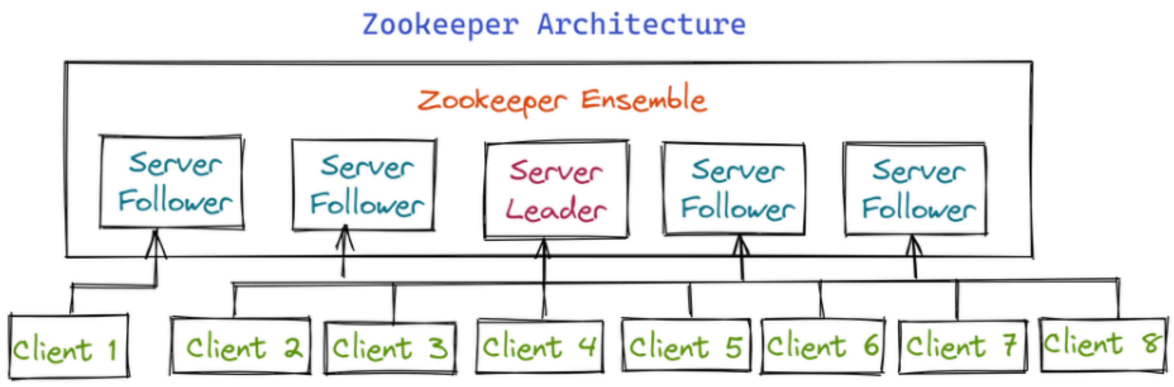
Raft is a library but Zookeeper is a general purpose standalone service.
which has an API.



Zookeeper runs on top of Zab which is similar to Raft and leader based.
Linearizability of writes is provided by Zab.
The type of linearizability which Zookeeper provides is A-linearizability (asynchronous linearizability). We allow a client to have multiple outstanding operations we choose to guarantee FIFO order.
However reads are processed by each server locally and interaction with Zab is not required during reads. Zoo Keeper processes read requests locally at each replica. This allows the service to scale linearly as servers are added to the system.

Zookeeper Guarantees

Linearizable writes: all requests that update the state of ZooKeeper are serializable and respect precedence.
FIFO client order: all requests from a given client are executed in the order that they were sent by the client.

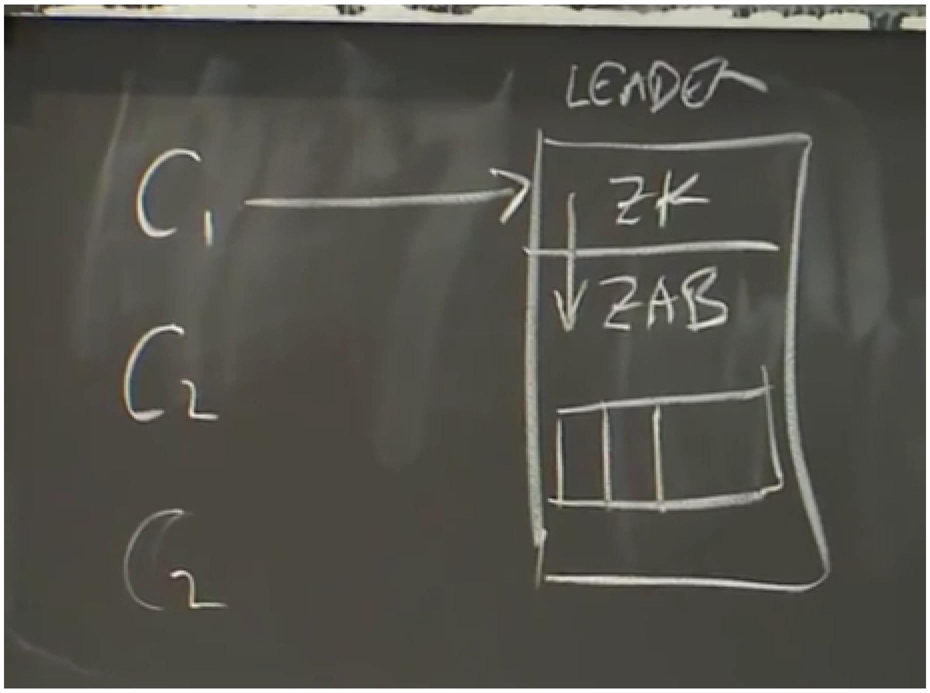
With ZooKeeper, the new leader can designate a path as the ready znode; other processes will only use the configuration when that znode exists. The new leader makes the configuration change by deleting ready, updating the various configu ration znodes, and creating ready. All of these changes can be pipelined and issued asynchronously to quickly update the configuration state.

