what distributed applications are and what sort of problems arise while using them.

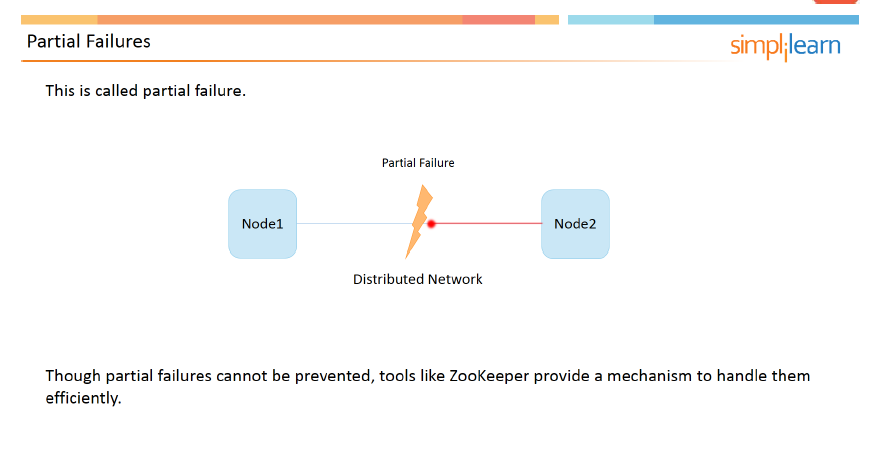
Distributed applications are run on multiple machines in parallel. They function by following the divide and conquer principle. This means that they divide large jobs into smaller jobs, which are then run in parallel on multiple machines.

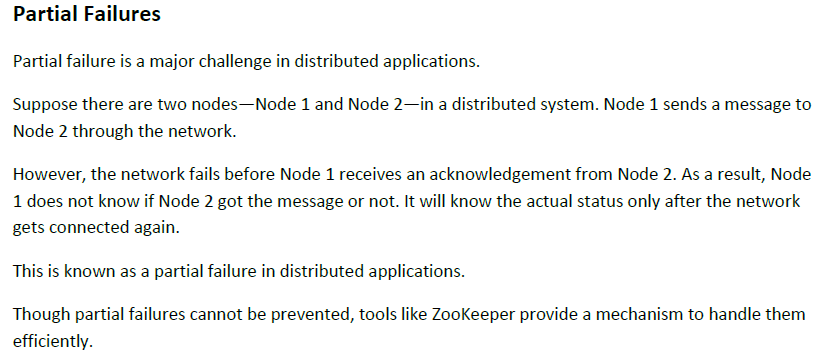
They are horizontally scalable if adding more machines reduces the execution time. For example, if 10 machines do a job in 10 hours, adding 10 more machines may halve the execution time to 5 hours.

They are vertically scalable if increasing the memory, CPU, or other resources of each machine reduces the execution time. For example, increasing the memory from 100 GB to 256 GB may reduce the execution time of a job from 10 hours to 5 hours.

The diagram shows three machines connected by a network switch. An application can be distributed to run on them in parallel. These machines are also referred to as nodes in a cluster.







**Race Conditions**

A race condition takes place in distributed applications when multiple machines are waiting for one resource to become free.

Suppose there are four different nodes in a distributed system.

Let us assume that currently only Node1 is using the resource. So, it has an exclusive lock on the resource.

All the other nodes—from Node 2 to Node 4—are waiting for the resource to become available.

When Node 1 releases the resource, Nodes 2 to 4 race to acquire the resource.

Only one of them succeeds, while the others go back to the waiting state.

This process continues till all the nodes get the resource. This is called a race condition.

**Deadlocks**

Deadlocks occur when there is a cyclic dependency on resources.

There are two machines—Machine 1 and Machine 2; and there are two resources—resource A and resource B. Machine 1 has locked resource A and is waiting to lock resource B. At the same time, machine 2 has locked resource B and is waiting to lock resource A.

Since none of the locks can be acquired or released, it leads to a deadlock. To resolve a deadlock, one of the processes has to be killed and redo the processing.

Detecting deadlocks are generally CPU-intensive and expensive operations.

