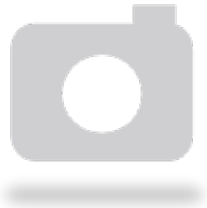


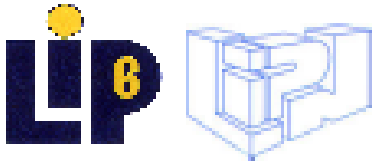
XXX No Theme

NEO DZUG PRESENTATION

Nexedi Enterprise Objects

Presents:

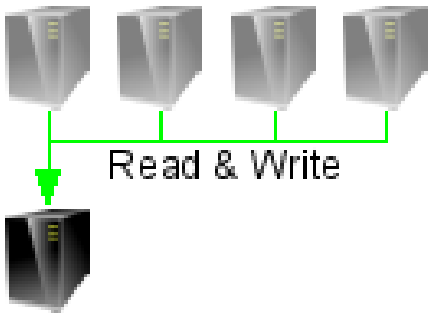
- NEO components
- Load balancing & redundancy
- Locking scheme
- Disaster recovery



Support NEO !
www.neoppod.org

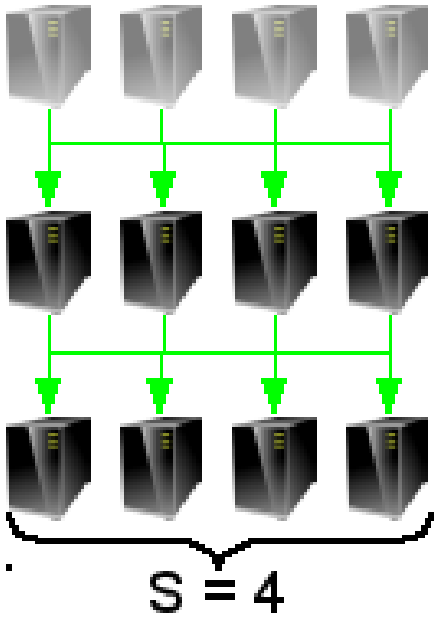
Scalability & fault tolerance

ZEO



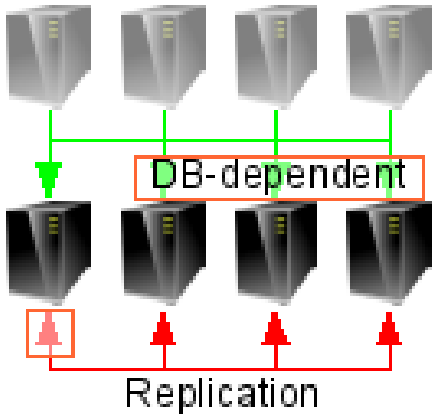
Space	1
Tolerance	0

ZEORaid



Space	1
Tolerance	S-1

