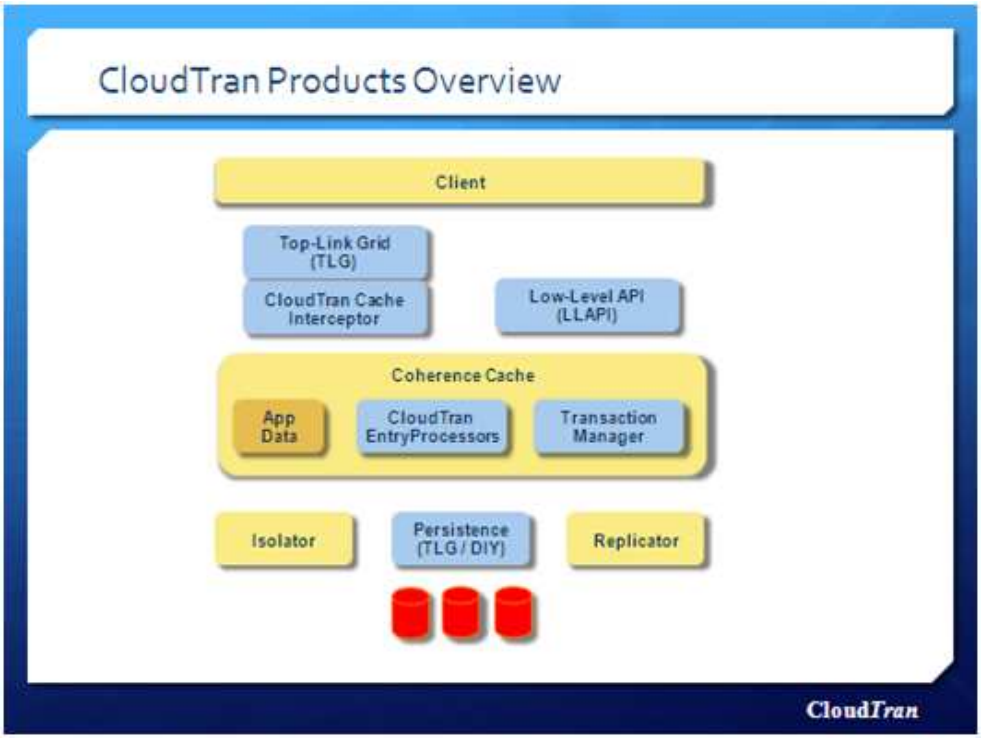
This talk is mostly about Data Center Replication, but along the way we'll have to talk about why you'd want transactionality arnd the Low-Level API.

Topics

- Overview of CloudTran, Transaction Manager and Replicator
- CloudTran TopLink-Grid
- Low-Level API
- Replicator - What, How, Issues
- Summary



Roughly speaking, the yellow boxes here represenet nodes; the blue boxes are code libraries.

The client can use two interfaces to create CloudTran transactions:

- TopLink Grid, which uses a specialised CloudTran interceptor to add cache-based transactionality
- the Low-Level API, which operates at the cache level.

These APIs have pieces that run on both Client and Cache nodes.

Most of the remaining functions are based in the Coherence cache - they sit alongside the App Data.

- there are CloudTran EntryProcessors that implement the transaction manager functionality
- there are also Transaction Manager functions outside of entry processors such as persistence processes and repartitioning handlers.

The Isolator and Replicator modules typically run on special-purpose nodes; they are service-based, typically with 2 or 3 nodes running the service.

The Isolator preserves order of transactions going to a data store: it prevents GCs on a transaction manager node allowing later transactions to overtake the existing one.

The Replicator does transactional data center replication

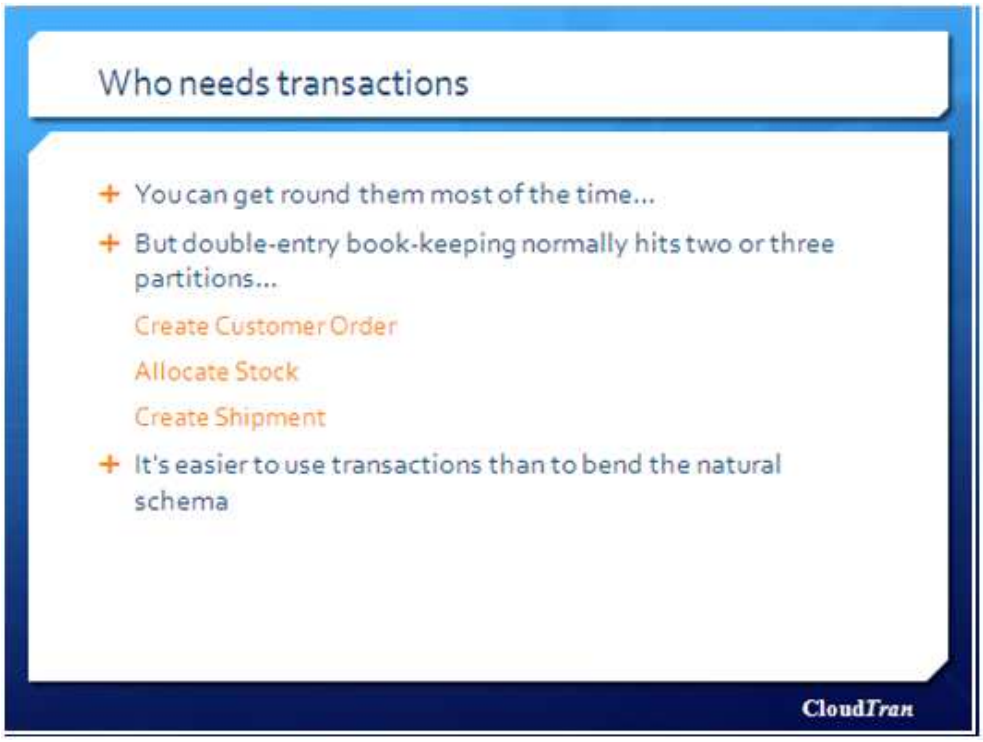
The client API supports special transactional operations for put, remove, invoke and map triggers. Other operations are delegated to the underlying Coherence implementation.