**Inconsistencies**

Inconsistencies take place when changes are not propagated to all the machines in a distributed system.

For example, let us consider a salary data which is initially equal to 100. This data is replicated to both the machines.

The data is later updated to 200. This change is first propagated to Machine 1. The value stored on it is now changed to 200. However, due to some failure, this update is not propagated to Machine 2. So, the value stored on it remains as 100.

In such a situation, if process A reads from Machine 1, while process B reads from Machine 2, each process will get two separate values of the salary data.

This leads to inconsistencies.

**ZooKeeper Characteristics**

ZooKeeper helps coordinate distributed applications.

It provides a very simple interface. It is expressive, which means that it provides basic blocks that can be used to build larger applications. It is also highly available and reliable. This is because it runs on multiple servers at the same time. So, even if a few servers fail, it continues to function.

To have a fault-tolerance for ‘n’ machine failures, it is recommended to have two ‘n’ plus one machines running the ZooKeeper service. For example, if you want to have a fault tolerance of 3 machines, then you should have ZooKeeper running on 7 machines.

Another feature of ZooKeeper is that it has loosely coupled interactions. Machines using it do not have to know each other. ZooKeeper is actually an extensive library of recipes for distributed coordination

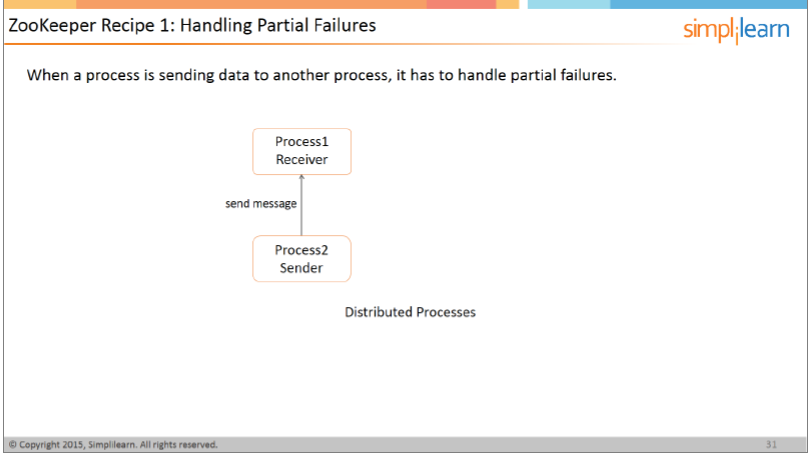
**Types of Znodes**

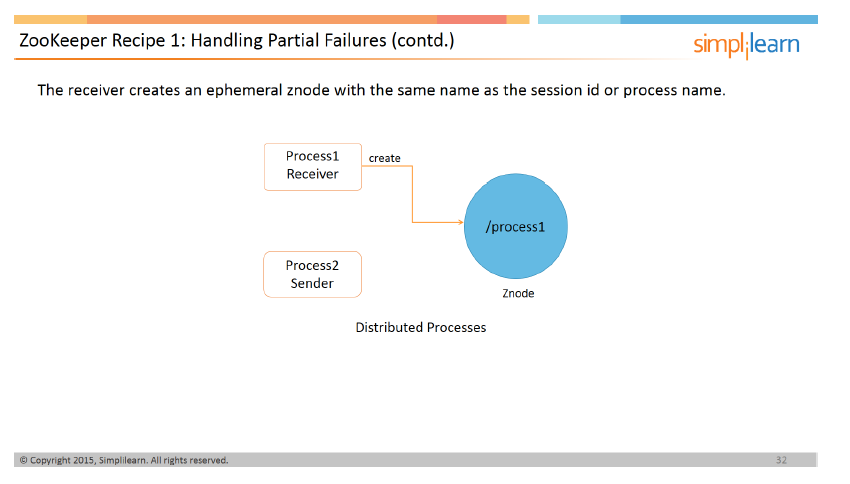
ZooKeeper znodes can be of two types: Persistent znodes and ephemeral znodes. Note that when a znode is created, its type is specified and it cannot be changed later.

