CloudSave Overview

1. Goals

CloudSave is a sort of 'object-relational mapper' (ORM) for Cloud and Grid architectures. It is a software layer used by data-oriented applications (e.g. a typical 'shopping basket' web application) that provides the following features:

- for arbitrarily scalable applications
    - provides fast, reliable distributed transactions
    - a Java view of data, with relations that span computers in the grid
    - defines a strategy for placing data across nodes in the cloud to maximize performance
- for time-critical applications,
    - improves performance by bringing data and processing power together on the same machine (rather than having separate 'compute' and data grids). This applies to both data accesses, "business object" functions and generic services
    - removes write latency, via a transaction coordinator and buffering service.

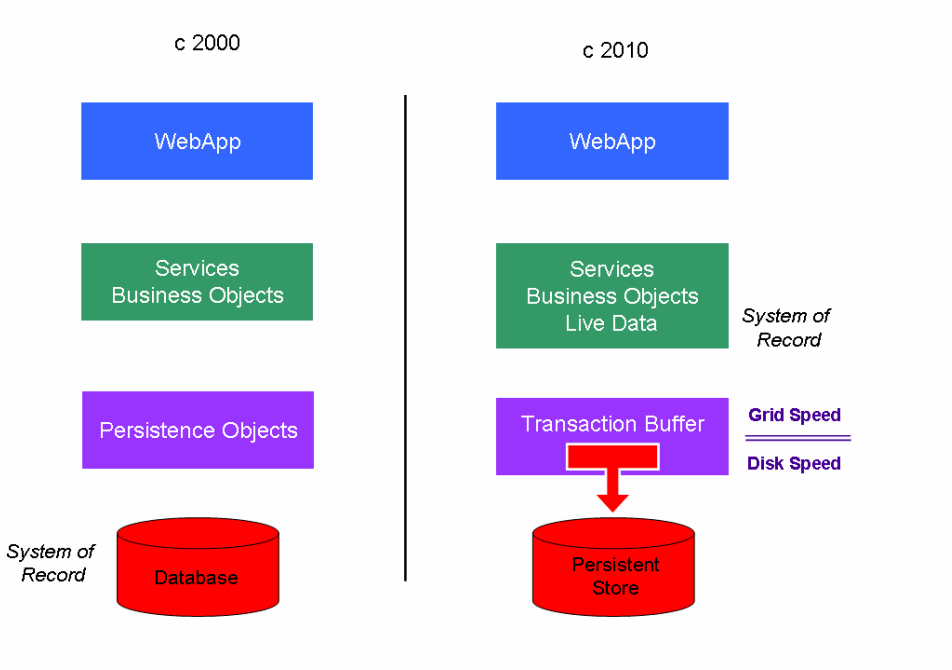
Furthermore, the system is designed to be used with minimal training for developers. The details of the Cloud implementation and configuration are largely taken care of by an Eclipse-based modeler and code-generation capability; these are separated out from the business logic of the application.

1.1 Good-bye to the Noughties...

The typical enterprise architecture of the 'noughties' (c. 2000-9) has multiple layers (web, service, persistence) and data management dependent on a transactional database. The overheads of communication between the layers and time for transactions to the database make this architecture slow and difficult to scale.

Grid/cloud facilities support a different architecture: the "system of record" can be maintained in memory alongside services and business logic, enabling much faster processing with a simpler architecture (see Nati Shalom's article on this approach).

In other words, we want to move the "system of record" - i.e. the master copy of the data - from the database (in the c. 2000 version) into the mid-tier (c. 2010), and use the database as a time-delayed record of the live data in the application:



The 2010 architecture collocates the live data and business services in a mid-tier grid for use by web applications or other clients. This collocation means that information processing applications can operate much faster than if they have to load or save information from disk.

CloudSave helps developers build and deploy seriously large and fast applications in the new architecture. It operates on service and persistence operations (i.e. across all the new layers). Rather than using an object relationship mapping (like Hibernate), CloudSave provides a "transaction buffer". Its job is to accept transactions for persisting to a data store (which is probably an RDBMS, but not necessarily). The key point is that transactions are accepted with very low latency - without waiting for writing to disk, or transactional commit to the database. In other words, the transaction buffer is like a gearbox, matching the speed of the grid to the speed of the persistent store. Final persistence is done by plug-ins for persistent stores, like databases.