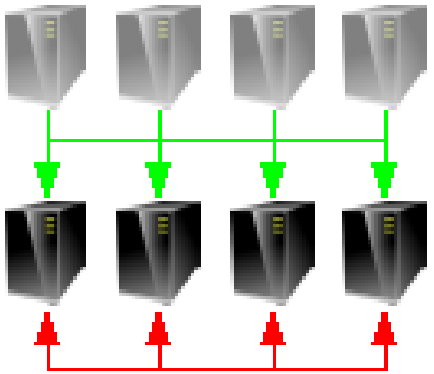
relStorage



Non-native invalidations

Space	1
Tolerance	S-1

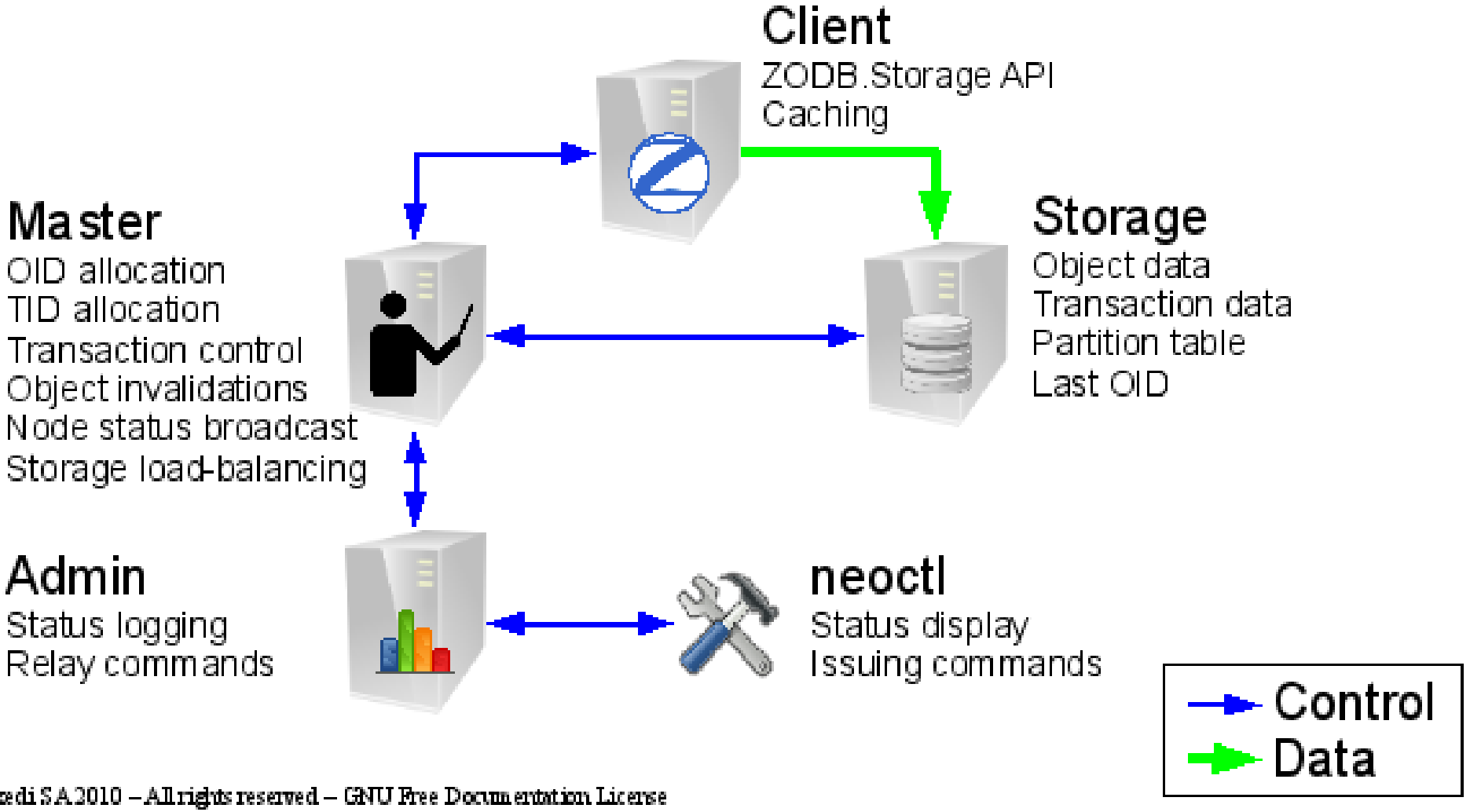
NEO



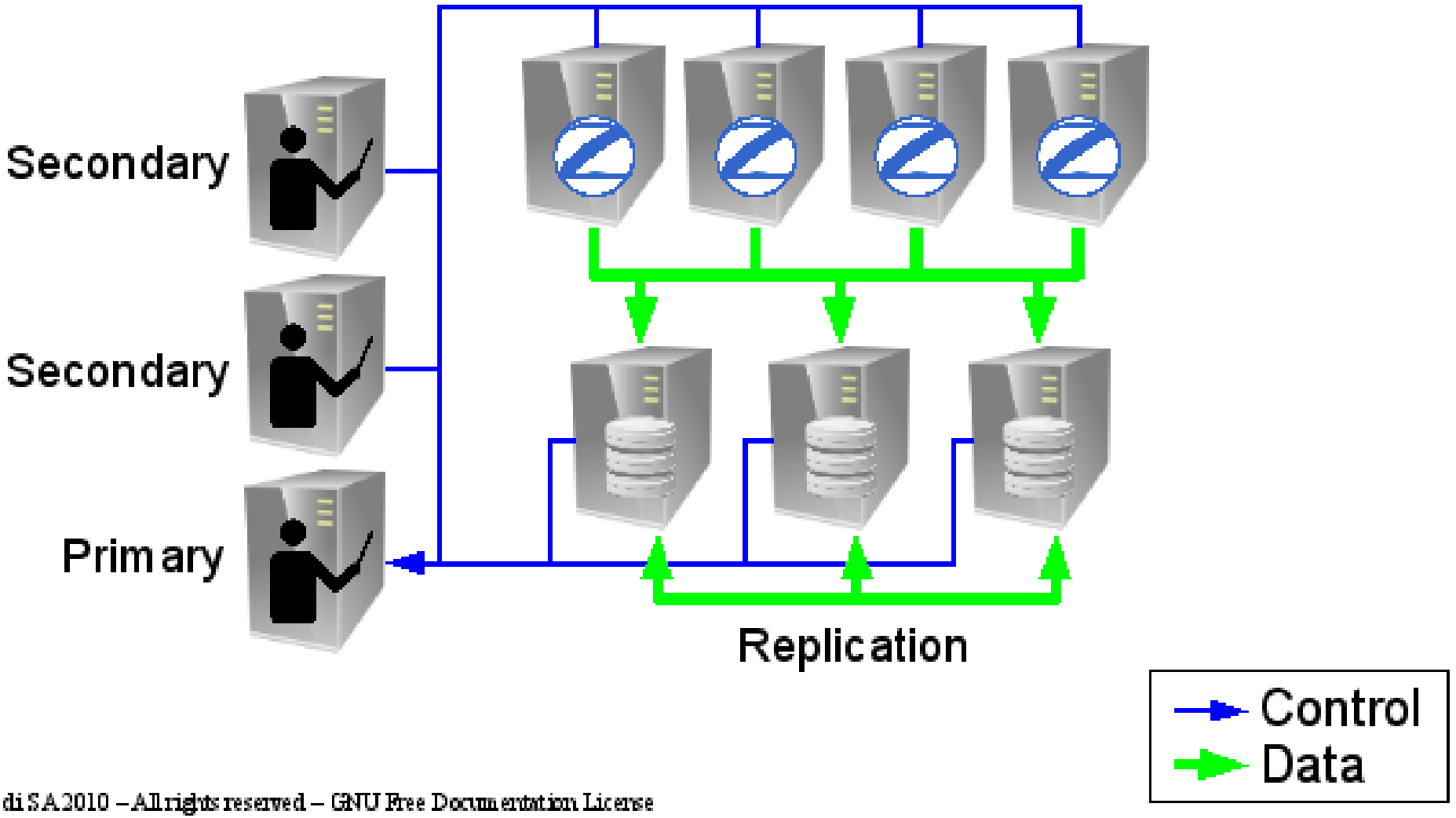
$0 \leq R \leq S-1$

Space	S-R
Tolerance	R

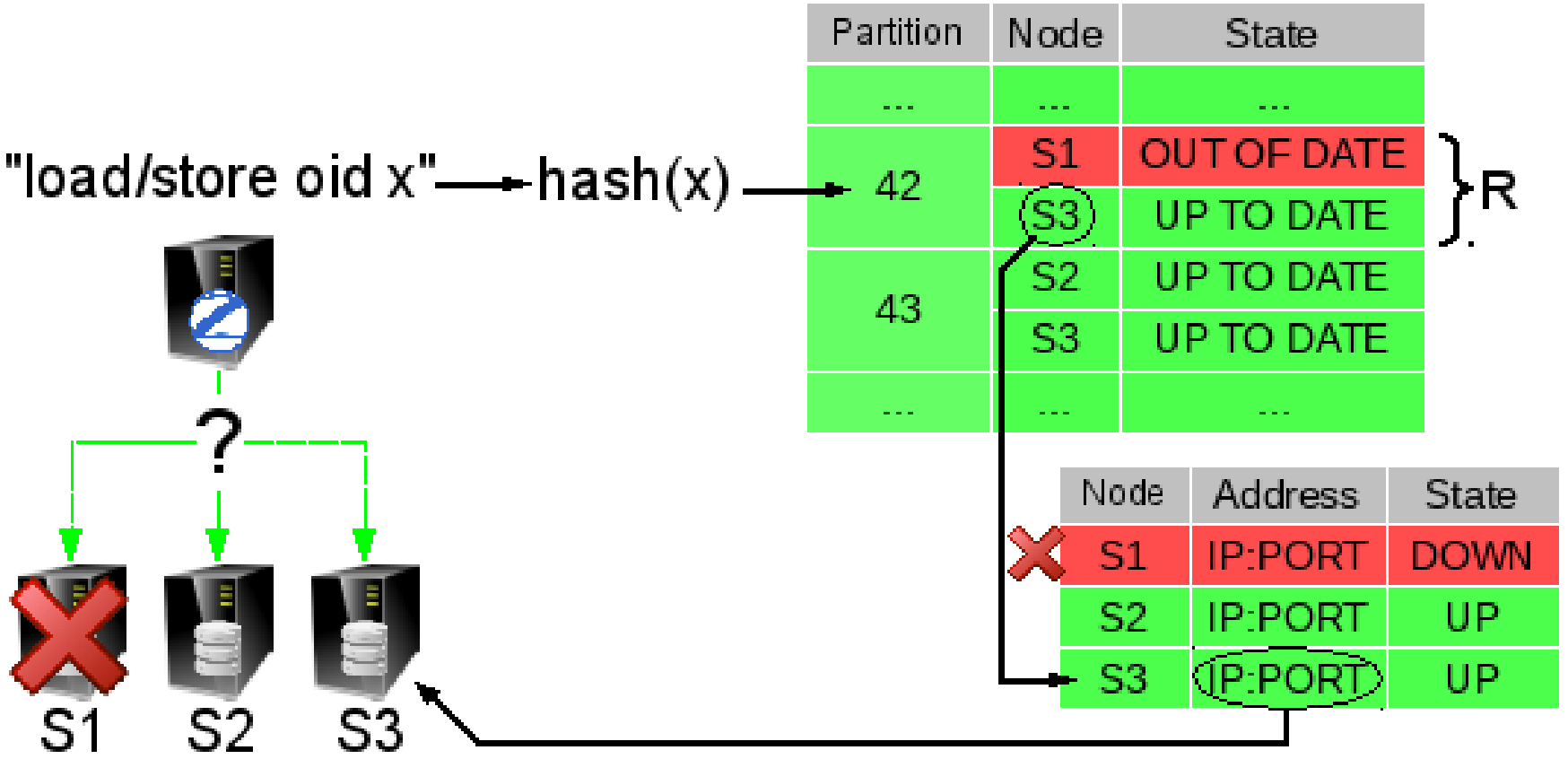
Components & exchanges



Parallelism

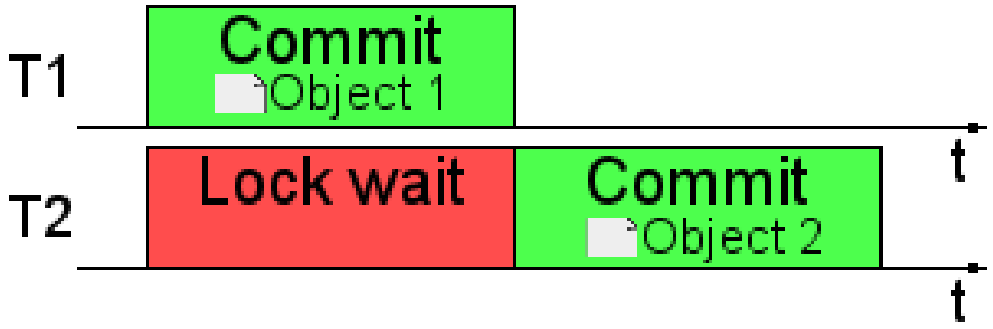


Partition table & node state



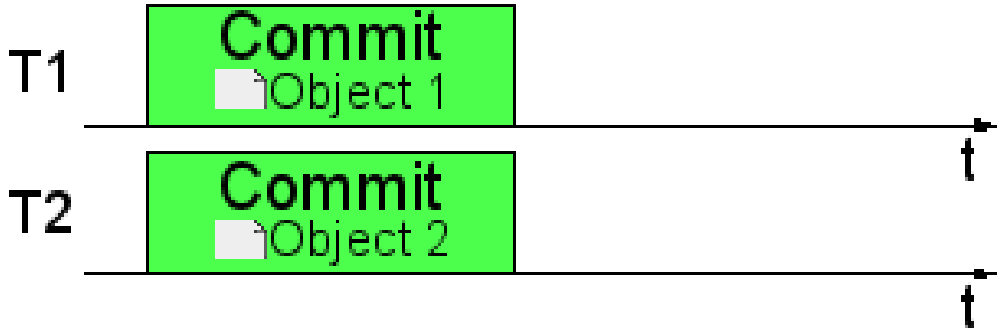
Locking

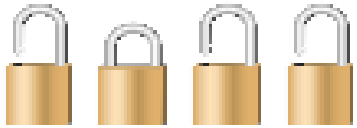
ZEO, relStorage,...