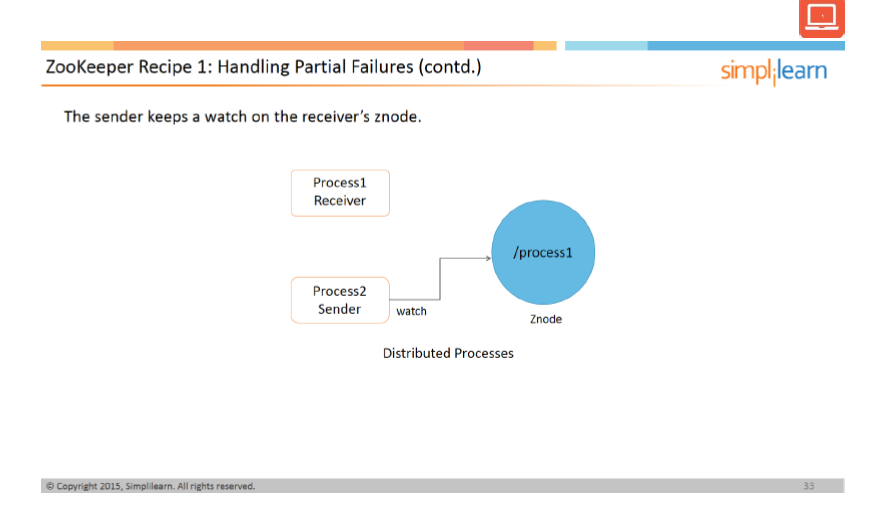
Persistent znodes are permanent and have to be deleted explicitly by the client. They stay even after the session that created the znode is terminated.

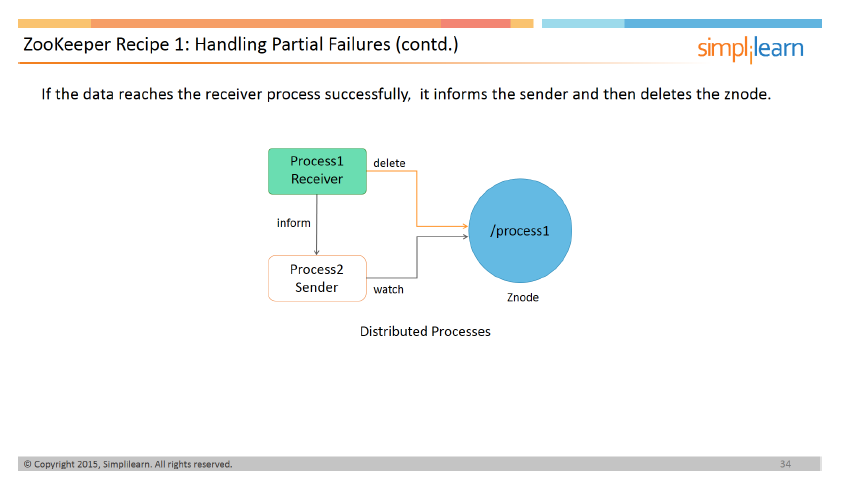
Ephemeral znodes are temporary. These znodes are deleted automatically when the client session creating them ends. Ephemeral znodes are used to detect the termination of a client.

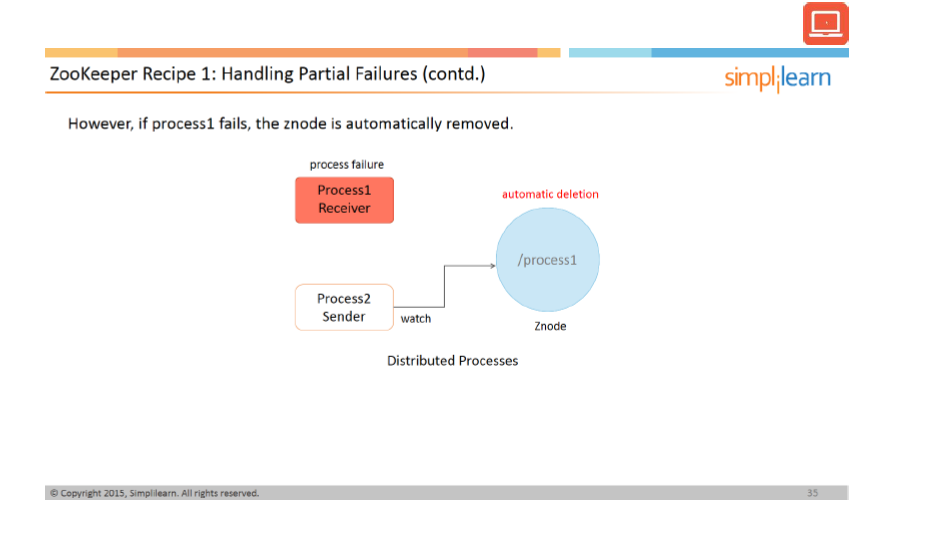
Alerts, known as watch, can be set up to detect the deletion of the znode.