The Guarantees in Action

If a change happens while a client is reading configuration:

1. **With Watches**: The client will receive notification that "ready" changed before it completes reading the configuration, allowing it to restart
2. **With Version Checking**: The client can detect if "ready" changed its version during the reading process
3. **With Multi Operation**: The client can perform an atomic operation that will fail completely if "ready" changes during execution

Clients cache important metadata like the current leader, it refreshes this metadata using Zookeeper's watch mechanism with which a client can watch for an update to a data object and receive a notification on an update to this object.
Watches are one-time triggers associated with a client session. They are unregistered once triggered or the session closes. Watches indicate that a change has happened, but do not provide the change.
For example, if a client issues a getData(‘‘/foo’’, true) before “/foo” is changed twice, the client will get one watch event telling the client that data for “/foo” has changed.



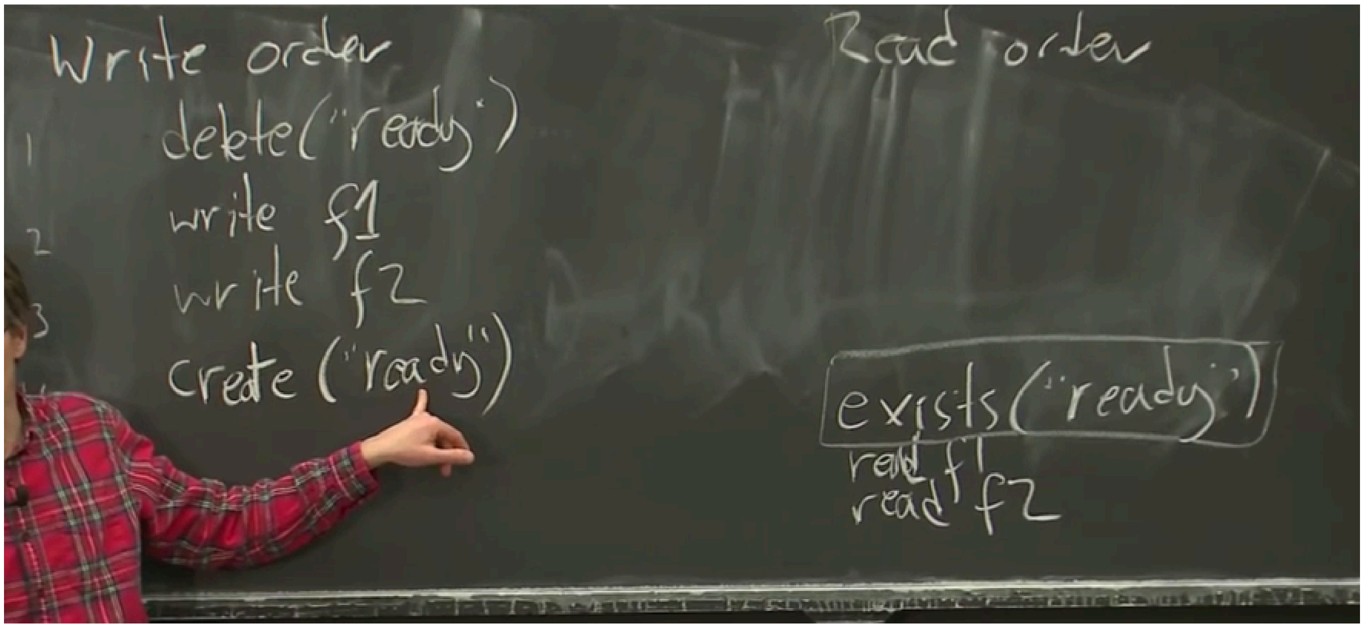
Zab will replicate the client requests.
As we add more servers, will our service get faster??
Nope, cuz the leader is anyways a bottleneck and will have to do more work if there are more servers.