The Transaction Manager module is scalable - we have run as many as 100 transaction managers in a deployment. It also handles failover/repartitioning whilst committing a transaction and guarantees to commit correctly.

The Isolator is not scalable: it is a service with a primary node handling all traffic.

- However, there is no further IO from this node: in the event of a repartitioning, all state is recreated by querying the manager nodes.
- The isolator handles about 200,000 rows per second per core.

Other bullets are discussed in more detail later.



Many cache applications can avoid transactions

- by aggregating related objects together
- using partition level transactions to update multiple items in a single partition.

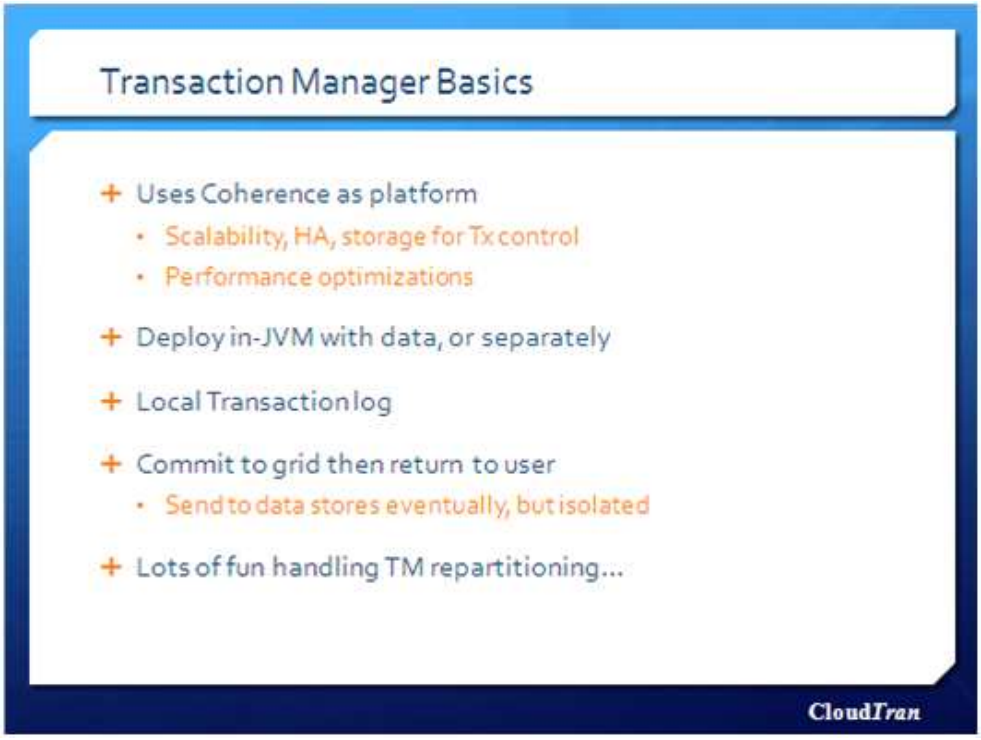
But there are cases where the you cannot easily avoid this: the classical double-entry book-keeping transaction, where the objects involved typically do not scale in the same way.

In these cases, it is much quicker and easier to use transactions than avoid them, by bending the schema or implementing a form of transactionality at the application layer.

- initial development is faster
- less risk of service outages from data corruption or edge cases
- reduced complexity means quicker and easier maintenance.

Without CloudTran, you have uncomfortable alternatives in transactionality:

- good persistence speed using write-behind, but lose transactionality (in the grid and the data store)
- use a distributed transaction facility, but lose either cache-based persistency (with Transaction Framework) or performance (with an external transaction manager).



By using Coherence as its platform, CloudTran can

- store transaction information in the cache, which is faster than using disk or going to a separate tier
    - this automatically gives HA and scalability.
- use various optimizations, such as
    - storing a transaction's metadata (tx control info) on the same node as (some of) its data
    - sending transasction control and application data in the same message

Normally it is easiest to deploy data caches and transaction managers on the same node.

For huge grids, it may be easier to manage if the transaction managers are on separate nodes.