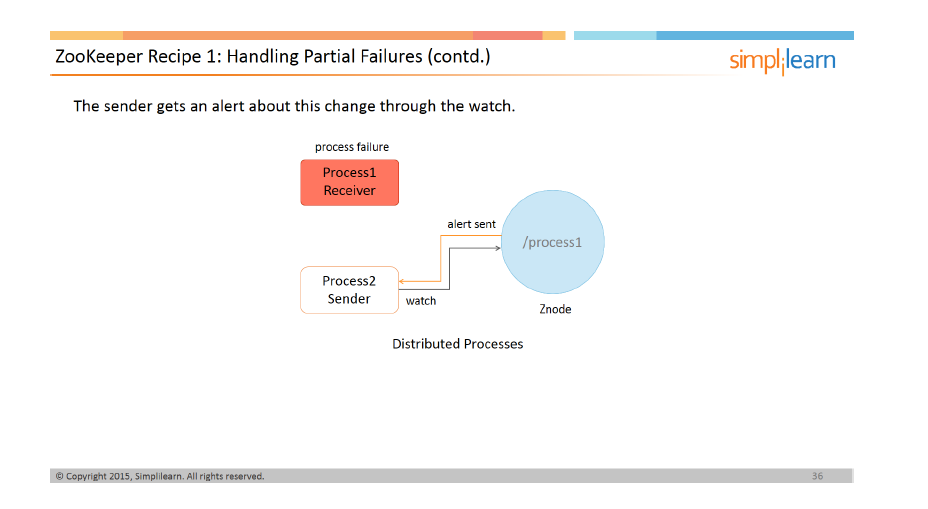


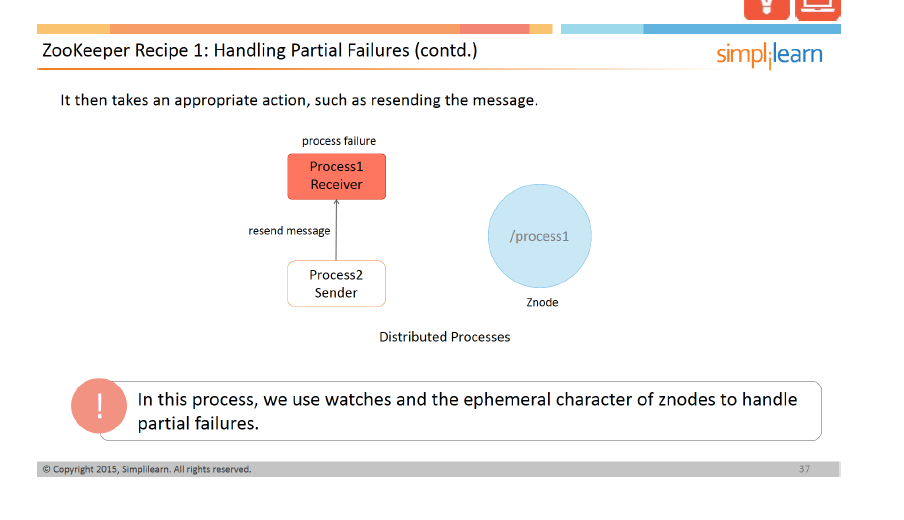


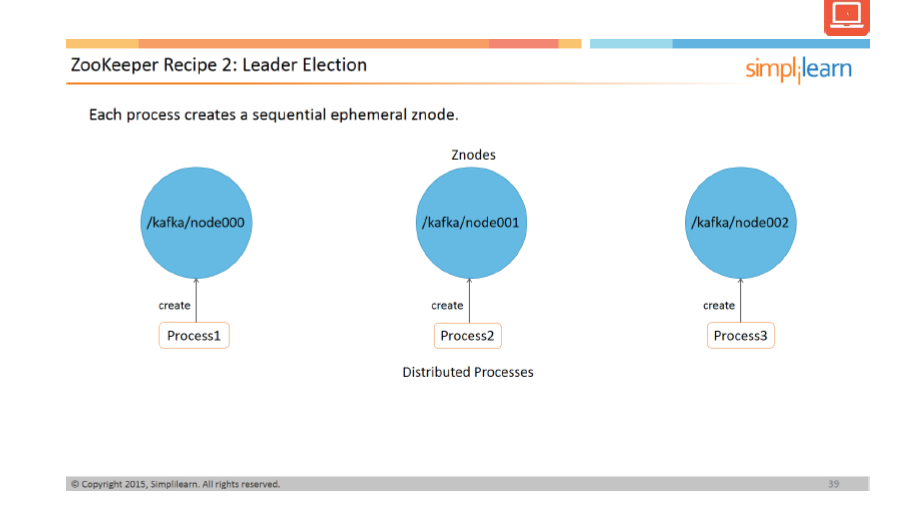