ZOOKEEPER GUARANTEES

1. LINEARIZABLE WRITES
2. FIFO CLIENT ORDER- write order can be specified by the client

3. The writes all go through the leader, whereas the reads go through the replicas, when the replicas reply to reads they attach an id(which is sort of the index in the log up till which the client has read) to the replies which the client keeps track of and attaches it into subsequent reads.
4. Does not guarantee a client reads the latest data(the reads are not linearizable across multiple clients)-> READS can be stale.

ZOOKEEPER CONFIGURATION



DESIGN OF THE API OF ZOOKEEPER

Zookeeper allows us to make a fault tolerant TEST-AND-SET service, a configuration information service for multiple servers, helps us elect a master.
Zookeeper maintains a file-directory like structure, so say if app1 had a file x, for zookeeper this would be app1/x.
The hierarchical namespace is useful for
allocating subtrees for the namespace of different applications and for setting access rights to those subtrees.

For example, in a leader-based application, it is useful for an application server that is just starting to learn which other server is currently the leader. To accomplish this goal, we can have the current leader write this information in a known location in the znode space.

These files and directories are called ZNODES.
ZNODES can be :-
1.REGULAR
2.EPHEMERAL
3.SEQUENTIAL

Znodes store data and can have children(except for ephemeral nodes).

ZooKeeper Node Types

ZooKeeper nodes (znodes) come in three main types, each with specific characteristics and use cases:

1. REGULAR (Persistent) Nodes

Regular znodes are persistent by default, meaning:

- They remain in the ZooKeeper namespace until explicitly deleted
- They survive server restarts and client session expirations
- Ideal for storing configuration data, status information, or any data that should persist long-term

Example use cases:

- Application configuration settings
- Service registry information
- Distributed locks that need to survive client failures

2. EPHEMERAL Nodes

Ephemeral znodes are temporary and tied to client sessions:

- Automatically deleted when the creating client's session ends (disconnection or timeout)
- Cannot have children nodes
- Perfect for indicating presence or temporary state

Example use cases:

- Service discovery (showing which services are currently available)
- Leader election mechanisms
- Temporary locks that should be released when a client fails

3. SEQUENTIAL Nodes

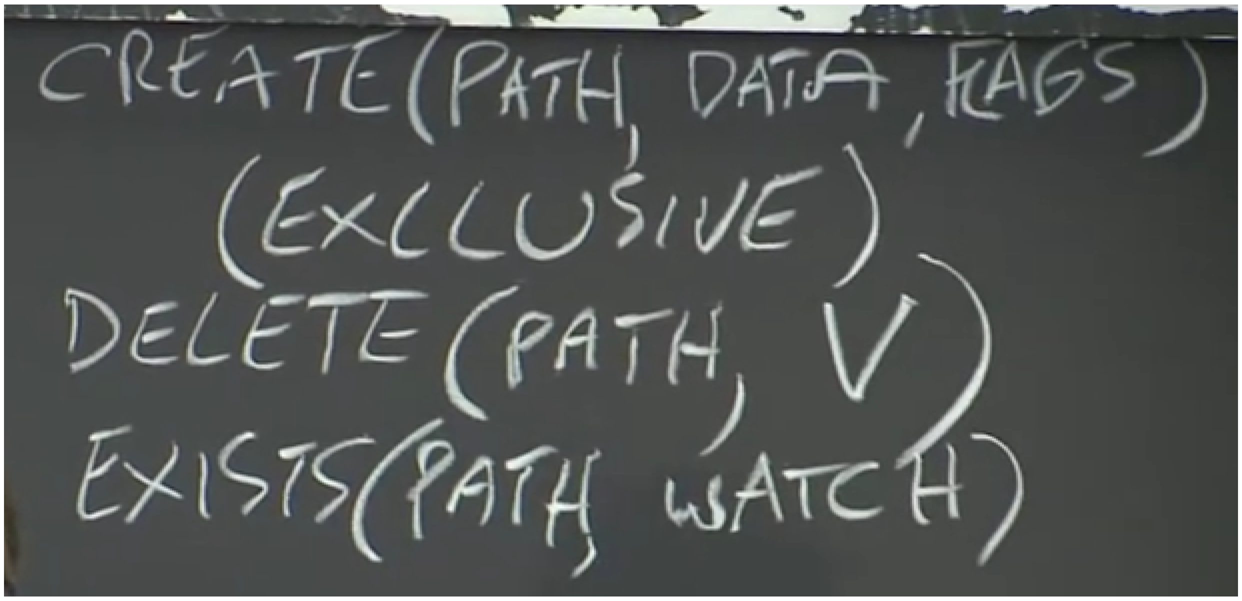
Sequential znodes have a unique auto-incrementing sequence number appended to their name:

- Can be either persistent or ephemeral

- Created with a sequence number that is greater than any previously created sequential node
- Provides natural ordering among siblings

Example use cases:

- Implementing distributed queues
- Managing distributed processes in order
- Creating unique, time-ordered identifiers


CREATE(PATH, DATA, FLAGS)
(EXCLUSIVE)
DELETE(PATH, V)
EXISTS(PATH, WATCH)

APIS

ZooKeeper APIs

Operations Shown in the Image

1. CREATE(PATH, DATA, FLAGS)

- **Purpose**: Creates a new znode at the specified path
- **Parameters**:
    - PATH: The location in the ZooKeeper hierarchy
    - DATA: The information to store in the node (up to 1MB)
    - FLAGS: Node type and properties
- **(EXCLUSIVE)** indicates this operation ensures the node doesn't already exist

2. DELETE(PATH, V)

- **Purpose**: Removes a znode from the hierarchy

- **Parameters**:
  - PATH: The znode location to delete
  - V: Version number for optimistic concurrency control (prevents deleting if the node was modified)

3. EXISTS(PATH, WATCH)

- **Purpose**: Checks if a znode exists
- **Parameters**:
  - PATH: The znode location to check
  - WATCH: Optional flag to set a watch for changes to this node

Additional APIs You Mentioned

4. SETDATA(PATH, DATA, V)

- **Purpose**: Updates the data stored in an existing znode
- **Parameters**:
  - PATH: The znode location to update
  - DATA: New data to store
  - V: Version number for concurrency control (ensures you're updating the version you expect)

5. GETDATA(PATH, WATCH)

- **Purpose**: Retrieves data from a znode
- **Parameters**:
  - PATH: The znode location to read
  - WATCH: Optional flag to register for notifications when data changes

`getChildren(path, watch)`

This method provides two important functions:

Purpose:

- Retrieves the names of all child znodes under the specified path
- Optionally sets up a watch for changes to the children of that znode

Parameters:

- `path`: The znode path whose children you want to list (e.g., "/app/services")
- `watch`: Boolean flag that determines whether to set a watch on the children

`sync(path)`

This method ensures consistency in a distributed system:

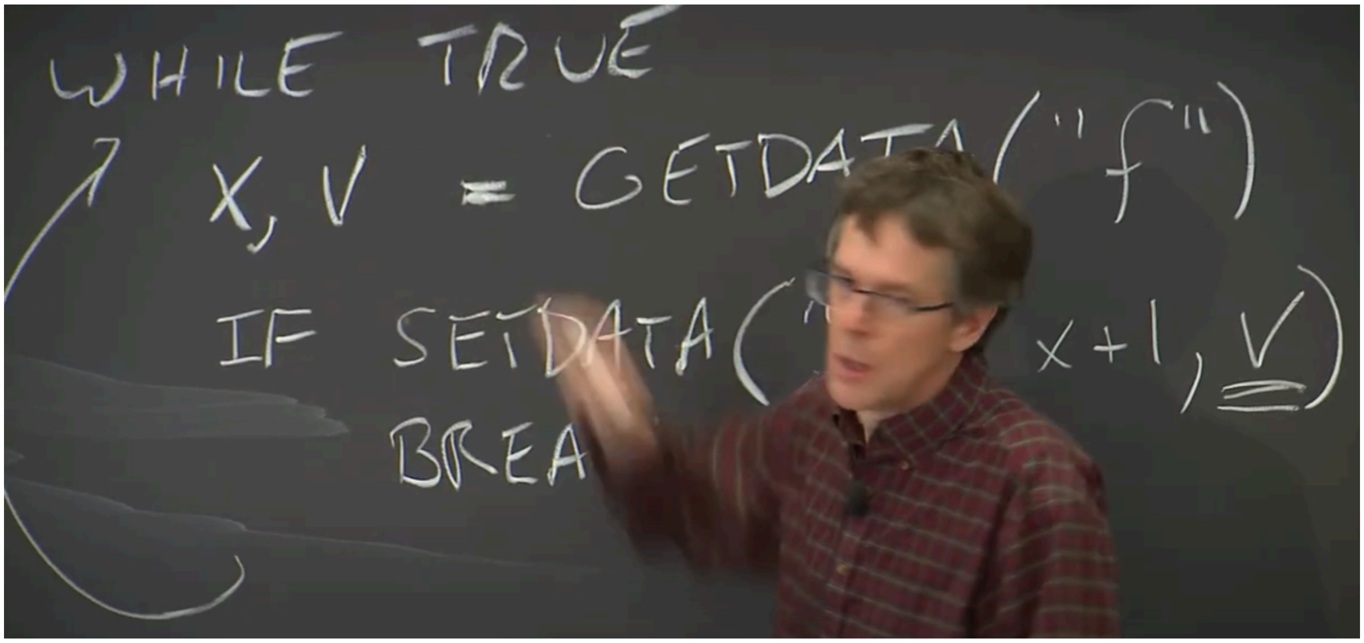
Purpose:

- Forces synchronization of a client's view with the ZooKeeper service
- Ensures all pending updates have been applied at the server the client is connected to

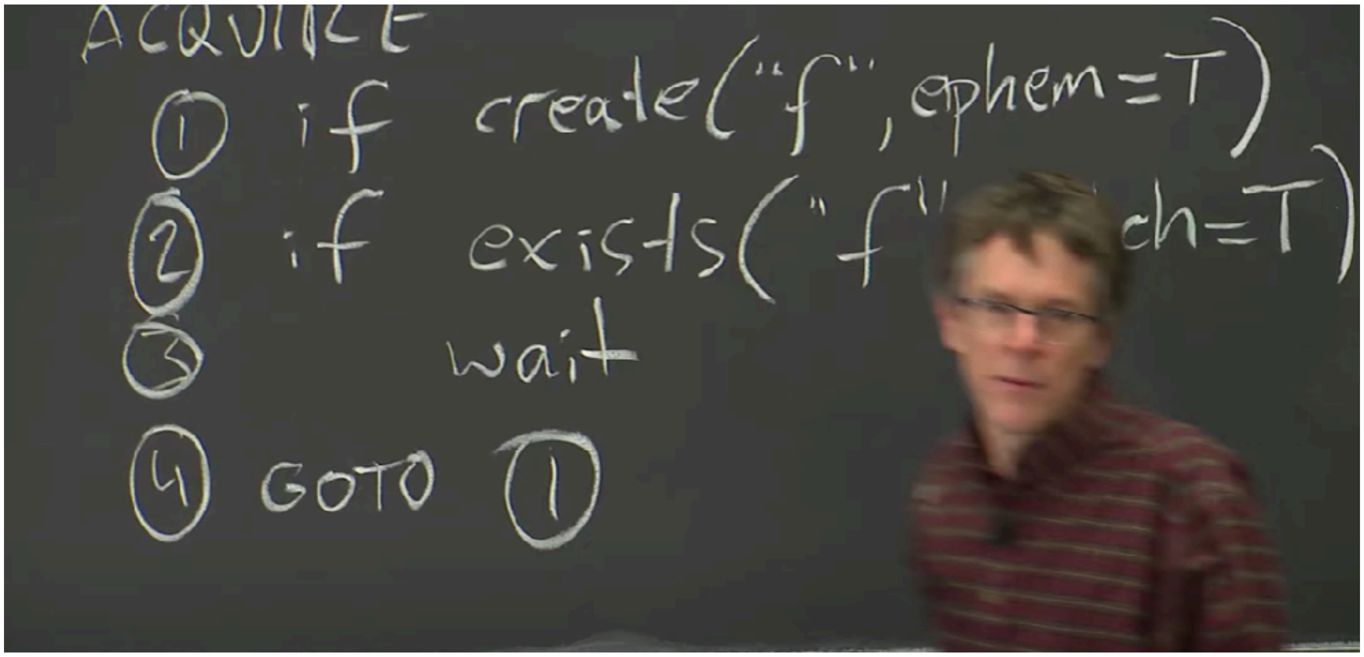
Parameters:

- `path`: The path parameter is accepted but currently ignored in implementation

All methods have both a synchronous and an asynchronous version available through the API. The asynchronous API, however, enables an application to have both multiple outstanding ZooKeeper operations and other tasks executed in parallel. The ZooKeeper client guarantees that the corresponding callbacks for each operation are invoked in order.

