CLOUD*TRAN*

Scalable Transaction Programs in the Cloud

A Technical Overview

December 2009
Matthew Fowler
Director, NT/e

Contents

1 CloudTran : Summary

CloudTran is a middleware product that enables Java developers to build scalable commercial applications for the Cloud quickly and easily.

Public and private Clouds are increasingly popular, but developers looking to build mission-critical applications that take advantage of cloud features (scalability, commodity hardware) have faced a problem: there was no infrastructure support to

- overcome the inherent difficulties of distributed programming
- provide fast, reliable data transactions combining memory nodes and storage services.

Now CloudTran overcomes this problem with its unique features :

- ORM : the Object-Relation Mapper layer provides a simple Java view of distributed data. It finds data in the grid, collects it for the application and then redistributes any changes;
- In-Memory - Storage : CloudTran automatically transfers data between storage services and in-memory data formats. RDBMSs are supported by default and plug-in points are provided for non-RDBMS storage systems;
- Cloud Transactions : CloudTran is the first Transaction Monitor specifically designed for clouds, providing ACID guarantees coordinated across distributed nodes, messaging and storage services.

The solution is highly reliable as it provides a buffer between the customer-facing application and back-end resources, such as databases or transaction systems. This means that the application runs even when there are spikes in traffic or back-end resources are unavailable.

CloudTran complements other options for application development in the cloud. It is built on the GigaSpaces eXtreme Application Platform (XAP) and so can be used to build scalable, superfast (sub-millisecond) applications. It can also be used in conjunction with Storage As A Service. So architects can combine all these options to achieve price, performance and quality of service goals (see Appendix 2).

This technical overview is for architects and developers of applications who are tasked with building commercial applications in public or private clouds, and are interested in middleware solutions to deliver higher performing applications at greatly reduced cost.

2 The Story Behind CloudTran

CloudTran story started when I was in a GigaSpaces update in September 2008, about their "application server" features - how to port your J2EE application to work in the cloud and be scalable.

The "transactionality" word was used ... but this was used in the sense of "in-memory transactions". Nothing that old men with an ACID database would think of as a transaction.

Coming from a J2EE background, I looked for a solution to allow application programmers to easily write big, fast applications on the cloud using transactions. The more I looked, the more it seemed there wasn't one, and no appetite for one either. There were three stances from the cloud camp:

- "You don't understand" - you don't need to persist memory-base transactions to a database, because the grid is so reliable.

  Right [I thought] - tell that to your CIO.

- "Eventual, non-consistent save is good enough".

  Well, that's at least got the 'D' part of ACID ... but nothing else;

- NoSQL - solve the scalability and consistency problem in a different paradigm.

  Shame about the databases and information feeds that drive commerce and decision-making today.

There was also deep pessimism about distributed transactions in a cloud environment, e.g. "they're too slow and complicated" (Höller) or "too slow and unreliable" (Helland). So it sounded like a problem crying out for a solution - even if it was difficult!

As it happened, working on the problem was slightly trickier than we thought - it took us three attempts to get a workable design! Design 1 failed the "atomicity" and "simplicity" tests. Design 2's problem was coordinating failover and isolation in distributed nodes at a reasonable cost and speed. Design 3 (like Goldilocks porridge) was just right. Our thanks to Dan Stone (of scapps.co.uk) for: destroying designs 1 and 2; pointing out many abstruse wrinkles in design 3; and instilling the "mission-critical" values needed to persevere with achieving reliable transactions.

The other part of CloudTran is the O-O programmer view. When I read Pat Helland's *Apostate* paper, my view was that his proposed solution - essentially, application programmers doing robust distributed programming at a low level - was not the best way. We have been building model-driven development environments for the last 8 years, and it was relatively easy to port this environment across to the CloudTran environment.

3 Terminology

In the technical discussions in this paper, we use ***grid*** and ***cloud*** interchangeably: they are groups of machines connected by high-speed LANs, with additional machines readily available. In other words, the commercial differences between grids, public and private clouds are not relevant to our discussion. The key characteristics are the high speed of the interconnect, so that the groups of machines can form a largely (but not completely) reliable unit.

There are ***data grids*** and ***compute grids***. Many companies deploy these as distinct layers, so that large calculations can be farmed out across the compute grid, using data that is accessed easily from the data grid.