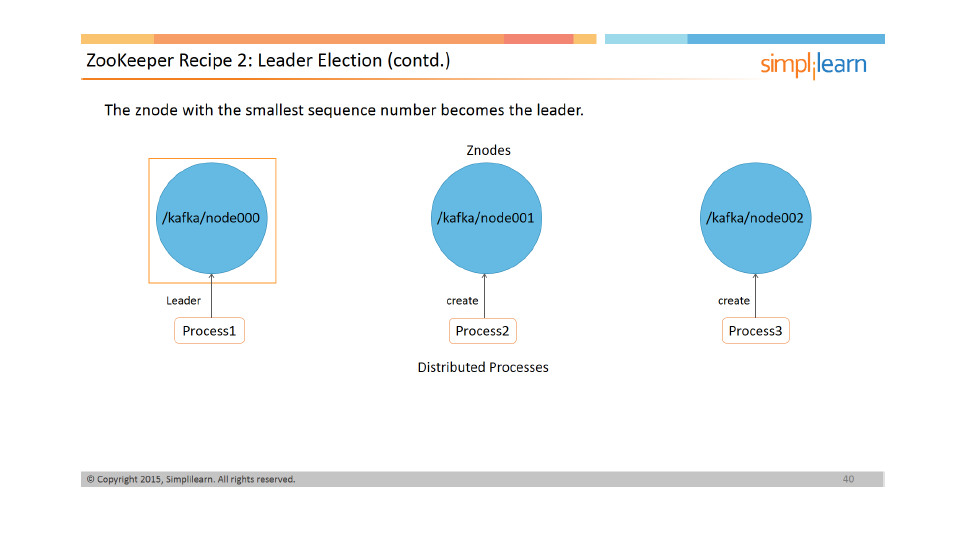


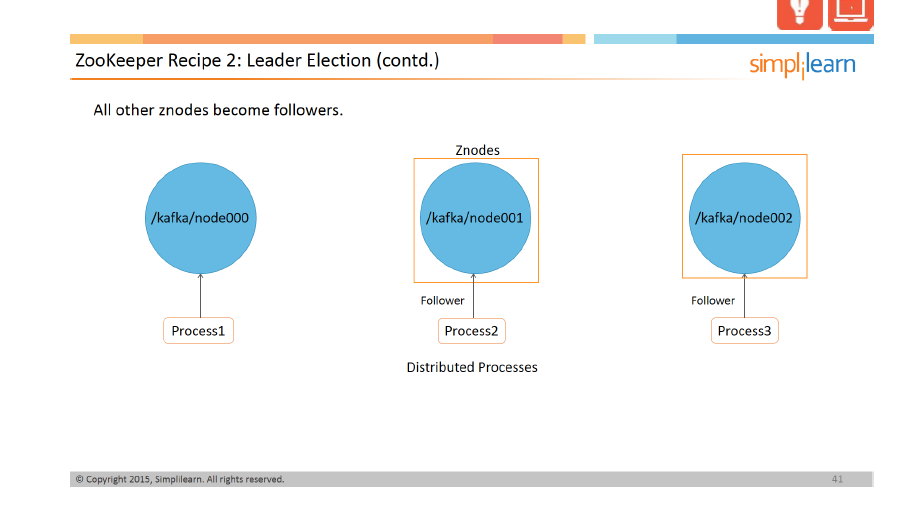


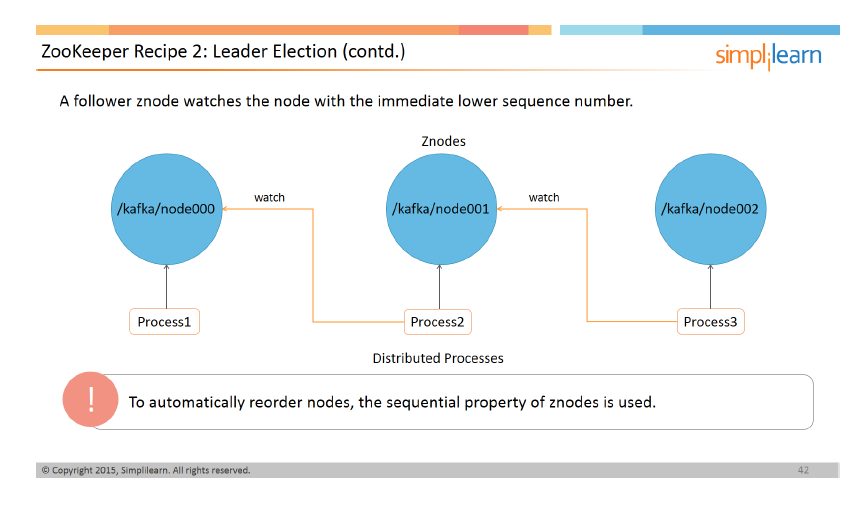