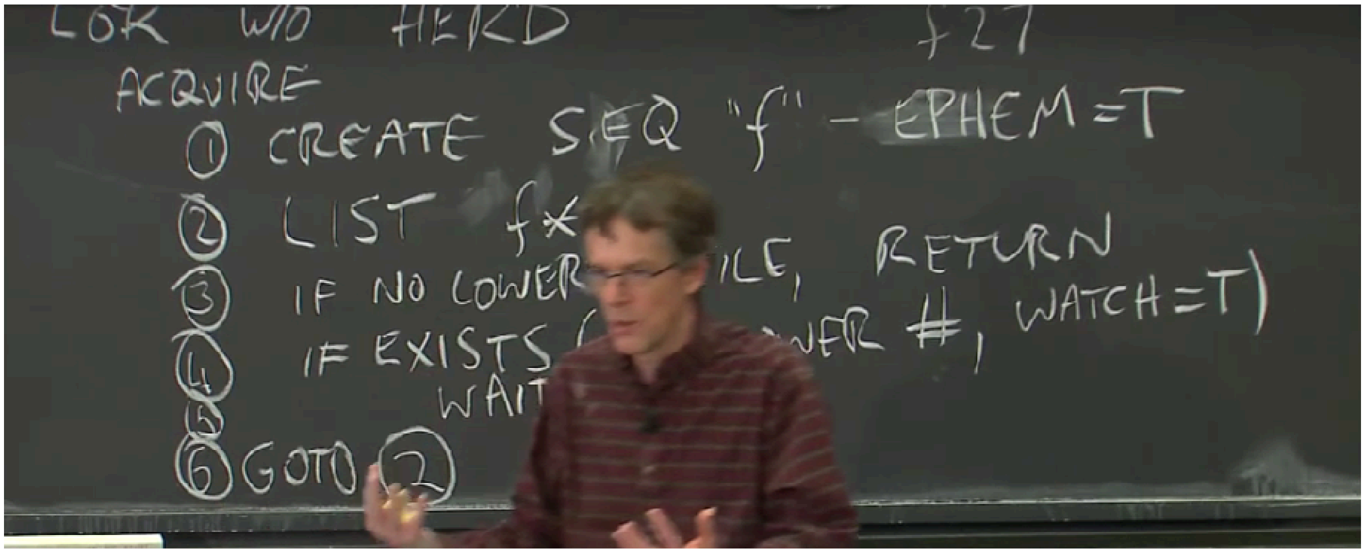


This would be a standard way of setting a value in Zookeeper.
First we retrieve the value and then we set it.
There is a possibility that the value provided by GETDATA could be stale when we get it but in that case our SETDATA would anyways fail and the write would be repeated.
There is also the possibility that GETDATA provides a fresh value however before we call SETDATA the value is changed by some other client and then we again repeat the loop.



```
ACQUIRE
① if create("f", ephem=T)
② if exists("f'[illegible]ch=T)
③     wait
④ GOTO ①
```

ACQUIRE LOCK
We try to create a file and if our create succeeds then we will hold the lock and we return else we check if the file exists with a watch flag and once the file is deleted we try again.
However this algo is quite costly as if a large number of clients is trying to lock a file then it would be quite costly to keep on repeating the loop in case we are still not able to acquire the lock.



```
LOK W/O HERD                    f27
  ACQUIRE
    ① CREATE SEQ "f" - EPHEM=T
    ② LIST f*
    ③ IF NO LOWER[illegible]ILE, RETURN
    ④ IF EXISTS ([illegible]WER #, WATCH=T)
    ⑤       WAIT
    ⑥ GOTO ②
```

ACQUIRE LOCK W/O herd effect
We create a bunch of numbered files in ascending order and we check if there is no lower file then our current file else we wait for the file right below ours in numbering.
In this case each client only waits for one file and the watch event for a client is triggered only once so there is a huge amount of processing power being saved.

However these locks do face a certain problem as in they don't protect us from failures while this lock is being held. So if a client crashes and has modified a bunch of files while holding the

lock, the modified files would show gibberish to us when we try to read it.
So other clients on reading modified files should probably have a mechanism to check if the file data has a corrupted state and run a clean-up mechanism in that case.