This sort of architecture is not optimal for applications where calculations are simpler: then the requests from the compute grid to the data grid become a performance bottleneck. Applications are quicker and cheaper if the compute and data functions are combined into a single grid. This is Google's approach to building large-scale systems; it is also a key component of ***Space-Based Architecture (SBA)*** discussed below.

The other major architectural shift behind CloudTran is the need to store the ***System of Record*** in the data+compute grid. The System of Record is the permanent, live data - the lifeblood of many organisations. As we will see next, there are situations where this is the only way to meet size and performance requirements. We use the term ***In-Memory Data Grid*** (IMDG) to mean a data+compute grid holding the system of record.

The IMDG collapses the architecture so (a) all necessary information is in-memory and (b) it removes the communication hop to the database and its slow disk accesses. The bottom line is that IMDG systems are 10- or 100-fold faster than their tiered, database-base cousins.

4 Commercial Drivers for In Memory Data Grid (IMDG)

There are a number of reasons to use an IMDG. This section looks at the drivers, and the cost-benefit equation.

4.1 IMDGs When All Else Fails

A common pattern in the evolution of successful applications is to

1. Start with a simple database architecture
2. Address performance issues with caching
3. Finally, when the size of the application and its user base reaches the point where the user experience degrades unacceptably, move to an IMDG.

Many developers apply a lot of effort in stage 2 - using all the tricks in the book to avoid using the IMDG approach. So IMDGs have been the final resort.

Of course, using an IMDG is not acceptable for every application. YouTube wouldn't dream of putting 50MB videos into an IMDG. A free "store and share your photos" site couldn't afford to put billions of photos into memory. However, commercial data-oriented applications usually do not have such large data requirements. A company with 100,000 employees and contractors could store the complete employee database, at 5Kb per record, in 500Mb - which is certainly amenable to in-memory storage.

Relatively few applications today need to use an IMDG. However, many forces are driving more and more applications in this direction, such as

- the size of markets;
- the continued growth in worldwide numbers of mobile phones and their use for charged transactions and data applications;
- the increase in the number of automated transactions.

4.2 IMDGs As A Competitive Weapon

The previous section viewed the IMDG as the last line of defence, but it is possible to turn defence into attack.

Consumers are getting used to fast response times and are increasingly frustrated by slow sites. In 2006, (http://www.akamai.com/dl/reports/Site_Abandonment_Final_Report.pdf) 28% of visitors would abandon a site if a page took longer than four seconds to load. By 2009 (http://www.akamai.com/2seconds) the figures were 40% of visitors leaving after 3 seconds.

Fast response times don't just stop visitors leaving: they can set an application apart from its competitors, and create a positive and enjoyable experience that draws visitors back to a site. Using an IMDG architecture in an enterprise application gives the platform for performance and fast response times to users.

4.3 The Cost Of IMDGs

My instinctive reaction when I first heard about IMDG was, "that's crazy - it's going to cost a fortune". But those instincts were forged a long time ago - when main memory was extremely expensive.

The world has changed a lot since RDBMs came to market 30 years ago - so much so that Stonebraker *et al* (http://db.cs.yale.edu/vldb07hstore.pdf) wrote in 2007:

> we believe that OLTP [OnLine Transaction Processing] should be considered a main memory market, if not now then within a very small number of years. ...
> In summary, 30 years of Moore's law has antiquated the disk-oriented relational architecture for OLTP applications.

So let's do a napkin calculation to get the order of magnitude costing for an IMDG approach - what does it cost to hold our 100,000 5Kb employee records per year in an IMDG?

To install 64GB in a server (from a large manufacturer) costs roughly $8,000 at today's prices. This means the 500Mb for the employee database costs $62.50, or $125 when you add a backup (as you would). Of course, that is a 'raw' cost - the 'usable' cost after Java's overheads will be higher by a factor of 3 or so.

Say you bought 10 machines, 5 primaries and 5 backups, giving 320GB of raw information, or say 100GB usable. The IMDG memory would cost $80,000 - and give ultra-fast access to 20 million 5Kb records.