**ZooKeeper Recipe 2: Leader Election**

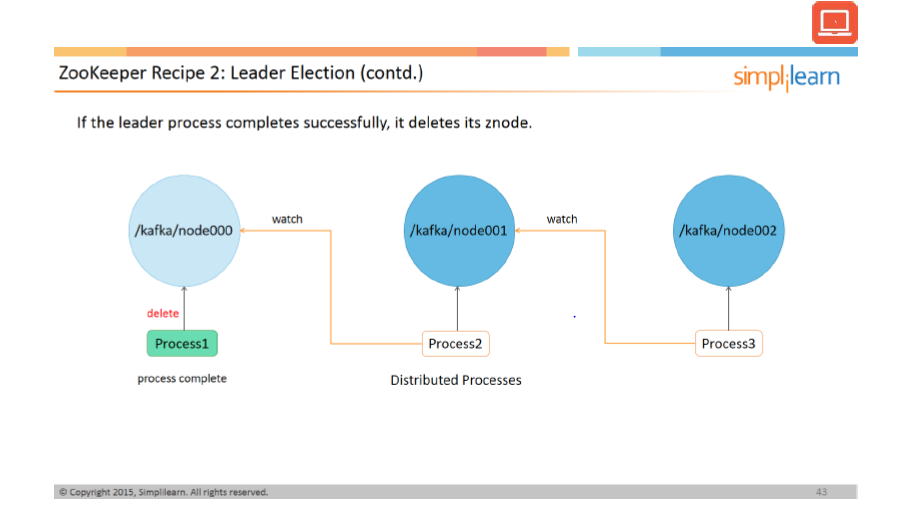
Let us now look at another ZooKeeper recipe.