Database-level lock

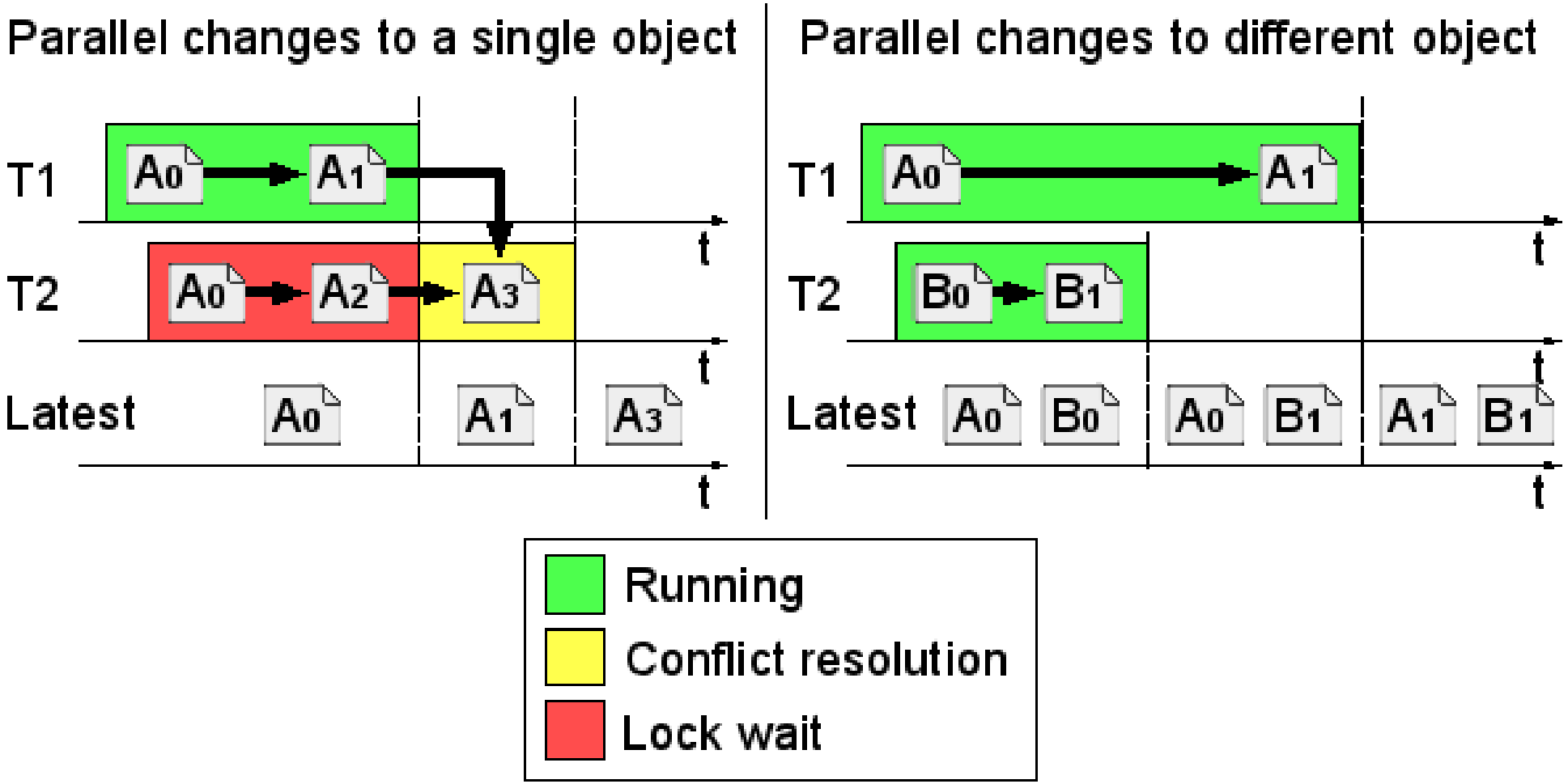
NEO



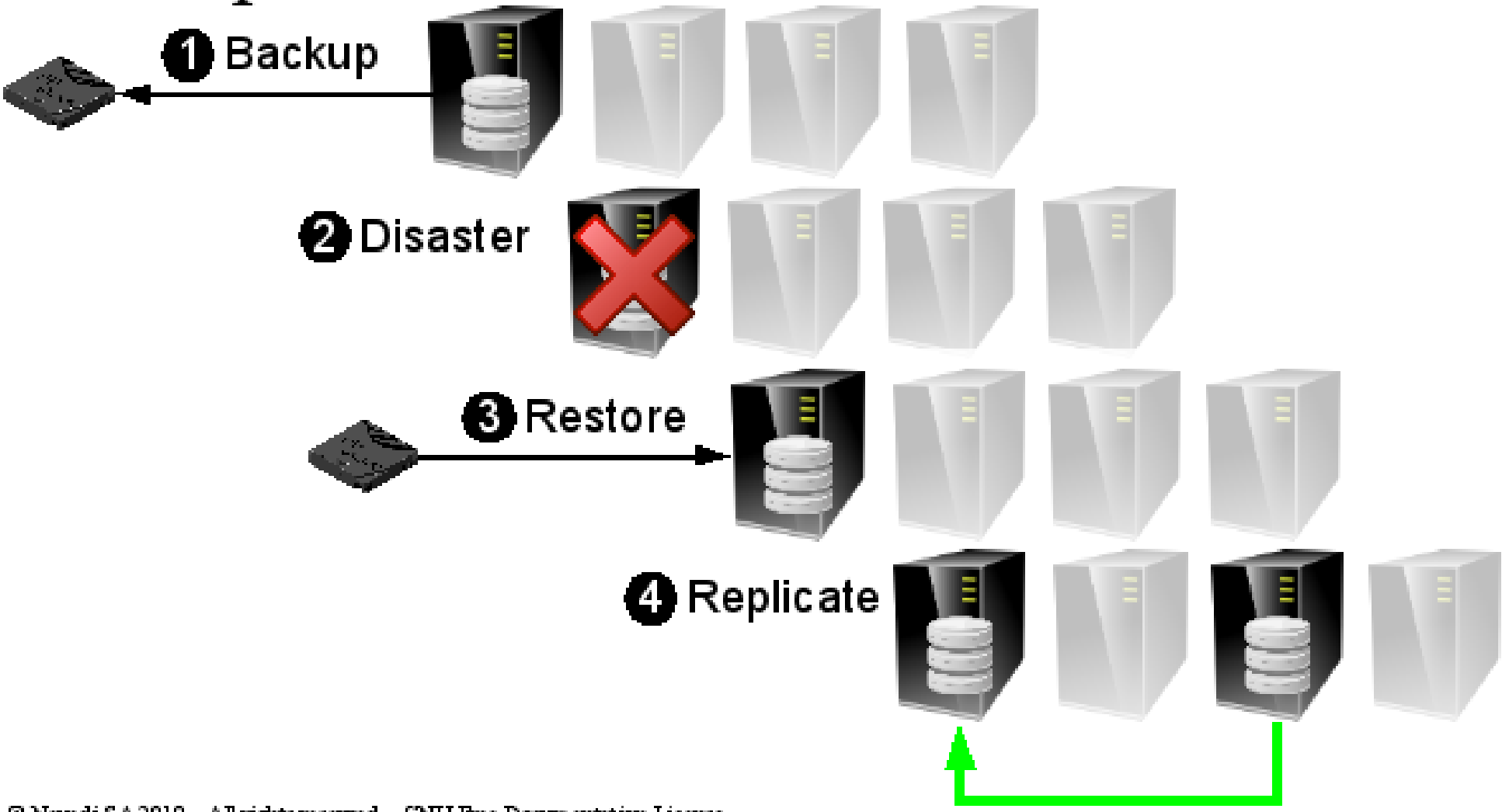


Object-level locks

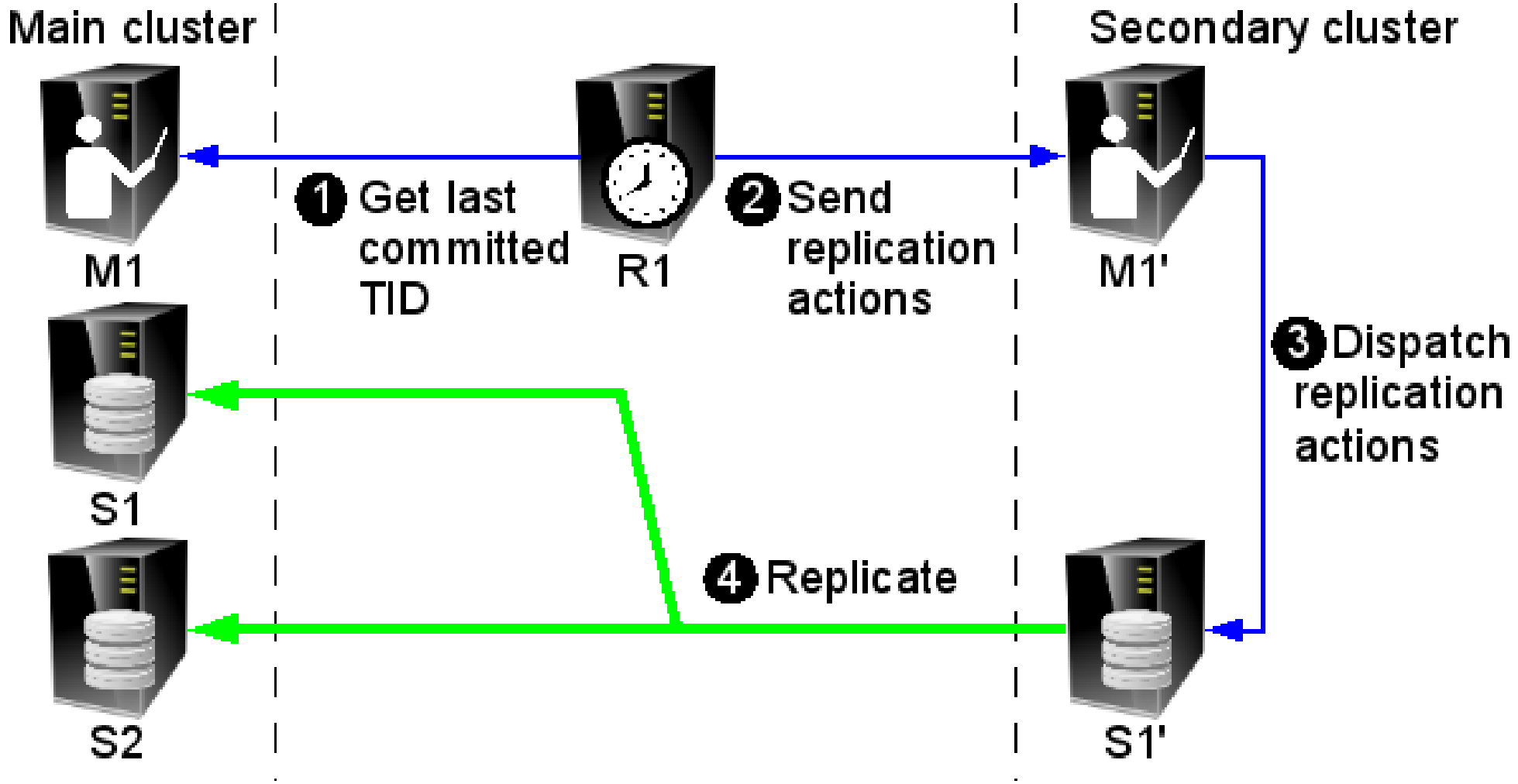
Locking effects



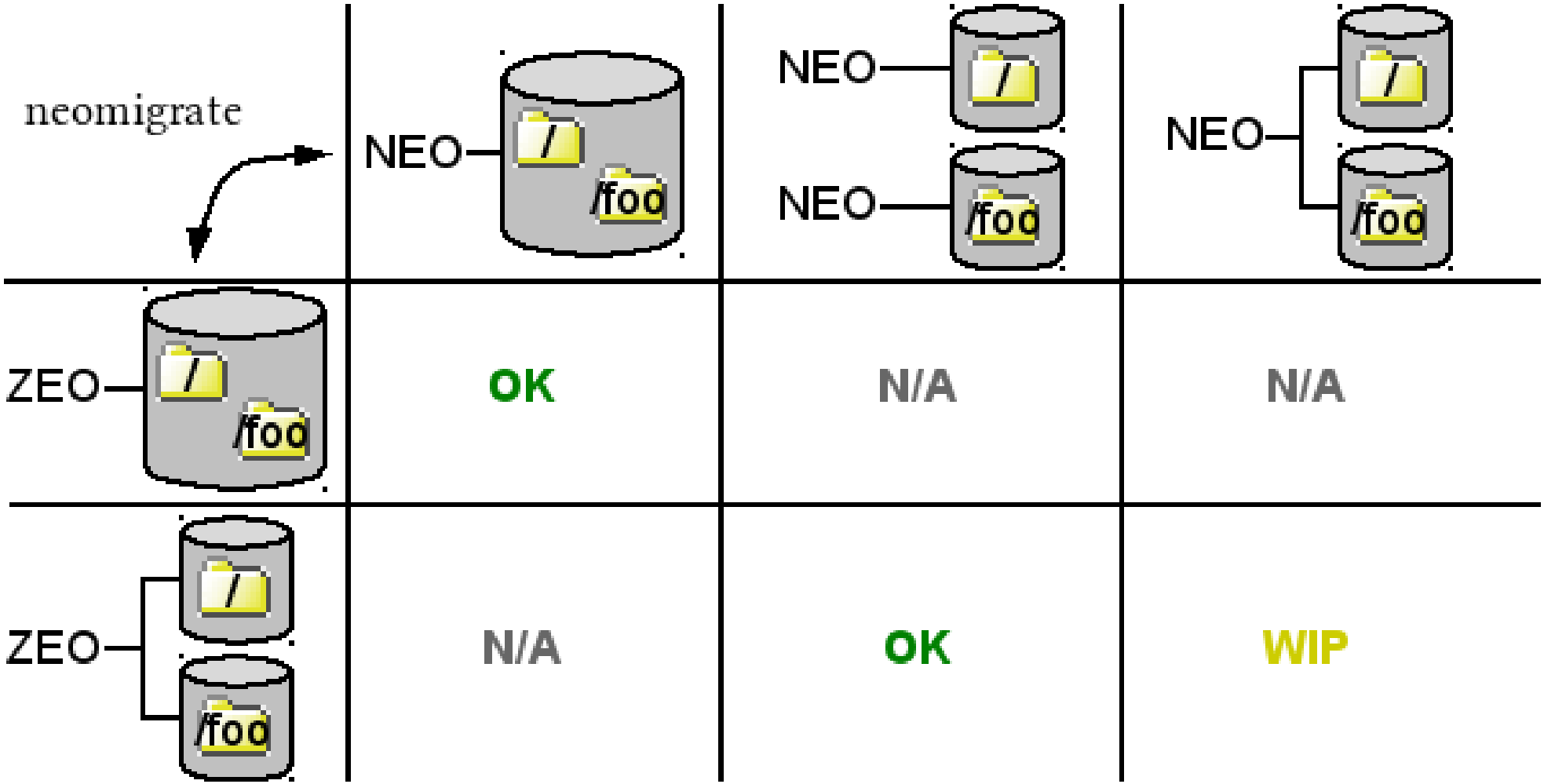