4.4 Tiered Storage In The Cloud

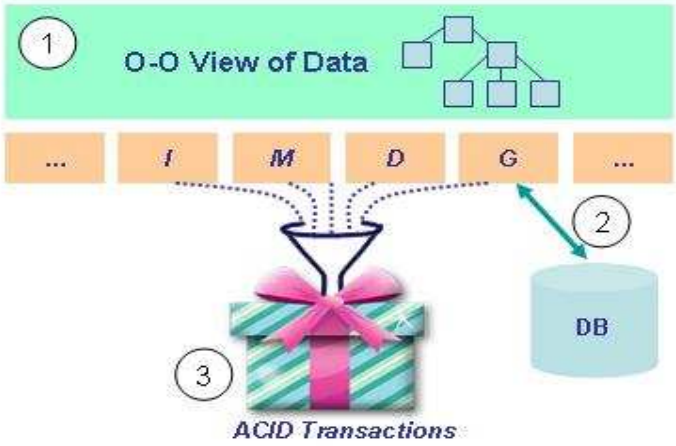
The above discussion has focussed on the sort of information used in transactions, while noting that storing large lumps of data in the IMDG wouldn't be economic for many applications.

However, CloudTran can integrate with disk storage as well. For example, our server could have 1TB of hard disk attached, for another $800 - 16 times the data capacity at 1/10th the price. Commercial clouds like Amazon EC2 have attached disk storage that effectively comes for free. This gives an extremely low cost for storing large objects - the cost mainly being the programming.

Alternatively, an application could use a mountable file system that could be provided in private or public clouds, such as Amazon's S3 (Simple Storage Service).

For applications that have a "primary" transactional data component and a "secondary" blob component, an IMDG plus disk-based storage makes a very attractive scalable, high-performance, high-memory system. Some of the data stored transactionally would be meta-data for the secondary storage area, allowing the application to manage large volumes of storage without storing it all in the IMDG.

5 CloudTran Design



The functionality of CloudTran is divided into three layers and the next three sections describe these in turn :

1. The Object-Oriented View of data, so application developers can do their job without having to worry about distributed programming, but still take advantages of its performance benefits.
2. Data mapping - the programs operate on an in-memory datastore replica mapped back to a real datastore for persistence and links to database integration applications (eg ETL, data warehousing and BI). This implements a two-layer, primary-secondary, data mapping feature, with the system of record in the IMDG.
3. Transactionality - the most generic layer.

6 Cloud Transactions

6.1 Transaction Support

CloudTran is an add-on product for GigaSpaces XAP which provides highly performant, mission-critical and resilient transaction support (*local transactions*). In other words, one or more create, read, update or delete operations can be joined together by a space-based transaction, so that all operations succeed or fail *atomically*. By providing various locking schemes, space transactions also support *isolation* between the different operations of a transaction. Furthermore, there is a sort of *durability*, in that backups will also be written of the information in a space. The last part of 'ACID' - *Consistency* - is left as an exercise for the developer.

These transactions can also be distributed across spaces, and are then called ***distributed transactions***. These transactions have the same properties, but at a significant performance cost (see http://blog.scapps.co.uk/?p=19)

If you are an architect or application programmer for data-oriented applications, at this point you may be experiencing the same cognitive dissonance that I did: there's no database or other persistent storage with these "transactions". To avoid conflict with these transactions is part of the reason for talking about "cloud transactions".

Finally, there is a connection to persistent storage in GigaSpaces. This is the GigaSpaces mirror feature which is done via a single connection to a database using a Hibernate mapping, with approximate "transactionality". The mirror service will typically be used in conjunction with local or distributed transactions and therefore provides strong isolation at the space and durability - the 'I' and 'D' of ACID - but no guarantees of atomicity or consistency **at the database**.

So who needs strongly transactionality all the way to the database? And furthermore, who needs a database anyway?

There are definitely people who don't, but an awful lot of people who do, for whom it is a "Door 1" issue. Events that will never happen - but do – include :

- Intermittent hardware failure causing outages - for example, Amazon S3 outage on Jun 23, 2008 - which in an IMDG will cause eventual data of the System of Record;
- acts of nature or accidents disrupting power to data centres;
- the electrician cutting the wrong cable - everyone's got a story.

Our aim with CloudTran is to make robust, transactional persistence so easy that it is a no-brainer for application teams.