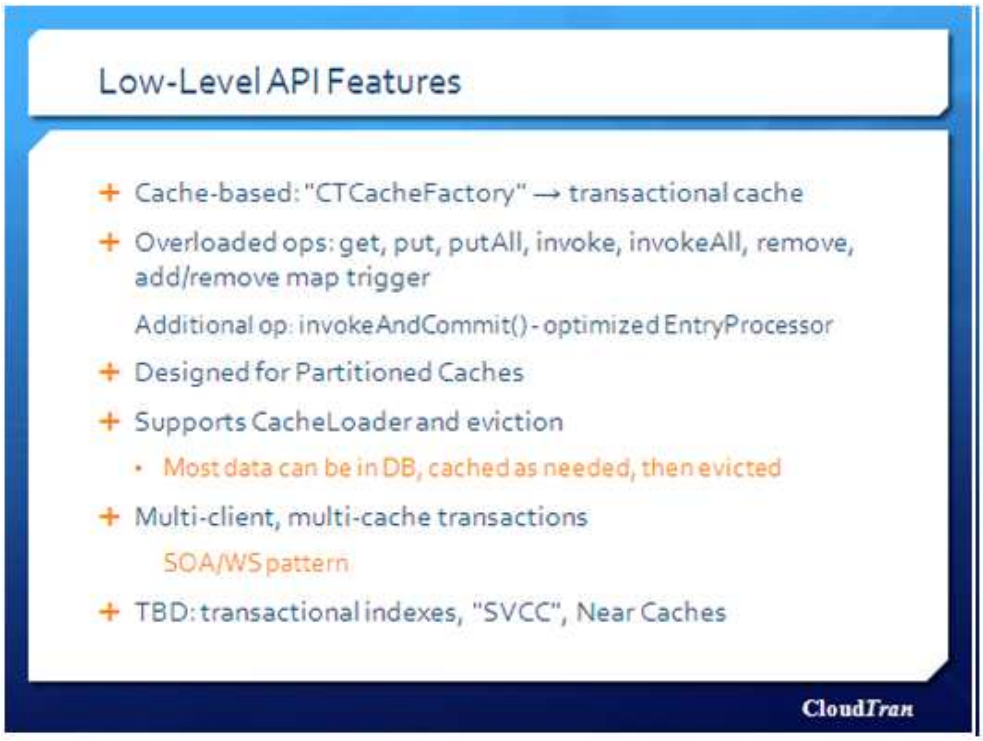
On Commit, before returning to the using

- data is committed to the grid, and to the (local disk) transaction log if necessary
- eventual persistence to data stores - it is queued and normally proceeds in parallel, but does not hold up the app.

After repartitioning, a different node may get "ownership" of the transaction - its control information moves to a different node.

The new manager node takes responsibility for completing the transaction commit using state information in the cache. The manager handles edge cases, like

- a retry of the client commit overlapping with the manager-initiated commit
- the original manager is still running persistence threads - so must be allowed to complete the persistence.



The Low-Level API operates at the Cache level.

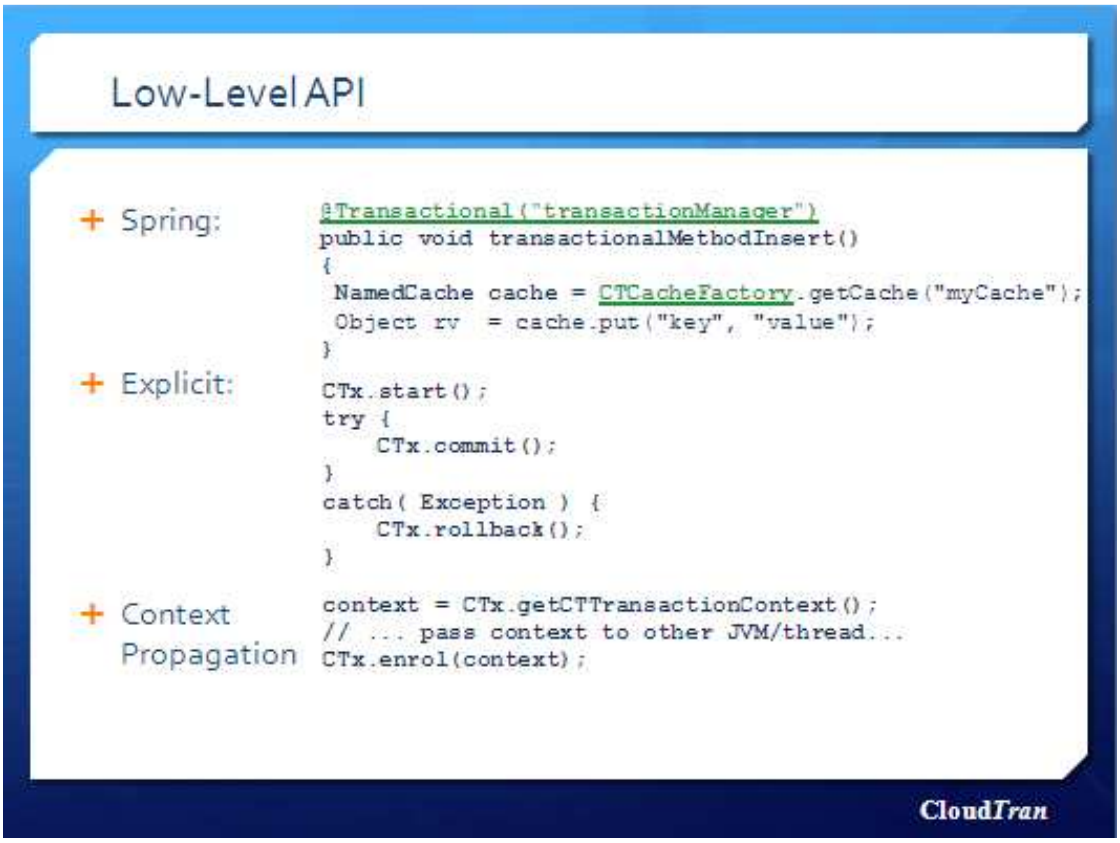
If you get a cache from CloudTran, it will add transactional semantics to the data operations - put, remove, invoke etc.