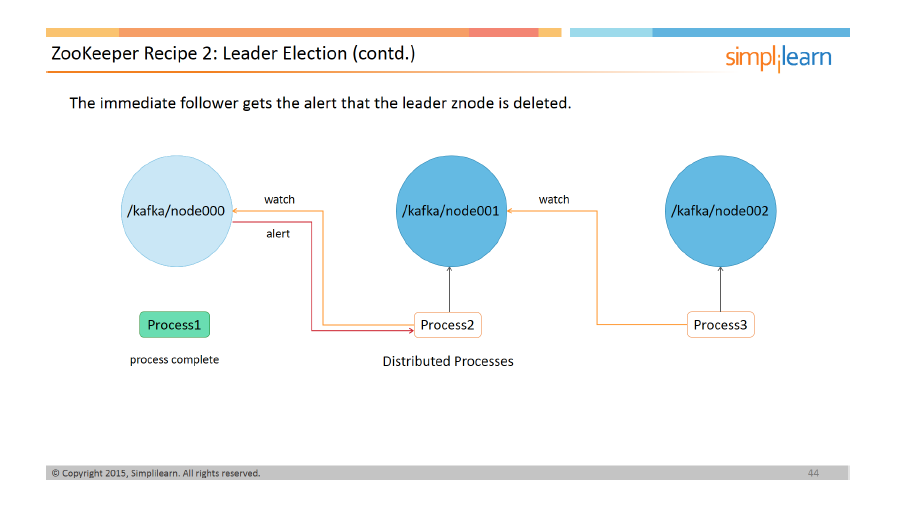
Leader election uses ephemeral sequential nodes to create an automatic node order. Let us see how the three processes—Process 1, Process 2, and Process 3—use the leader election mechanism.

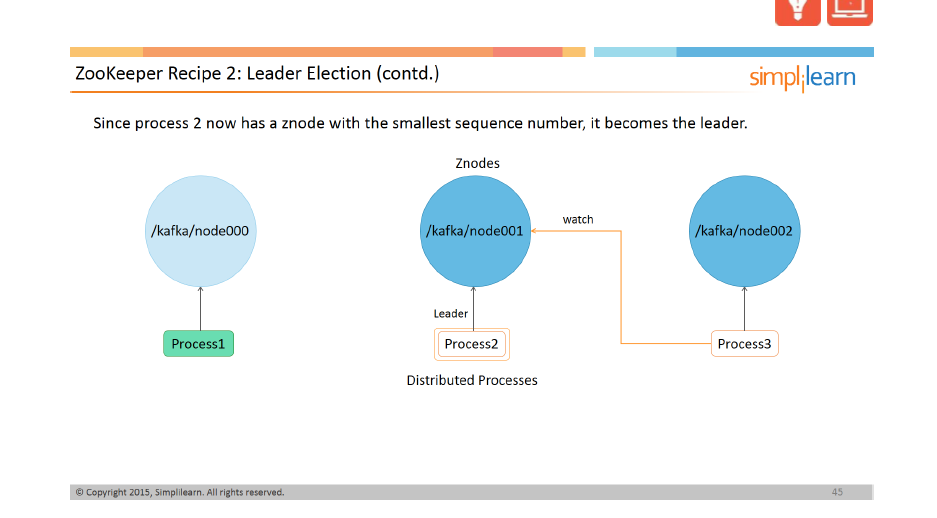
Each process creates a sequential ephemeral znode under parent ‘/kafka’ with the prefix node. Thus Process 1 gets the znode ‘/kafka/node000’, Process 2 gets the znode ‘/kafka/node001’, and Process 3 gets the znode ‘/kafka/node002’.

The znode with the least sequential value is chosen as the leader. Process 2 and Process 3, in turn, become followers. So, Process 2 watches its immediate preceding sequence number ‘/kafka/node000’. Likewise, Process 3 watches ‘/kafka/node001’.

To automatically reorder nodes, the sequential property of znodes is used.







**ZooKeeper Recipe 2: Handling Partial Failures (contd.)**

If Process 1 completes successfully, it deletes the znode ‘/kafka/node000’. This alerts Process 2 as it has a watch on the leader znode.

Now Process 2 becomes the leader as it has the lowest sequence value. Process 3 remains a follower as the preceding znode is not modified.

In this case, the watch feature serves to change the leader. This process helps you avoid a race condition. Note that only Process 1 and Process 2 are handling the leader change