Backup : Local disaster



Replication : Global disaster



Moving to NEO



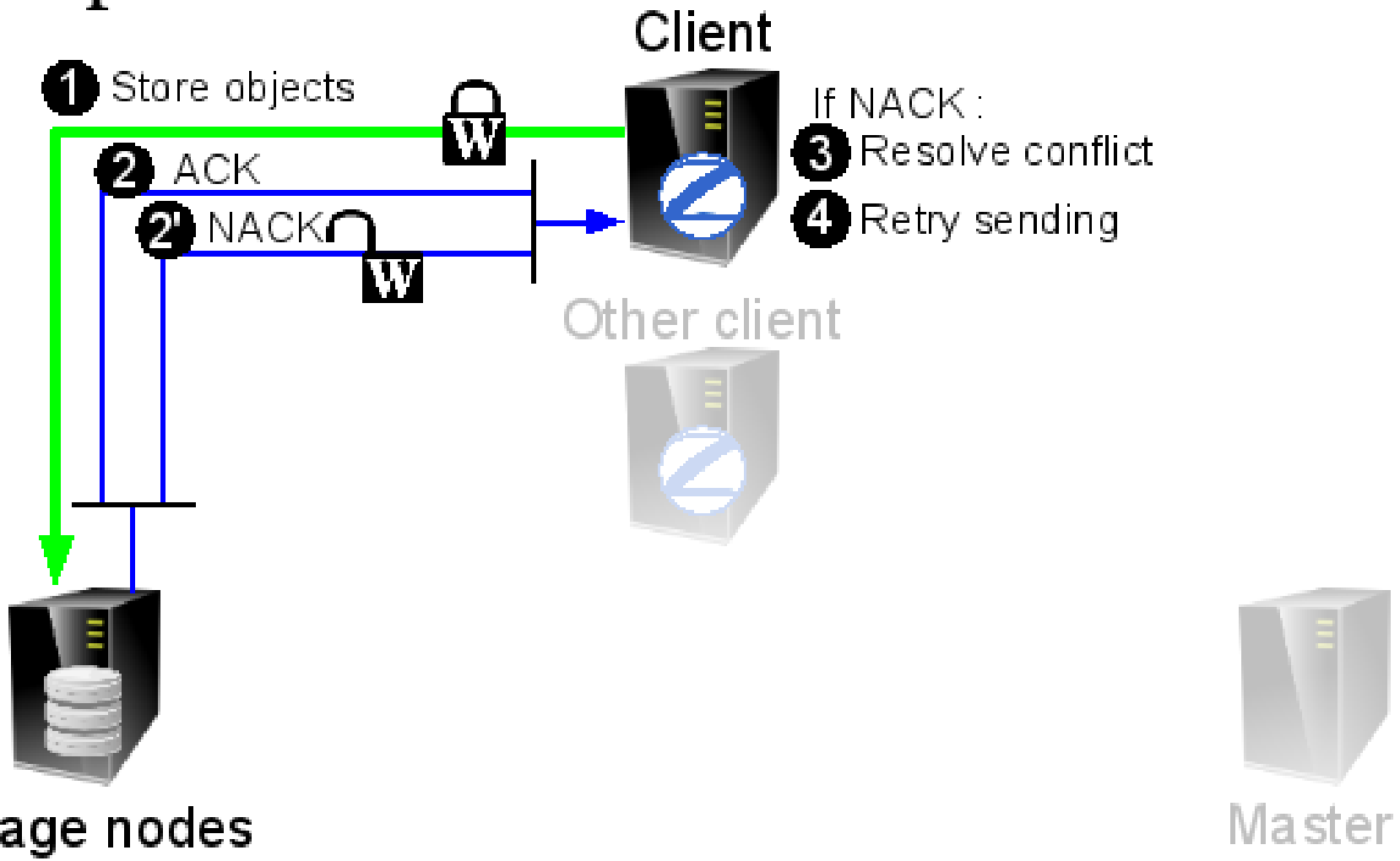
TODO

- Implementation performance (better client-side caching, "zero-copy" protocol module, ...)
- Better admin tools (monitoring, control)
- Backend-independent backups
- Replication node
- Existing ZODB features : BLOBs, readCurrent API, ...
- Contributions to ZODB : transaction garbage collection API
- NEO2 : Long-distance NEO