- It does this by adding decorations to the cache entries involved. This means that other applications can still use the committed values while the updates in a transactional are in process.
- Note that the "get" is in the list of overloaded operations: a transactional read acts like a pessimistic lock, and will fail if another use has a transactional read outstanding.

The combination of CacheLoader/eviction - so you can bring data into the grid and then release it when no longer needed - means that CloudTran provides an effective cache to databases with much larger capacities than the Coherence grid.

- It is "effective" because transactionality is preserved and transactions run close to normal grid speed (rather than other distributed transaction operations).

"TBD": Future features are likely to include support for these items. "SVCC" stands for "Single Version Concurrency Control" and gives a guaranteed consistent point-in-time distributed read across the caches.



Spring

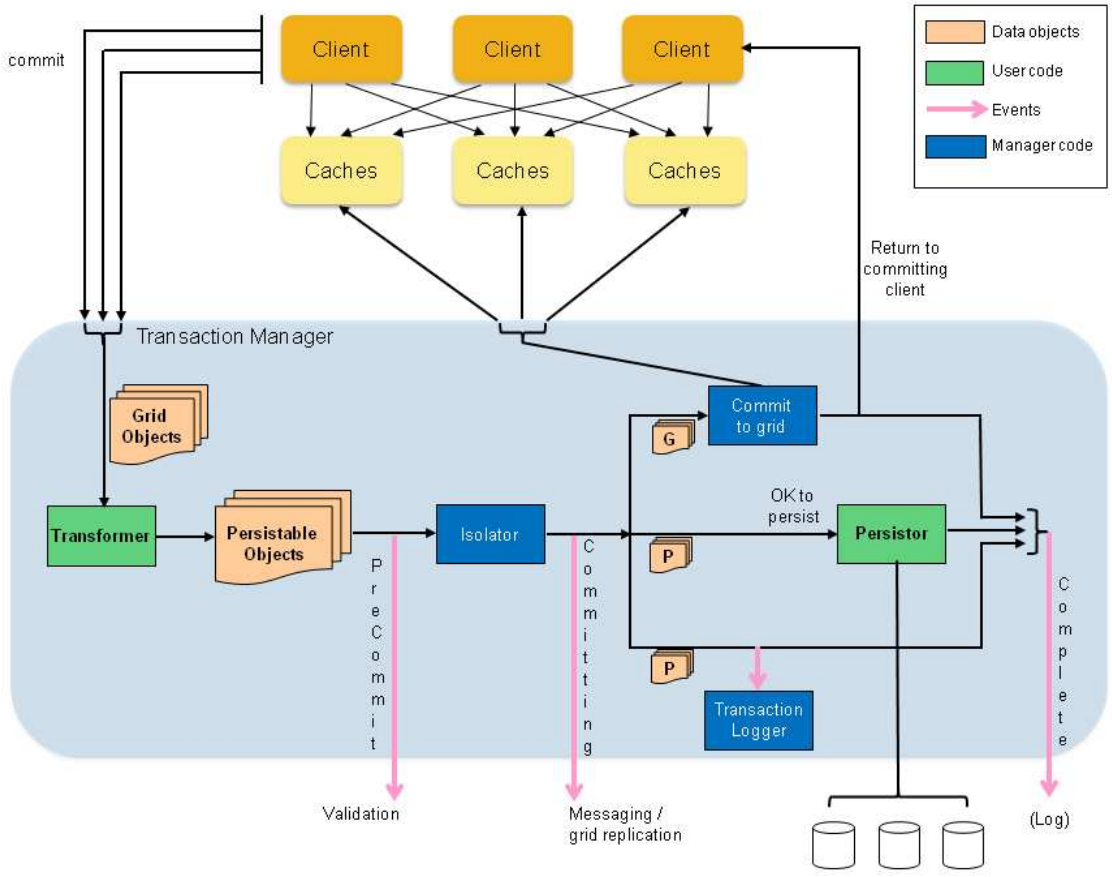
- uses the @Transactional annotation at class or method level. Add transaction manager in with spring config
    - transaction is started implicitly on entering the method and committed/aborted on exiting the method
- get a transactional cache using CTCacheFactory (rather than plain CacheFactory). Any number of caches can be involved in a transaction

Explicit

- Without Spring, you need to explicitly execute start, commit and rollback.
    - like in Spring and J2EE, the transaction is attached to the thread.