2. The Big Problem

Most J2EE developers we've talked to are suspicious of the persistence capabilities of grids/clouds. In building a mission-critical application, they need to trust that their data is as safe as a database can make it, because sales, purchases and other transactions tend to be commitments by a company to deliver or do something. The mindset of trading systems - "if the data's half an hour out of date, it's not relevant so we can start again" - doesn't cut it.

The view of J2EE developers is mirrored by the experience of current practitioners. Pat Helland reports similar concerns (in his "Life Beyond Distributed Transactions" paper):

> Unfortunately, programmers striving to solve business goals like eCommerce, supply-chain-management, financial, and health-care applications increasingly need to think about scaling without distributed transactions. They do this because attempts to use ***distributed transactions are too fragile and perform poorly***.

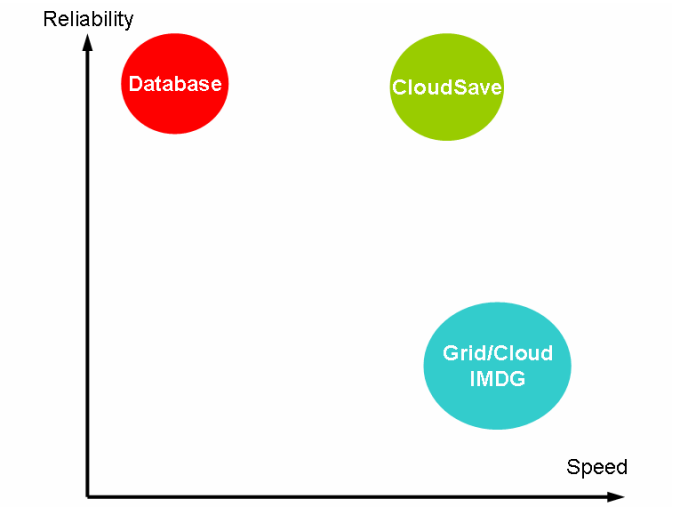
The basic idea of CloudSave is to win over application developers by being "**as safe as a database, as fast as the Cloud**". In other words, CloudSave provides distributed transactions that are reliable, but also perform at Grid speeds, as illustrated in the diagram below.

Looking at this diagram from the "database" circle, CloudSave offers

- much faster response times
- an architecture that can scale arbitrarily, to cope with large datasets or very low latency requirements.

Looking at the diagram from the "Grid/Cloud IMDG" spot, CloudSave offers

- security of persistence
- a strategy for distributing data in an IMDG (discussed later).



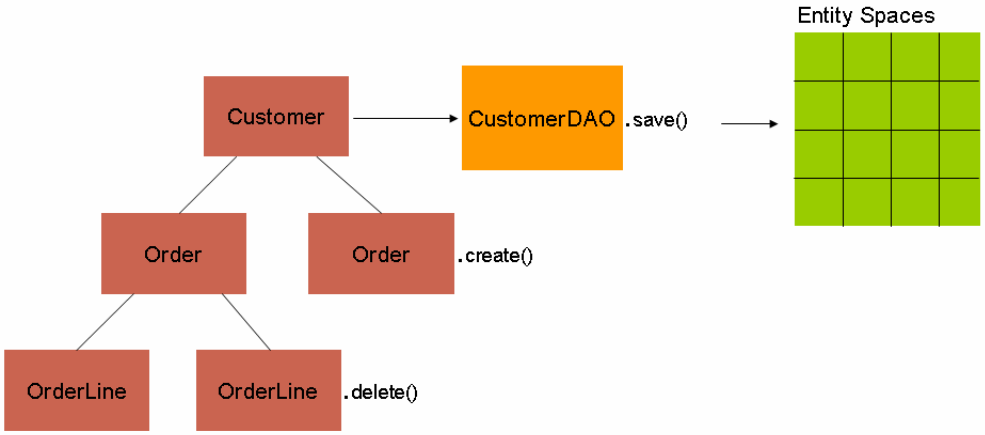
3. Platform and Features

3.1 Basics

Using the GigaSpaces platform, CloudSave provides object-to-store management features.