6.2 Cloud Transactions - The Hard Truth

> ***The general problem of cloud transactions hasn't been solved before CloudTran.***

If coordinated transactionality at the space and at the database is required, there is a two-layer transactionality problem:

- there is the copy in the space, in-memory - this is what you need to provide performance;
- and there is the copy in the persistent store(s), which is durable even if the power is switched off.

These two versions must be coordinated with full ACID properties.

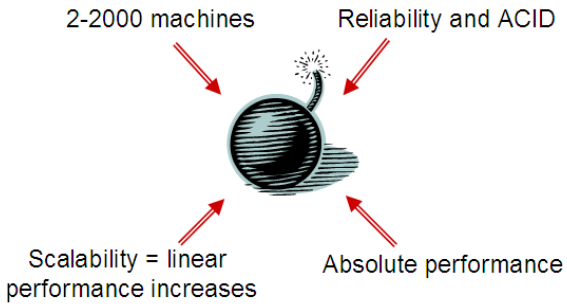
Classical 'distributed transactions' provided the ACID properties across multiple processes or machines. Now, in adding the IMDG layer, we are adding another dimension to the transactionality problem.

The transactionality problem is not restricted to the GigaSpaces SBA platform. You would imagine that cloud providers would have strong transaction solutions, but this is not the case. Apart from 'native' connections to a single database, there is no generic, usable transactionality capability.

Classical "distributed transactions" should be the answer. However, in the seminal paper, *Life beyond Distributed Transaction* (at www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf), Pat Helland admits he has spent many years trying to get distributed transactions working effectively in large environments ... but concludes

> 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. Natural selection kicks in…

... and natural selection is where the less fit don't survive. This diagram illustrates the deadly forces on the application designer :



CloudTran provides transactionality that will help you overcome the challenges of scalability, distribution and performance in scalable applications.

The fact that there is no general solution to the cloud transaction problem is a symptom of how difficult this problem is. Here are the key design points:

- support services writing into any number of nodes in the cloud, from 1 to 100 or more;
- handle very small and very large transactions. We have written applications writing in excess of 50,000 rows - so we'd like to do the same or larger on the cloud;
- minimise overhead for the happy path, because most transactions will succeed;
- roll back across all nodes in the case of failure, even if a node fails as you are trying to restore its state, to guarantee Atomicity in failure situations;
- ensure all users of objects involved in a transaction in different nodes get a consistent copy of the data, possibly delaying the delivery until the up-to-date data arrives at the node (Consistency). This is not just an issue in a space-based architecture: the problem will arise in general, because developers will want to use local copies of data, whatever form they are in (i.e. space or not). This means that the extra dimension of transactionality - coordinating database and in-memory versions - applies generally in cloud-based applications;
- make sure updates from different transactions don't stamp on each others' data (Isolation);
- coordinate acidity at the space level with the Durable version of the transaction;
- handle "possible failure" - GigaSpaces spaces may 'scale out' and be unavailable for a variable number of seconds, so you must handle temporary and permanent outages;
- provide timeouts, so an out-of-control transaction doesn't lock up resources.

6.3 CloudTran Transaction Support

CloudTran provides scalable, rock-solid, high-performance distributed transactions.

The transaction layer in CloudTran meets all the design goals set above:

- Handles any number of nodes and any amount of data in a single transaction.
- Fast commit mode. Although the usual 'secure transactions to disk' mode can be used, developers can also use 'secure transactions to backed-up space'. This mode is about 5

times faster than the 'disk-first' mode and allows a single-CPU machine to support thousands of transactions per second.

- Supports scenarios to coordinate message passing with persistence operations
- Plug-in architecture to support any type of persistent store. JDBC connections to RDBMSs are provided as standard.
- Handles any number of persistent stores. Consistency is supported across databases.
- Buffers transactions that are to be persisted. This means the main application can
  - handle demand spikes without getting delayed waiting for the database
  - continue to run at full speed even when the database is down.
- Restores in-flight transactions from a backup copy at the transaction controller if a primary node fails over to a secondary.

7 The IMDG-DataStore Mapping.

CloudTran automates the definition, load and storing of the IMDG to a datastore.

The second layer of functionality in CloudTran is to map between the datastore view and the IMDG view of data.