Context Propagation - extending transaction to other threads or clients

- to get the transaction context (which holds the transaction number, timeout etc.), call getCTTransactionContext().
- this can be passed as an object to other threads or processes using Coherence processors
- they can enrol in the transaction with the enrol() method
- the last client that executes commit() (which is normally the original starter) triggers the real commit at the manager.
    - When that happens, all the pieces of the transaction from the different clients are committed at the different Coherence nodes.

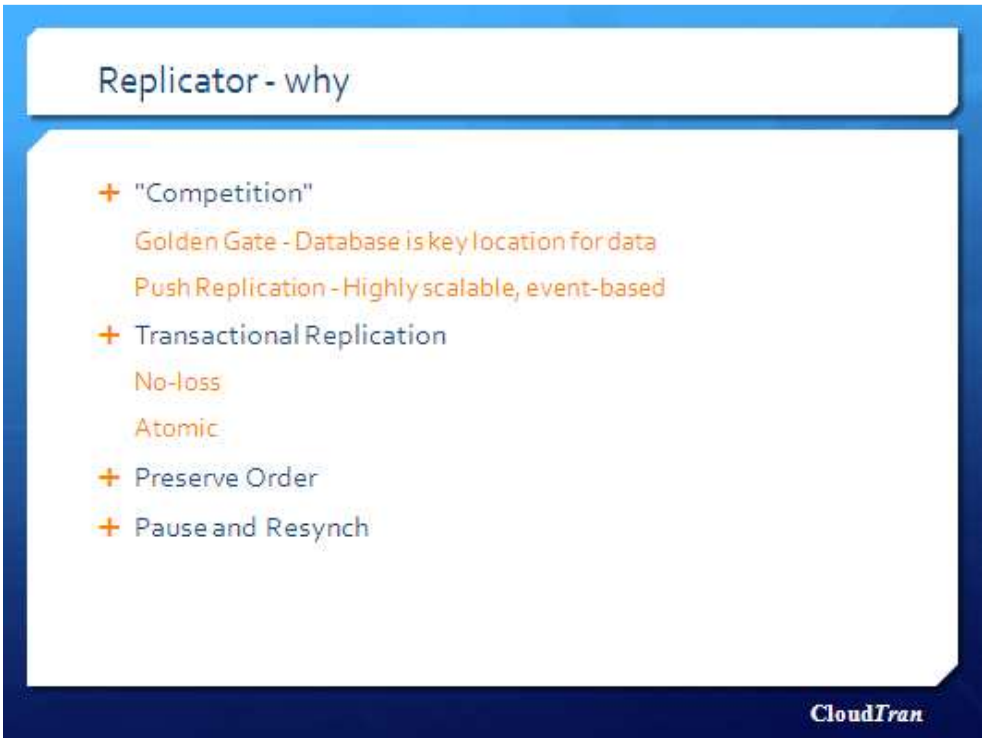


In the LLAPI - unlike TopLink Grid - there is no ORM. So we have to do our own mapping.

There is a plug-in - in the two green boxes - to (a) transform grid objects to persistable object and (b) send them to the persistor.

User code can also be invoked by events at various points - the pink arrows.

- The replicator uses the "Committing" event as standard, which is fired when the transaction is guaranteed to commit.
- The "hospital" event is not shown - this is normally used when a conflict must be resolved.



Normally GoldenGate or Push Replication are used to replicate grids.

We offer the replicator service because it is

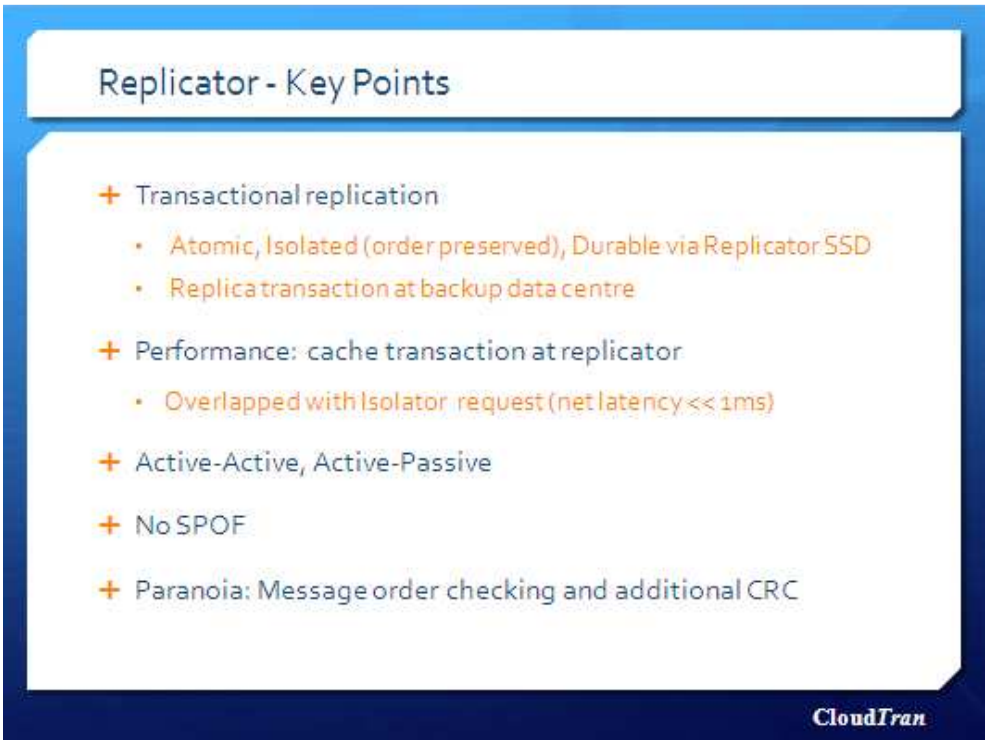
- grid-to-grid, preserving order
- transactional
- lossless.

