CloudTran

# CloudTran Technology Genesis

## 1% Inspiration, 99% Perspiration

**A Technology Whitepaper**

CloudTran, Inc.

4000 Pimlico Drive, Suite 114

Pleasanton, CA 94588

www.CloudTran.com

*CloudTran is the industry's only solution for managing the complete data layer in distributed computing environments. We integrate in-memory data grids with back-end data stores asynchronously and provide ACID transactionality across the entire architecture.*

## The First Vision

CloudTran was originally the invention of New Technology/enterprise, NT/e, a consulting business based in the United Kingdom. The original vision for CloudTran came during a vendor meeting about elastic caching systems in September 2008.

This particular caching vendor was encouraging developers to implement J2EE applications on their caching platform; the problem was, however, that there was no provision for general-purpose persistent transactions. The caching vendor had an in-memory view of the world, and hence they provided in-memory transactions that work on a single box or across a cluster. They also had a link to Hibernate, which helped in simple cases but did not provide general-purpose distributed transaction to databases.

NT/e's recent background was in J2EE architecture, trouble-shooting, and then automating the generation of application frameworks for enterprises. So our immediate reactions to the caching vendor's approach were:

- There was a need for an infrastructure product providing ACID distributed transactions for clusters, and,
- it needs to be highly performant so as not to throttle the in-memory application.

Our initial research in the development community showed a difference in views on such a product:

- Some said it wasn't possible to create; e.g., Pat Helland's (Amazon.com), “Life Beyond Distributed Transactions: an Apostate's Opinion” (http://www.CloudTran.com/pdfs/LifeBeyondDistTRX.pdf).
- Others said it would be better to go with the raft of NoSQL solutions and not to worry about ACID properties.
- Nevertheless, many developers at the London CloudCamp meetup in November 2008 were keen on the idea.

There are two common approaches to getting around the lack of scalable distributed transactions:

- The NoSQL approach is to manage the discrepancies resulting from transaction conflicts manually. This may work initially for smaller systems, but it is likely to cause severe problems at higher transaction rates.
- Another approach is to get heroic developers to implement complex software that looks a lot like transaction management.

The latter option is Helland's approach, but it is complicated, error prone, and crying out for a generic (infrastructure) solution.

Our interest was in the growth of transactions and application loads generated by mobile phone applications, micropayments, globalization, and the increasing acceptance of e-commerce. All these trends, and the investment in SQL-based business intelligence, gave us hope that a product linking scalable data platforms to databases at high speed with full ACID properties would enable new types of applications.

## Impossible or Improbable?

Why is it so difficult, if not impossible, to support the scalable - cloud-level - distributed transaction? Let's take a look inside Helland's paper for some clues:

> *Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various approaches to quorum. … In general, application developers simply do not implement large scalable applications assuming distributed transactions. When they attempt to use distributed transactions, the projects founder because the performance costs and fragility make them impractical.*

So the big requirements are acceptable performance and high reliability. And the difficulty is that, despite all those decades of work on 2PC etc., established approaches don't meet those requirements in the cloud or other scalable environments. Interestingly, you are guaranteed <u>not</u> to get a straight answer asking vendors about distributed transaction performance. Architects at major Systems Integration shops will tell you, and only after the direst of health warnings, that two transactions per second is about right pre-tuning, and then you might get up to 30-50 transactions per second. This certainly is not what we expect to handle "web-scale" applications!

Furthermore, classical distributed transactions only solve half the problem - they tend to work at the database level. For large high-speed systems, the in-memory data is equally important and must be considered as part of the transaction. For example, as a transaction is in the process of committing, how do we present a consistent view of the changing data across the whole grid? We knew that our vision for CloudTran would need to include a complete and consistent solution at both levels - in-memory and to the database.

There were many difficulties we encountered along the way developing the CloudTran product. Some were not so big, but others stumped us for quite some time:

- In scalable system, events that are .0000001% likely to happen on single machines ***will*** happen and <u>must</u> be catered for. In particular, failovers must be considered and handled at every step of the way.
- The need for "communications thinking," which takes into account the impossibility of two separate processes always having the same knowledge. Combining this with the failover handling requirement makes it very difficult for developers who have been used to a reliable "sandbox" provided by the infrastructure to work effectively at this level.
    - Related problems from these two are achieving bounded time for transactions, even in the event of failure.
- Handling out-of-order and duplicated messages, which occur even with the most advanced caching systems.
- Mundane things like providing consistent timestamps for debugging, generating unique primary keys for database tables, and getting start-up-sequences right for the grid to initialize correctly.

## Solution Part 1 - Divide and Conquer

To do the ~~impossible~~ improbable, we used two strategies: divide-and-conquer, and cheat.

To divide-and-conquer, we used an elastic caching platform as our base. We chose to use GigaSpaces' XAP in the early days and next worked with Oracle's Coherence. It was important to us that our design be agnostic to the caching platform overall, so that developers could pick and chose their favorites. These platforms are gradually converging on a standard set of features that are essential to CloudTran and represent many years of experience and investment. The main features we used to build

CloudTran were:

- Basic scalability via partitioning a logical data domain across multiple nodes. Crucially, this includes content-based routing to the partitions. This is conceptually the same as used in BEA's Tuxedo, which was familiar to us and well-proven in the market place
- Dynamic scaling up and down to increase capacity (e.g., in a cloud environment) on demand, based on application-defined SLAs.
- Fast failover using one or more backups of each primary node.
- A restricted SQL (as well as template-based) lookup capability.
- Local transactions.
- Services, mimicking J2EE sessions and web services.
- Simple and intuitive management of messaging (e.g. JMS, AMQP) interfaces.
- Cloud and/or data center deployment: e.g., code can be tested in one environment and then rolled out to another…or a hybrid of both.