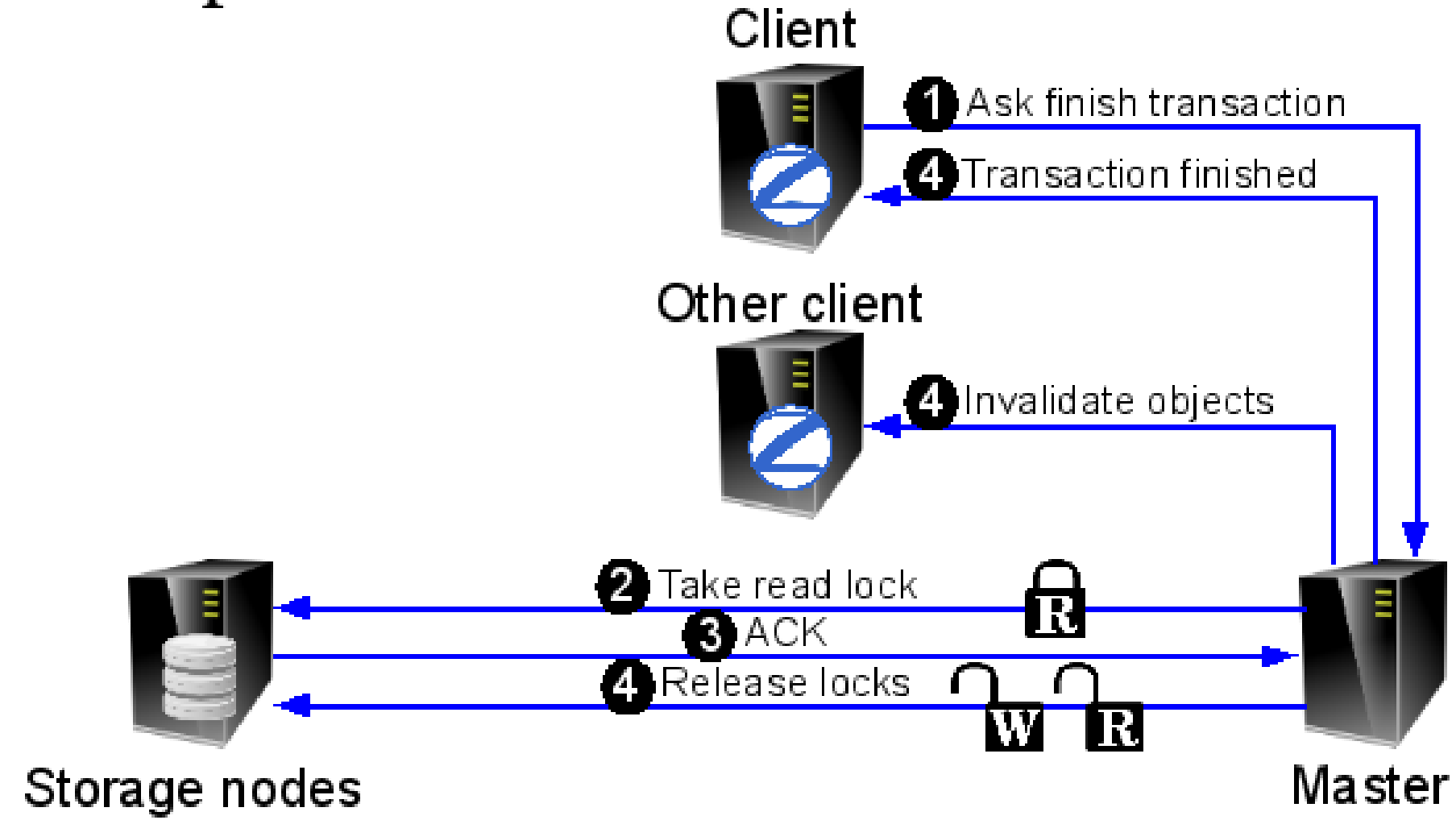
Support NEO !
www.neoppod.org

Extra slides

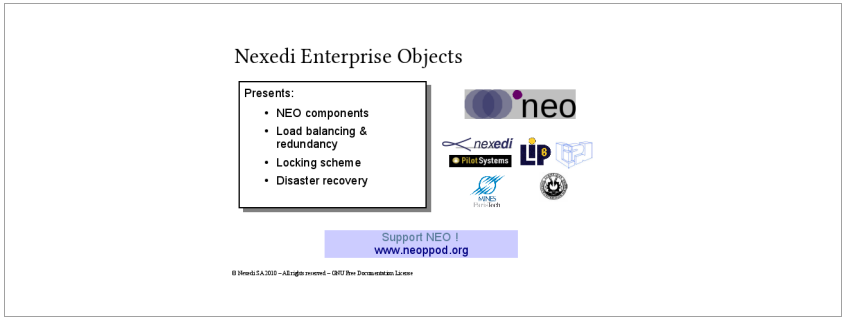
Two-phase commit : Phase 1



Two-phase commit : Phase 2



Notes



▼ Details

NEO (Nexedi Enterprise Objects) is a ZODB.Storage implementation aimed at scalability:

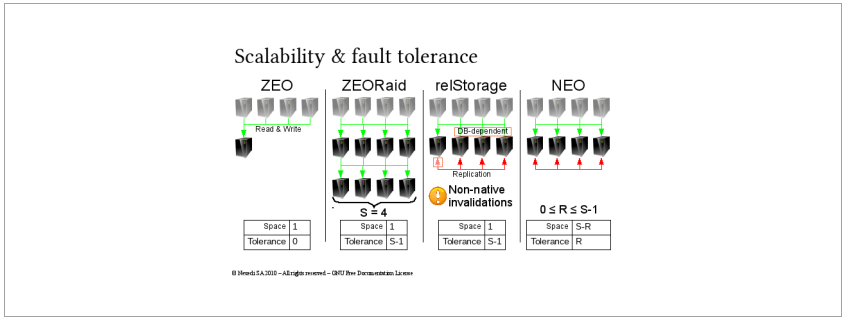
- high data volumes (petabyte scale)

- high availability (fault tolerance)

- high performances (clustering)

It is developped by Nexedi in collaboration with LIP6 and LIPN laboratories whose task is to formally prove NEO design, Pilot Systems, MINES ParisTech and Dakar "Cheikh Anta Diop" University whose task is to stress-test NEO and develop applications taking advantage of NEO design.

This presentation was shown at DZUG 2010 in Dresden by Vincent Pelletier from Nexedi.



▼ Details

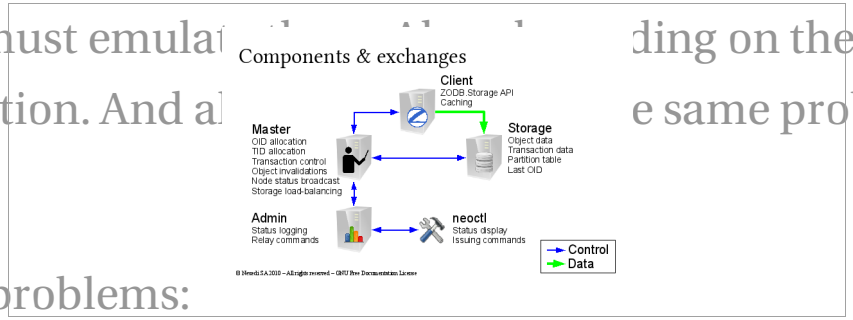
What makes NEO different from other storage implementations, and what needs it addresses.

Why not bare FileStorage ? Bare FileStorage doesn't allow concurrent accesses from multiple processes, so (Python) application cannot take advantage of multiple cores/CPUs, even less multiple machines.

Why not ZEO, which addresses this problem ? ZEO doesn't allow data distribution over multiple disks, so application cannot use more storage space than what is available to a single computer, or get better bandwidth.

Why not ZEORaid ? ZEORaid does not allow scaling in data size terms, as it does RAID-1-ish replication (each underlying storage contains the whole database).

Why not relStorage ? relStorage has scalability issues because Relational DataBases do not send invalidation messages, and it must emulate them. Depending on the underlying RDB, it might not support master-master replication. And also has the same problem as ZEORaid for data size scalability.



NEO's design addresses these problems:

▼ Details

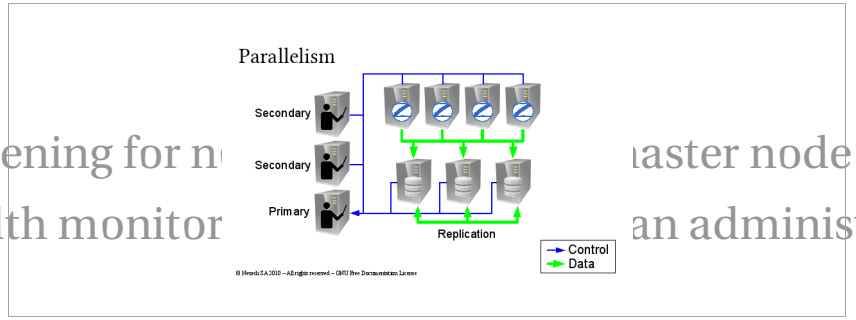
NEO allows tweaking desired fault-tolerance and space efficiency, by providing it with a number of replicates ("R" in above figure), achieving data distribution among machines. Master-master replication is natively implemented in NEO, as well as invalidation handling.

The client part implements ZODB Storage API, allowing NEO to be used as a drop-in replacement in any application using the Zope Object DataBase, including low-level client-side data caching. It uses the Multi-Queue caching algorythm, which improves memory usage efficiency when used as a 2nd-level cache as is the case with ZODB.

A master node is a process providing functions of sequential number generation (OIDs and TIDs), transaction control including object cache invalidation notifications for clients, and storage nodes load-balancing decisions. Due to its function, it is important for good performance for master node to be quickly reachable from any other node, limiting the distance between them.

A storage node is a process providing persistent storage space. This storage is used for database objects

and associated transaction meta-data, and also contains NEO meta-data: current partition table and number generators state.



An admin node is a process listening for n aster node and archiving them to provide administrator with cluster health monitor an administrative command relay.

Neoctl is a comman-line tool to allow querying cluster state and issuing commands, such as adding or dropping a storage from cluster.