Most companies interested in transactionality in one grid are equally interested in a replication alternatives, and so extending transactionality is an attractive feature.

By pausing replication from one data center and directing traffic to another data centre, the first data center can be paused and restarted.

> When the grid is restarted, the transactions made in the meantime are automtically replicated to the first data center, and it can then be restarted.

Restarting a grid in this way is easier to manage than the rolling restart.



Transactional Replication:

- the remote replicator updates the remote grid by duplicating the transaction at the local data center

Performance == Low latency

- the scheme depends for its low latency on caching the transaction at the replicator (which is a different node than the transaction manager)
    - by using SSDs, synchronous write times come down to order of 10 μs in current drives
    - if you don't do cache the transaction, to get committed replication at a remote data center takes between 50 and 500ms, simply because of the time to send messages between the two data centers.

      This is unacceptable in most grid apps.
- the hop to the replicator is overlapped with the call to the isolator so the additional latency on a transaction will normally be very small (i.e. order of microseconds) if SSDs are used

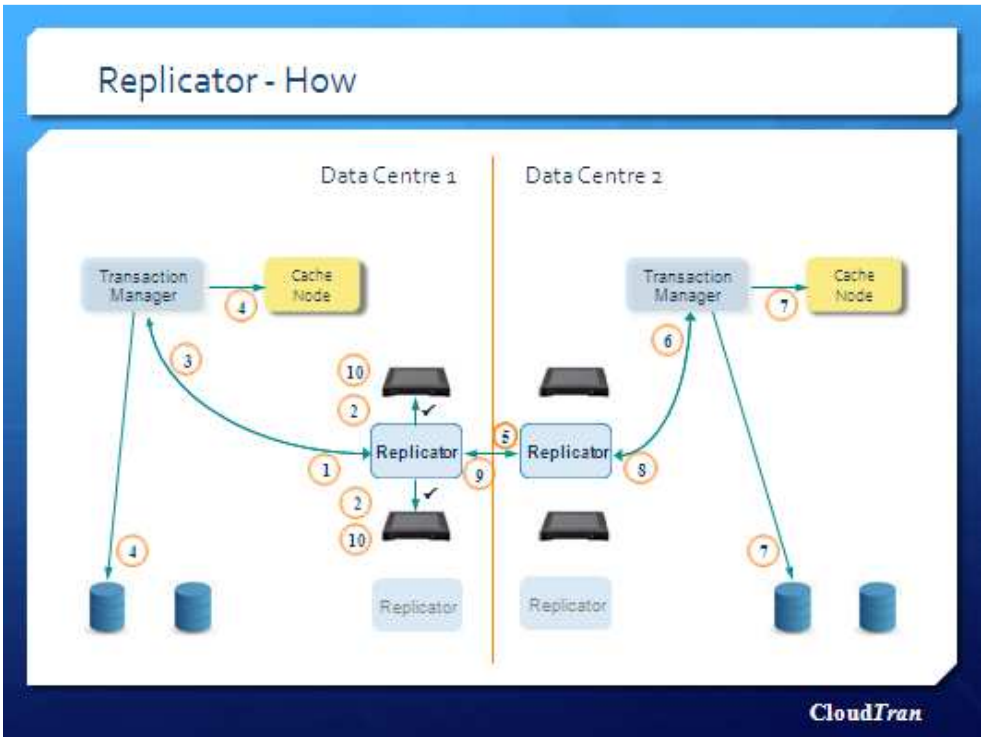
The replicator operates in both directions, so supports **active-active** and **active-passive**.