Having the "primary" data in an In-Memory Data Grid is the key factor in the high performance of the complete CloudTran solution. In many cases, the persistence target for the data will be a database. However, other types of stores, such as BigTable/Hadoop, are becoming popular, so we cater for those too.

To support this, there are a number of unglamorous mapping and support jobs that need to be done by this layer:

- Creation of Java objects to represent the in-memory data. It does this using code generation, which we prefer to AOP (Aspect-Oriented Programming) approaches - developers can see the live code and have the opportunity to alter the generation templates for special situations;
- Data in the IMDG is represented in user-natural format (not datastore format). However, it also uses foreign keys (like databases do) rather than object references to represent relationships;
- Addition of fields for optimistic locking - typically the row version number;
- Initial load of the grid from the database, after power failure or data restructuring;
- Store of each objects as they are created or update, over JDBC;
- For non-relational datastores and messaging interfaces, this layer defines the plug-in points and interfaces for objects to be included in a transaction, and for the store-specific load and store;
- Distributed primary key generator. Different keys, which can be integer or long values, are kept for each target database/datastore via a service in the transaction buffer.

Where the database or JDBC are mentioned above, there are plug-in points to support other types of datastores.

In future versions, we will add support for a "local load" and "standing data". "Local load" addresses the issue that the initial load of the whole grid from a datastore will be a very lengthy process in some cases. Storing the information on each node will make the load much faster. "Standing data" will be a marker on an entity that this table is small and read-only, so should be distributed from the database to all nodes.

Note that while the IMDG is implemented in Java, this does not preclude .NET clients. This is another feature for future versions.

8 The ORM (Object-Relational Mapping) Layer

The ORM layer of CloudTran gives application developers a higher level platform to work on. As far as possible, the ORM layer provides a complete Java development view.

From the application developer's point of view, the ORM occupies the same slot as Hibernate and JPA. However, because of the difference in the underlying subsystems, the implementation of the ORM in CloudTran is quite different and some concepts are different.

8.1 Something Old, Something New... Modeling and Generation

> ***CloudTran provides modeling tools for data definition and configuration management.***

GigaSpaces is a new platform in its own right and it takes some learning. CloudTran removes the need for application programmers to learn GigaSpaces immediately; using concepts and vocabulary they are used to, programmers can be immediately productive. For example, some of the user concepts in the modeler are entity, relation, subscriber, receiver, service and business-method.

CloudTran provides tooling in the form of an Eclipse-based modeling plug-in and automatic generation of the specialised classes required for the application. From an entity specification, CloudTran generates the IMDG data class, the loader and store data classes, the Entity object (the user view) and internal services to handle cross-node interactions. From other aspects of the model, CloudTran produces XML configuration files for GigaSpaces and the application framework. What is left for the developer after generation is to fill in the business logic.

8.1.1 Configuration Management

As well as modeling the structure of the application, developers can model configurations. It is common to have a number of different deployments, for different phases of development and deployment. Modeling the deployments gives a record for future reference - as well as generating correct deployment scripts for different environments. Currently, the deployment targets are Eclipse, Unix/Linux and Amazon EC2.

8.2 The Entity Group Pattern

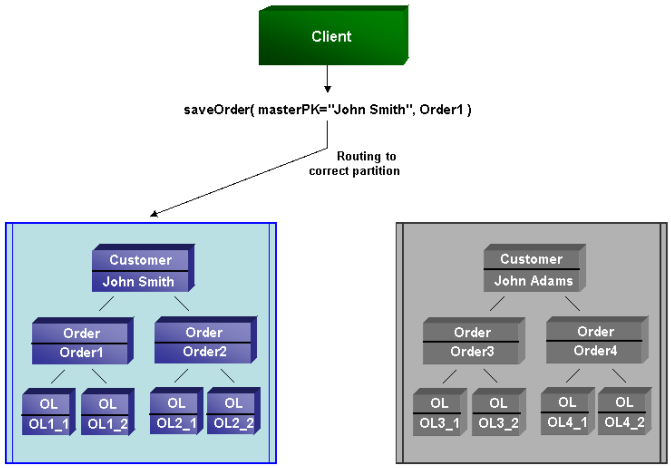
> ***Entity Groups help architects optimise locality of references between data objects.***

A common pattern in scalable systems is to group entities that are tightly coupled into ***entity groups***. (We started using this term internally in the CloudTran development, and then discovered it is well-known!)