▼ Details

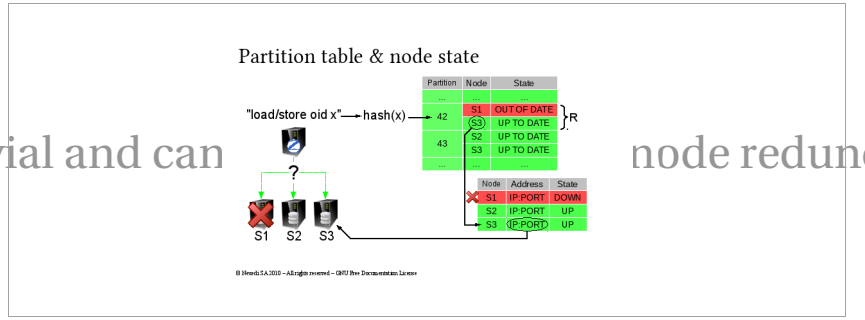
It is important to note that object data never goes through any node but storage and client. NEO objectives are being scalable and resilient to hardware failures. To achieve this, it involves parallelism by scattering multiple instances of each kind of node over different machines:

It is obviously possible to access the same cluster from multiple client nodes (ie, Zope processes), allowing processing power scalability.

Master nodes are also possible to scatter on multiple machines, although at any time only one will be actually use (the "primary master"), while others are just spares ("secondary masters"), ready to take over in case of primary master failure.

Storage nodes parallelism allows two kind of distribution: load balancing, by containing only a part of the entire object database, and redundancy by storing any database part in multiple storage nodes and

making the cluster resilient to hardware failures – as long as at least one copy of every database partition is available.



Admin node redundancy is trivial and can node redundancy.

▼ Details

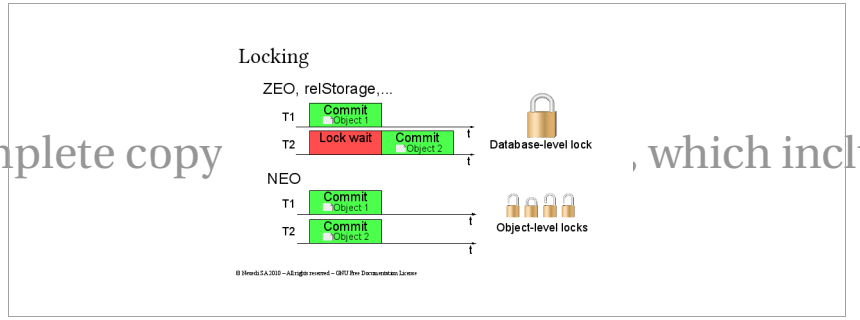
The partition table is known to all nodes in a NEO cluster, and updated by the primary master node. It provides two indirection levels.

The first indirection is between an object and a partition number. This is achieved by a simple hash on persistent object identifier (OID), which is known whether we want to load or store an object. A partition is identified by its number.

The second indirection is between partition and nodes which contain it (we then reffer to cells as a partition on a given). A partition can exist in multiple places in a cluster, which is how fault tolerance is achieved. This level of indirection contains a partition state property, which can have several values:

Out of date: a node in such state might be completely unavailable or in the process of replicating

partition data from other storage nodes. It cannot be used for reading data, but data can be written to it if node is up.



Up to date: this node has a complete copy , which includes all objects and their revisions if any.

Feeding: this node is busy providing data to another storage node. This is a variant of "Up to date" state, but nodes should refrain reading from this node, so replication can operate at maximum speed.

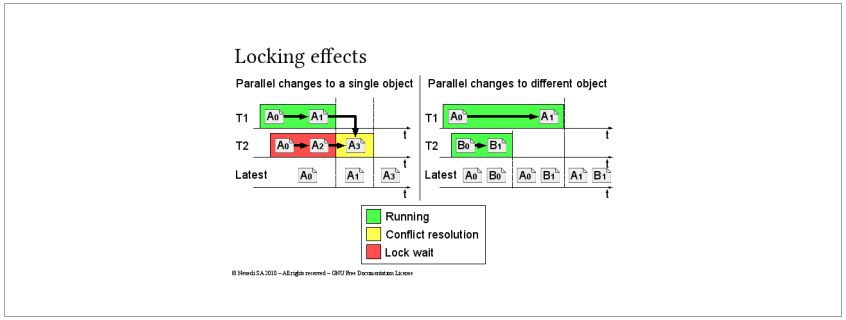
▼ Details

Once the node is chosen, it can be accessed by it's listening ip/port present in the node table.

In usual implementations of the ZODB.Storage API, at least one storage-level lock is used to serialise commits. In given example, T1 delays T2 commit by the duration of its 2-phases commit 2nd phase (implementation-dependent), while each transaction actually modify unrelated objects.

The induced delays grows as the number of applications accessing the database grows, and puts a limit on the benefit applications can expect from multi-processing / clustering.

To remove this limit, NEO uses a different locking scheme: in a way similar to relational database engines using row-level locking rather than table-level, NEO uses object-level locks.



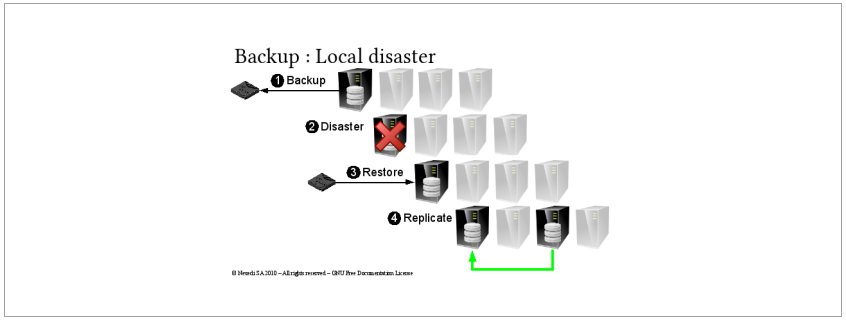
▼ Details

Object-level locking scheme doesn't alter conflict detection and resolution: modifying the same object in two concurrent transactions will result in the latter waiting for the former to commit. When the former is done committing, conflict resolution will happen if object class implements it.

When different objects are being altered in parallel, they do not block each other. This is the desired effect of finer locking granularity, to make commit scale better by being able to run in parallel.

Note: this locking scheme has not been proven correct yet. It is believed that it should not harm existing applicatoins, because the only application-visible effect is that database view does not depend on transaction commit wall-clock start time order. This means that the result from a transaction T2 committing later than a transaction T1 might be visible to a third transaction before T1's result. In

above example, this is shown by having B1 visible while A0 is still visible.



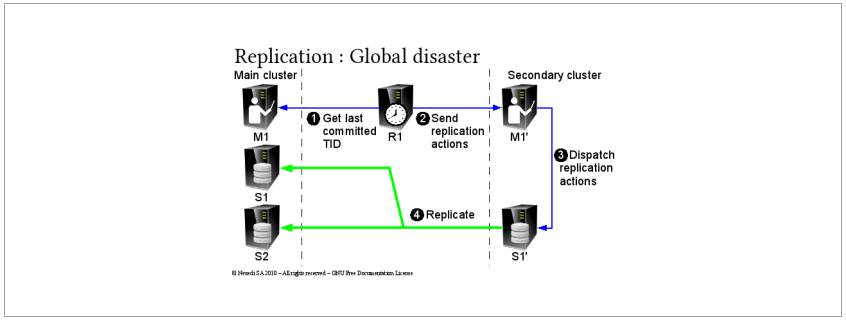
▼ Details

High-availability challenges disaster recovery mechanisms, because restoring a lot of data should not take too much time.

The first disaster recovery mechanism is backup. In NEO, each storage node can handle its own backups independently to then recover from failures which involve few enough nodes so service is preserved (ie, there is still at least one copy of each object and each transaction in the database). Restoring data from node's backup and starting storage process will trigger a replication between surviving node(s) assigned to partition(s) assigned to failed node, and it will catch up with what was missing in its backup. This is just the regular mechanism used when any storage node gets disconnected from a running cluster.

This way, the amount of time needed to restore data does not matter for cluster availability, only the

time needed to replicate missed data is to some extent (it increases cluster load without interrupting normal function).