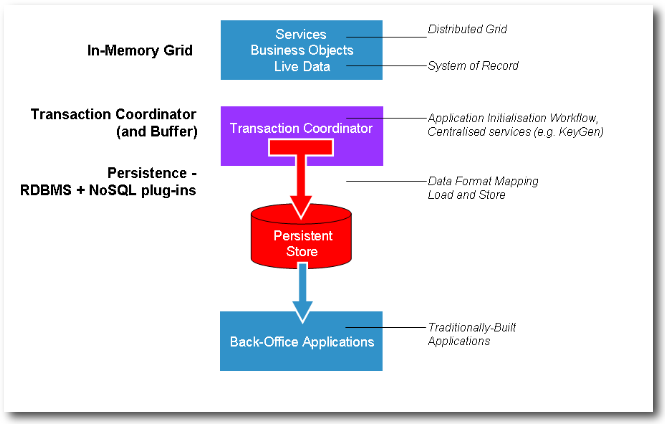
Without this basis, it is unlikely we could have implemented CloudTran: it saved us development effort, but more importantly reduced the overall difficulty of the problem.

## Solution Part 2 - Cheat

Cheating is the way to further simplify the problem!

1. As described above, we ***split the two layers of the transactions***. This allows the commit of the transaction to proceed in memory first, and then to databases or NoSQL stores in parallel. In particular, the persistent store commits eventually, but consistently.
2. ***2PC or not 2PC?*** Given the failure of the 2PC and quorum approaches in scalable environments, we rejected these in favour of a 1PC approach. Sort of. In fact, it is a story of two halves. The first half is the application-initiated phase: one component of the application starts the transaction and then takes responsibility for any sub-processes. This component watches over its children - it "counts them all out and counts them all back" - and, combined with any work it may do, decides whether the transaction is good.

Note that this half is "1PC" - this phase decides whether to commit. It avoids any quorum, voting, distributed control and so on. Note also that the application controls this,


In-Memory Grid
Services
Business Objects
Live Data
Distributed Grid
System of Record
Transaction Coordinator (and Buffer)
Transaction Coordinator
Application Initialisation Workflow, Centralised services (e.g. KeyGen)
Persistence - RDBMS + NoSQL plug-ins
Data Format Mapping
Load and Store
Persistent Store
Back-Office Applications
Traditionally-Built Applications

not CloudTran.

During the first half, the "coordinator" part of CloudTran has been the subservient bookkeeper. It has been collecting the records to commit, sent to it as they are "committed" locally in each caching node. The first half ends when the application commits the transaction to the coordinator: once this message is noted, continuing control passes to the coordinator.

The coordinator then takes responsibility for committing the transaction - naturally, even if the application crashes. Any "in-question" transactions after a client crash can be retried; they can be identified by a business transaction ID and rejected.

The coordinator has a "buffer" as well. This indicates that it can match different speeds and buffer during demand spikes in the grid, and also can continue operating when one or more data stores is unavailable.

3. ***System of Record in the Grid.*** A corollary of 1PC is that we recognize the system of record is in the in-memory data grid, and as such, programs changing the data must observe all the constraints that a database would apply.
4. ***Timeouts***. In a clustered environment, the likelihood of any given transaction being affected by an error is minuscule, so we don't want to keep huge amounts of information to be able to patch up a broken transaction. Instead, we use the principle, "if in doubt, time out." The cost of a failed transaction is negligible compared to the programming cost of trying to recover long-term, in-doubt transactions.

The above principles meant that most of the mechanics of handling a distributed transaction became fast, lean, and bounded.

## Solution Part 3 - Third Time Lucky

Working through the above principles took a significant amount of time by a significant number of folks! We worked through two detailed designs, and we're indebted to Dan Stone for trashing them both, by identifying the final problem: how to present a consistent view of data to the user of the grid while the transaction was in doubt. We ended up solving this problem as well and rolled it into the "secret sauce" that has now become the commercially available product, CloudTran.

### Additional Features

Around the core features mentioned above, there are myriad other useful features. Some we have implemented:

- Persistence-only records (e.g., for business transactions), which are generated in CloudTran but managed by another application - they don't need to be saved in the grid. This means CloudTran can be used as a high-speed front-end application, backed by slower maintenance applications and business intelligence feeds.
- Transaction logging can be done either before or after commit. This means that applications dealing with low value may choose to log after commit, reducing commit latency by 8ms.
- Reordering of SQL transactions in flight to databases. We take great care to preserve Atomicity and Isolation en route. Within those constraints, reordering SQL actions speeds up database operations. In particular, MySQL has an optimization for inserts.
- Plug-ins to persist to non-RDBMS stores, or to redirect to smart database links which could replicate to a separate data center or data warehouse as well as persisting.

We have a wish-list of some 40+ further features. One we've been working on is to scale out the coordinator, which is currently a single node (with backups) - but in practice this is not yet an issue: CloudTran does over 2,000 transactions per second today with CPU utilization at 40% - this is on single-socket machines with Gigabit Ethernet connections, leaving a lot of headroom for improving performance. Still, scaling out the coordinate would be an interesting project!