SPOF

- there is no single point of failure in the recommended deployment, because the replicators operate as a service, and elect a new primary if the current primary goes down

Paranoia:

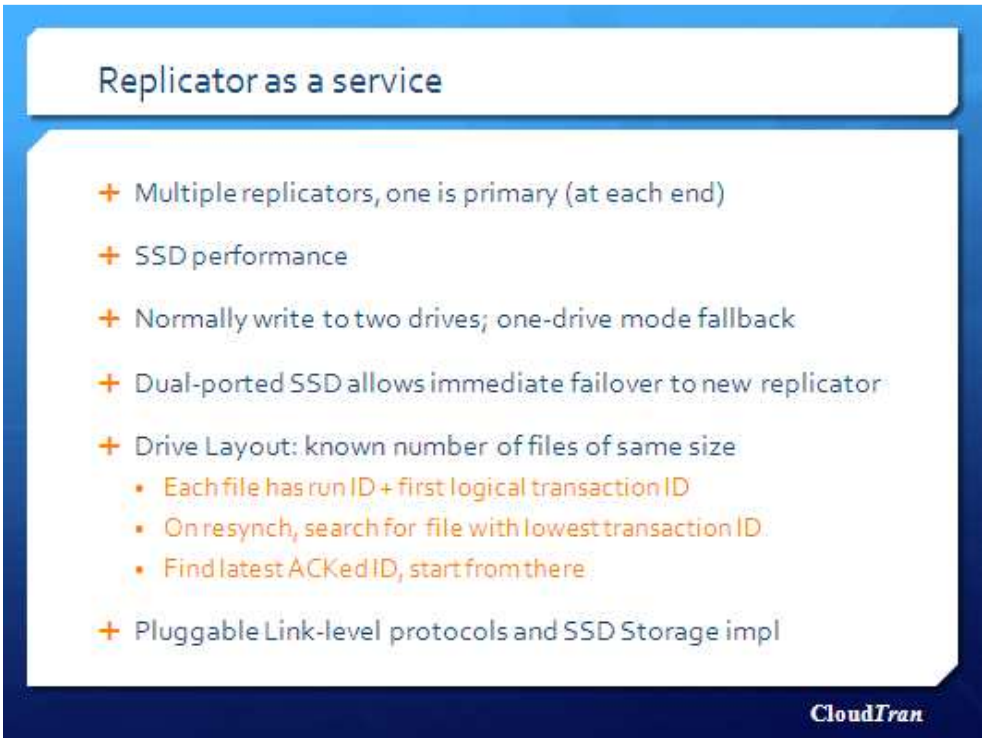
- - even though TCP/IP orders messages and has 48 bits of CRC, we add further order and CRC checks (either Adler or CRC-32).



If you run the slide show, you see the separate steps in this animated slide.

1. Once a transaction is marked as committing, the transaction data is sent to the replicator. This is in parallel with the request to the isolator (not shown).
2. The replicator writes to two SSDs, using an append log.

   Once the first write operation is complete, the request from the transaction manager returns.
3. The request returns to the manager.
4. The local transaction now proceeds to commit.
5. The local replicator sends the transaction to the remote replicator.
6. The remote replicator acts like a transactional client and commits the transaction.
7. Normal transaction writes to cache and disk at the remote DC.
8. The remote transaction call returns to the replicator.
9. The remote replicator sends a TxACK message to the local replicator.
10. The local replicator writes a "completed" record to the append log.



1. We assume SSDs. They cost about £200-400 retail, £2-4000 enterprise. Random write latency is as low as 12-15 μs these days.

   It doesn't make economic sense to use hard disk.

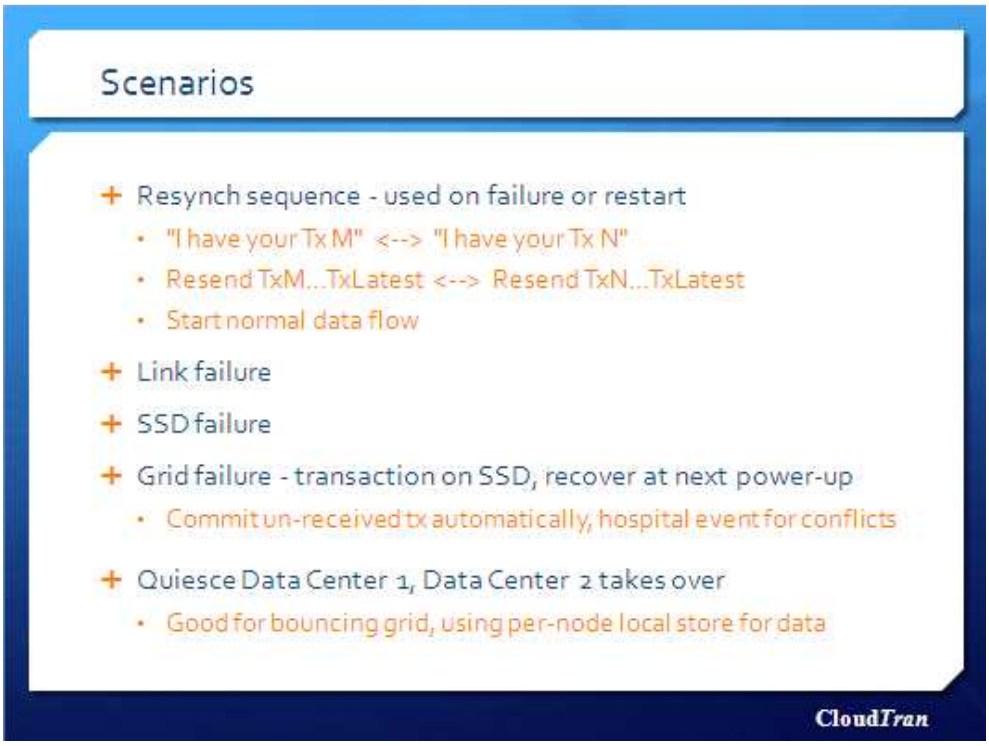
2. The drives should be dual-ported, so a newly-elected replicator can take over by reading the data just written by previous replicator.