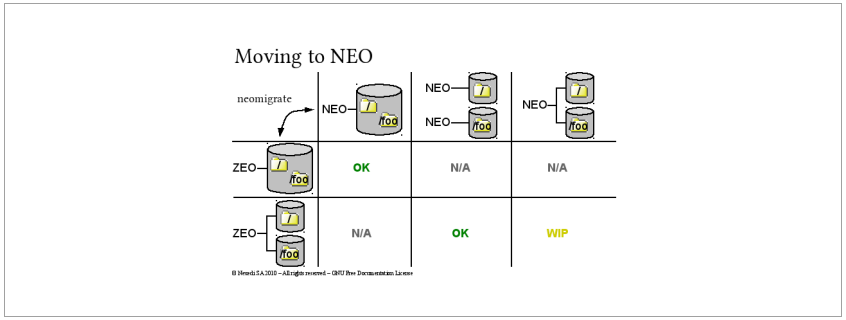


▼ Details

Preserving high-availability when a broader disaster occurs (power loss, fire, ...) is the most difficult part, because it becomes limited by the time needed to restore a node (multiple node could be restored in parallel).

So the solution here is to have a second cluster in a different place (as different as needed to be unlikely to suffer from the same problem as the main datacenter) and updated periodically from main cluster (to be consistent at any point in time).

This is achieved by having a replication node periodically build the list of actions secondary cluster must do to reach a newer consistent database state, and sending them to it. This reduces the impact of increased network latency, which prevents joining both cluster into a single one.



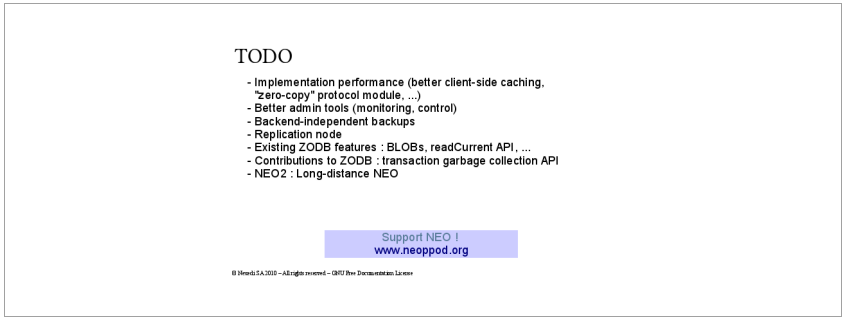
▼ Details

It is already possible to migrate an existing database to NEO, and out of NEO again if needed, via the neomigrate command.

There are currently two possible setups for the NEO cluster side. One is a single-cluster setup, the other is a multi-clutser setup. The former suits single-ZODB applications, while the second would suite applications which already span over multiple ZODBs via, for example Zope mount-points.

The problem with a multi-clutser setup is that it involves a significant overhead: duplicated configurations, duplicated processes. As there is currently no known reliable way of merging two databases into one, the remaining possibility is so-called "multi-export": a single NEO cluster hosting multiple independent ZODB. Work to support this has already been done, but is not finished as of this

writing.



▼ Details

NEO is not mature yet. It is not ready to be used in production, and lacks some functionalities.

First of all, its implementation has not been tweaked for performance. An example of this is the current protocol implementation, which tends to copy data over and over before using it or sending it.

Also, the MultiQueue implementation used in client nodes has been shown to take a huge benefit from being rewritten in a compiled, statically-typed language.

Administration tools are currently quite rough, and doesn't prevent an absent-minded admin from destroying his database.

Backup creation and restoration currently depends on storage back-end own backup implementation.

As NEO should be able to move to different back-ends (KyotoCabinet support is being considered), it should implement this.

Replication node presented in page on Gl y is not implemented yet.

NEO lacks some ZODB features, such as B dCurrent API.

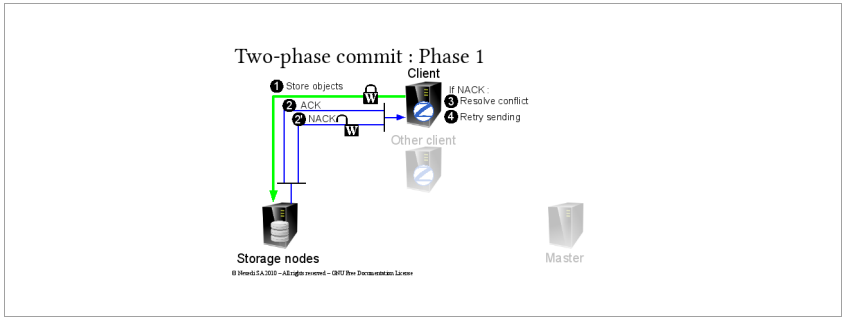
Extra slides

▼ Details

Also, NEO could contribute back code to ZODB, such as transaction garbage collection for a more scalable pack implementation on storage implementations such as NEO, where transaction metadata is not necessarily on the same place as object data.

The following pages were shown as a support for questions asked after the DZUG 2010 presentation.

And finally, a long-term goal would be to develop a long-distance-capable NEO architecture, allowing to spread nodes around the planet while preserving ZODB semantics.



▼ Details

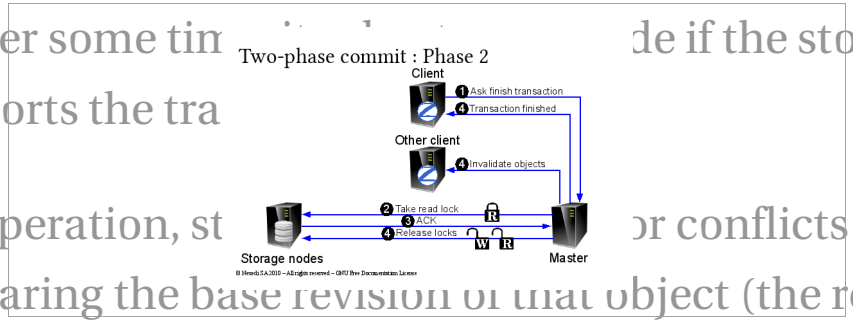
The two-phase commit starts by asking the primary master node to generate a TID, unless one is provided by client application, in which case there is nothing to do at network level.

Then object data can be sent to storage nodes.

Sending an object has the effect of taking a write lock on that object, which is local to each storage node. If that lock is already taken when the node receives the request, the locking TID is checked: if it's later than the one trying to get the lock, a conflict is notified to the client. As object is already being modified by a later transaction, no conflict resolution can happen, so client will always raise a conflict error in such case. Otherwise, it will delay the store operation until the lock is released.

This locking scheme can lead to deadlocks if multiple client nodes send the same object to multiple

storage nodes. These deadlocks are recovered by a time-out mechanism: if a client doesn't get the response for a store request after some tim de if the store request is being delayed due to a write lock, and if so aborts the tra



When the lock is taken for an operation, st or conflicts on stored object. Those conflicts are detected by comparing the base revision of that object (the revision client got from database when it started modifying it) matches its current committed revision. If those revisions are equal, storage sends a acknowledgement to the client node. Otherwise, storage node doesn't take the write lock for that object, and responds to client with a conflict notification.

▼ Details

When client node is asked to finish commit, it sends a single requests to primary master node. That node will then take a read lock for all objects involved in the transaction. When storage nodes take that vote ends this commit phase by just waiting for all pending responses from storage nodes.

lock, they write received objects to a temporary location, where they will survive a node restart but are not accessible by other nodes.

If transaction must be aborted, it is enough to ask all storage to release locks held by that transaction.

Once all storage acknowledged the lock take, primary master node asks them to release all locks on transaction objects. When storage nodes release this lock, they also move previously written objects to the definitive location, where they become available as the current revision of given object to other nodes.

Also, primary master node answers committing client that commit is over, and sends cache

invalidations to other client nodes. In both cases, this is used to invalidate client caches, with the only difference that committing client should store committed data in its local cache rather than just flushing.

This locking scheme achieves a "network barrier": it prevents being able to read simultaneously different versions of the same object while storage nodes are independently committing data.