The reason for using entity groups is to put instances of related entities together into a single space, so that references between the entities is localised. The key goal of the SBA approach is to co-locate objects, thus avoiding the overhead of serialization, network hops and tiers of functionality; entity groups do this as far as possible for entities.

The grouping is application-specific, and up to the developer's judgement. For example, in a Customer system, one approach ('***Customers with Orders***') is to group Customer, CustomerAddress, Order and OrderLine into one group. An equally valid approach ('***Orders separate from Customers***') would be to split the group: Customer/Address into one entity group; Order/Line into another.

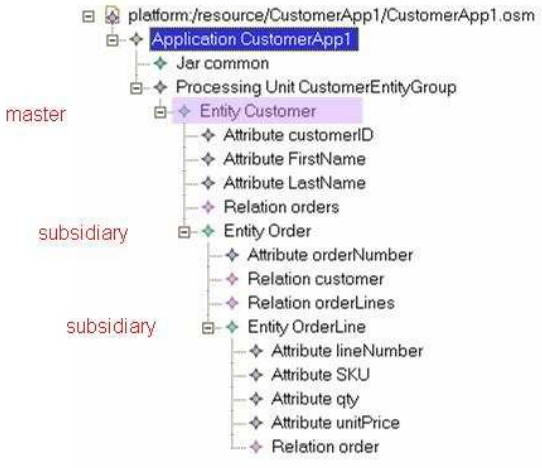
For scalable systems, entity spaces will need to be partitioned. To route to the correct partition, there must be a single routeing value for all instances of an entity group, to ensure that they all end up in the same partition. In CloudTran, this is done by using the primary key of the ***master entity*** for routing to a partition.



The diagram shows a space partitioned in two. Customer is the master entity of the entity group; we refer to the non-master entities in the group as ***subordinate*** entities. Customer John Smith and all subordinate entity instances are collected in one partition; John Adams and all subordinate entity instances are in the other. The client, to save an order, must ensure that the request can be directed to the space where John Smith's live entities are.

This is done by including the master entity's primary key in the call (in reality this would probably be a unique customer ID rather than a name). The parent/subordinate relationship is easy to express in CloudTran's modeler:

- subordinates are nested inside their parent. In the diagram below, entity Order being nested underneath the Customer entity means that Order is subsidiary to Customer. Similarly OrderLine is nested beneath Order, so OrderLine is subsidiary to Order, even though Order is not itself the master entity.
- there must be a to-one required relationship from the subordinate to the parent - such as Order.customer and OrderLine.order.



8.3 Distributing and finding data

> ***CloudTran provides a simple data view for programmers, solving both well-known and new issues in data management.***

In general, the instances of an entity, or entity group, will be spread across multiple spaces. This means that the IMDG version of an object must record relationships using foreign keys, and a Data Access Object (DAO) must present a façade of in-memory references, constructed from the foreign keys. CloudTran supports the DAO with standard ORM features such as

- mapping between the two versions of objects (client-side and DB-side)
- lazy loading and bulk loading
- cascade delete facilities, which are relevant even if an RDBMS is not used
- actioning validation and integrity constraints.

CloudTran provides additional features for issues that arise in distributed systems:

- the problem of locating the data. A foreign key to a related entity is just a number. This must be converted into a way of addressing a particular node to access the information; it turns out that accessing a master entity by foreign key can be much more efficient than the general case, so CloudTran generates different code for these two cases.
- consistency. We have mentioned consistency in the context of creating transactions. However, in building a Java view of the information in the system, there is the possibility of getting inconsistent information due to accidents of timing between the updates and reads. CloudTran provides the fast (live dangerously) and plodding (correct, by using a distributed transaction on the read to ensure consistency).
- error handling and timeouts. CloudTran uses a philosophy of fail fast here. In other words, any error causes a whole distributed activity to be aborted without retires (in other words, "don't chase the acknowledgement"). The net effect is that the CloudTran features and philosophy make error handling in a distributed environment as simple as it can be.
- the optimistic locking pattern, which is necessary to support the "go away and fill in the insurance form" type of usage: the data used by the actor may have changed in the

meantime. Although applications can choose to keep the complete try locked (and this is certainly possible in CloudTran), most architects prefer an optimistic locking.