3. Drive size dependent on how long an outage you want to handle and what volume.

   For 2KB transactions at 1000 transactions/sec, need 4MB (written in 4Kb blocks). 256GB drive can handle 64,000 seconds - 17 hours.

4. We use an append log style of writing.

   The writing uses a number of files. To recover after a crash, the new replicator searches for the most recently used file, then searches for the most recently written log. Because of the speed of SSDs, this whole process takes much less time than the recovery of the comms link.



These are various failure and usage scenarios.

The key step is the **resynch sequence**, where each replicator tells the other what the last transaction to be acked was. The replicator will start retransmitting from transaction ID.

For **replicator failure**,a new primary is elected followed by the resynch sequence.

In case of **Link failure**, the replicators at both ends run the resynch sequence and start retransmitting from the last ACKed point when the link comes back.

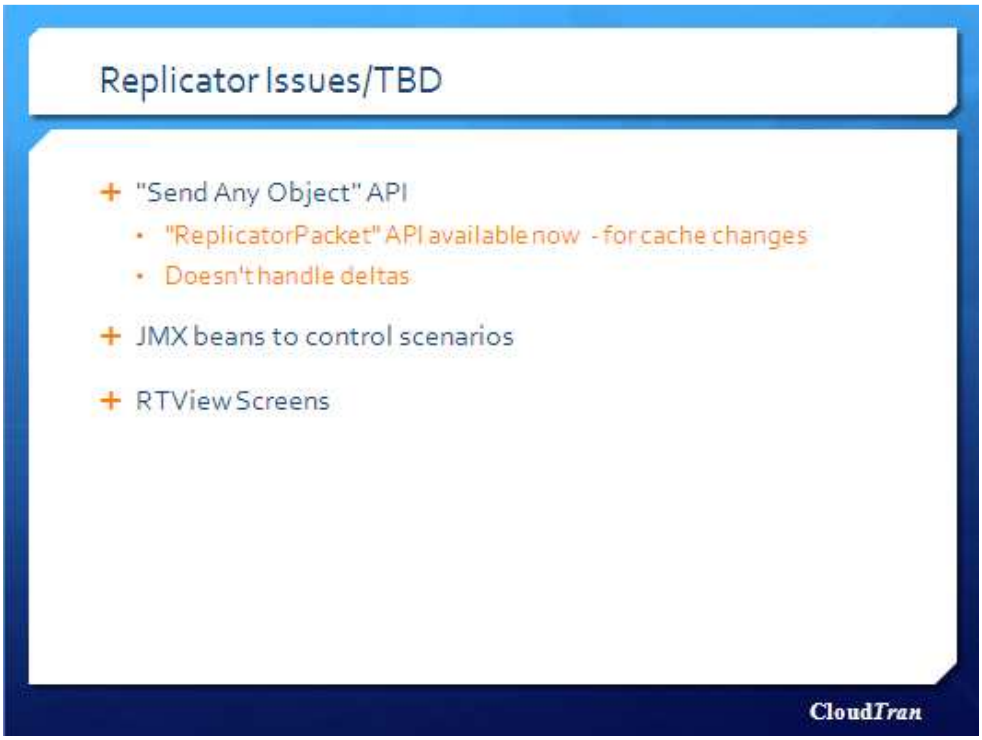
For **SSD failure**, the the replicator carries on in single-file mode – obviously this is equivalent to a partition being endangered.

For **grid failure**, the transactions to be committed are on SSD. At the next power up, there is a resynch sequence which flushes out any uncommitted transactions.

> Normally the replicator will continue sending transactions over the link, even if the grid fails. However, in the event of a complete failure (i.e. the grid fails and the link fails), you may want to run the remote grid and provide service – even though there are un-replicated transactions.
>
> In this case, there is a risk of un-replicated transaction conflicting with an update when the service is in single-data center mode. There is a hospital event to allow the application to handle conflict resolution.

A data centre can be bounced quickly by

- pausing the replicator link (at both ends)
- quescing the data centre to be bounced
- saving the data on each node - right now this is would need to be programmed, but it is planned to be a standard feature in Coherence.



Future features

1. We have been asked to provide a replicator API.

- - the current version of the product has a "ReplicatorPacket" API, which is specific to grid operations (insert/update/delete)
- there is a place for a "send a command" style of API too.

2. JMX beans to control the scenarios rather than APIs.

3. Management screens based on the JMX beans.

Summary

- Simple APIs
  - TopLink Grid
  - Low level API
  - Replicator
- High-speed, low-latency, loss-less transactions
  - Multi-cache
  - Multi-client
  - Within grids
  - Cross data centres

CloudTran