CloudTran must generate a version counter for each record in the IMDG to support this. Reflecting this to the database is not strictly necessary, so it is an option - auditing and statistics gathering may find the information useful.

While there is a lot of detail in these features, most of them are hidden by the ORM.

We try to make the API as simple as possible:

- recognising there are conceptual differences in the environment that must be exposed
- but taking advantage of the conceptual dependence on database technology to make the data specification easier.

9 Conclusion

There are increasing pressures on applications to move to an IMDG approach, to meet scalability and performance targets. Small data applications (using gigabytes) are economic off-the-shelf today; large data applications can program the use of cloud secondary storage to achieve a cost-effective solution.

However, the IMDG approach is fundamentally a new programming model. To bring this approach into the mainstream, application programmers will require a layer of infrastructure to provide usable cloud transactions, coordination of in-memory and persistent stores, and simplification of the programming environment.

CloudTran provides all the help application programmers need in a single integrated package. Whether application developers are looking to achieve competitive advantage by using IMDGs or have simply run out of options, CloudTran makes the conversion to this new approach low-risk and easy.

Appendix 1 – Performance Tests

As of this writing, very small distributed transactions commit at just over 2,000 transactions per second in a fully-backed-up environment. The configuration for this test is :

- a single (un-backed-up) client that starts the required number of transaction threads and has the logic to check that what is written at the service agrees with the database;
- a backed-up cohort running a 'service' that starts a distributed transaction, enrols its own node into the distributed transaction and then performs the write of random information (a price tick on one of a number of instruments) before committing the transaction;
- a backed-up transaction buffer, which receives the start and commit commands, coordinates the distributed transaction and, asynchronously, sends completed transactions to persistent store - in this case a MySQL database;
- the main boxes were Intel i7 920 single-socket (but quad-core) connected by GigaBit Ethernet. The primary and backup copies of the cohort and transaction buffer nodes were on different boxes, and so were the primaries of the cohort and transaction buffer. This meant that every inter-node message went across the wire.

The headline number before serious performance tuning was about 300 transactions per second. The biggest bottleneck we removed was a single thread to transmit commit/abort messages from the client to the cohort. This made the transactions per second jump from 900 to over 2,000 per second. The other major change was to run the right test program - the 300 figure was with a variant of the test program that specifically designed to cause interference on specific records and test Isolation. There is still more performance improvement to do - our test system was at less that 60% CPU utilisation, so 4,000 transactions per second peak may be achievable.

The handling of logs (writes to a local file system, protect against power failure) is also crucial. The numbers quoted are for a 'write log after commit' mode, so the client program does not have to wait for the log before continuing. We suspect this will be the most popular mode of operation - but it does introduce a second or two of unprotected time, in case of complete power failure. For extremely valuable transactions, there is a 'write log before commit'. This reduces the number of transactions per second to what the disk system can handle; on our test system, this was around 250 transactions per second. There is also a 'no log' mode, which gives about 20% boost in performance.

Along the same lines, we found a huge difference in performance of MySQL depending on whether transaction logging was enabled. If it is not enabled - which is the default - the MySQL engine can handle many 1000s of transactions per second. If transaction logging is enabled, the performance drops back to a few hundred transactions per second.

Doing performance testing has emphasised the need to protect CloudTran from excessive demands from clients. Without this, spikes in demand can cause throughput to drop; we also have a worry, based on previous performance testing, that the TCP/IP stack can deteriorate under excessive load. We provide tuning parameters that allow the cohort and transaction buffer to internally retry to allocate resources if they are not immediately available, in order to reduce the number of possible retries and overall network traffic.

Appendix 2 – Positioning Relative to GigaSpaces XAP and Storage as a Service

	GigaSpaces XAP	***CloudTran***	*SaaS*
App Latency	Sub-millisecond	**10s of ms**	
App Throughput	> 20,000 tx / second	**1000s / second**	100's
Data Size	100's GB	**100s GB**	multi-TB
Domain Model	<10 key classes	**> 20 classes**	n.a.
Values	Performance worth $m	**Fast, reliable service**	Low cost, big data
Quality of Service	Scalable, availability	**Scalable, availability, transactionality**	Get what you pay for