To develop applications, the modeler and programmer use:

- entity-relationship data modeling
- Java-based persistence. The programmer does operations on specific entity objects - e.g. order.create(), orderLine.delete() or changes to the Customer - and then saves, in one operation, a complete tree of objects via a complementary DAO (DataAccessObject).



Entities are mapped to GigaSpaces spaces as follows:

- A Processing Unit (PU) typically contains a single space and each PU runs on a dedicated virtual machine.
- Each space holds the data and services for one or more entities. This means that there is at least a serialization 'hop' required to access data between spaces and most likely a network call as well.

- Arbitrarily large data sets will exceed the memory capacity of a single machines so they must, in general, be mapped to partitioned spaces.

  Partitioning is the splitting of one logical set of data into multiple object spaces. There needs to be an unambiguous way of locating the space for a given object - this is the routing information. We recommend a numeric primary key on entities, and this is the natural way of unambiguously locating the space that holds an entity.

If the above sounds difficult, it is not: most of the details are taken care of by GigaSpaces, using generated deployment scripts.

The programmer accesses and manipulates information on the entity using standard getters and setters - customer.getName(), order.setOrderDate(date) - in classes generated by CloudSave. The values of relationships are also accessed by getters: to-one relations, like order.getCustomer(), yield a single value; to-many relations, like customer.getOrders(), yield a Collection.

To make *potential* changes to the persistent tree, the programmer uses Java features. To-one relationships are created or updated by setting a non-null value - customer.setAddress(newAddr) - and removed by setting a value null - customer.setAddress(null). Membership of a to-many relationship is changed by using add or remove on the collection object: customer.add(order).

To make these changes permanent, the programmer must persist a tree of objects through a DAO: customerDAO.save(customer). The DAO is also generated by CloudSave. CloudSave takes responsibility for saving the tree of objects at high speed in a secure transaction.

3.2 Additional Features

The previous section described features that are familiar to users of ORMs like Hibernate. There are additional features in CloudSave that may be unfamiliar, driven by the distributed space implementation.

- To improve the speed of operations, related entities can be grouped together in the same space. For example, in a customer-focused system, the related entities around a particular customer - Customer, Order and OrderLine (and possibly CustomerAddress) - could be held together in the same space, making the application run faster because inter-object references are in the same VM.

  These groupings are called **entity groups**; the instances reside in **entity (group) spaces**. We discuss the mechanics - which are easy - in more detail below.

- Data is persisted transactionally. Transactions are allocated with a unique **business transaction ID**, which is used by CloudSave to coordinate and rollback transactions across entity spaces.

  To read entities only - with no guarantee that there will be information saved at the end - there should be no transaction.

  To start reading information that may end up being updated, the programmer starts the transaction. Entity instances that are read are now automatically locked for the use of this application, and remain locked until the transaction ends; when there is a conflict, a RecordInUseException is thrown.

  In other words, this scheme automatically supports the scenario

  - read information

- go away for some time while the user thinks
- re-read information in preparation for persistence.

- There is a particular problem in large distributed systems of generating **unique IDs**; CloudSave provides helper classes that generate unique IDs for "autokey" entities (where the primary key is automatically generated by the infrastructure); the programmer can override the default implementation if required. The underlying infrastructure can also be used by programmers to generate unique IDs for their own purposes.
- **Referential integrity** is automatically provided (which, thankfully, is too detailed to discuss here).

4. What Must Be Done

Why can't we just use Hibernate, or some other object mapping facility?

There are two situations where CloudSave adds unique capabilities:

1. **Reducing Write Latency.** For the application to be sure that a transaction has been committed on a shared database, it must make at least one network call and wait for the transaction log to be written to a hard disk.

   CloudSave writes to a transaction buffer space, which is much faster than writing to disk - data centre networking hardware seems to have a round-trip overhead for IP packets of the order of 50μs (as of this writing!).

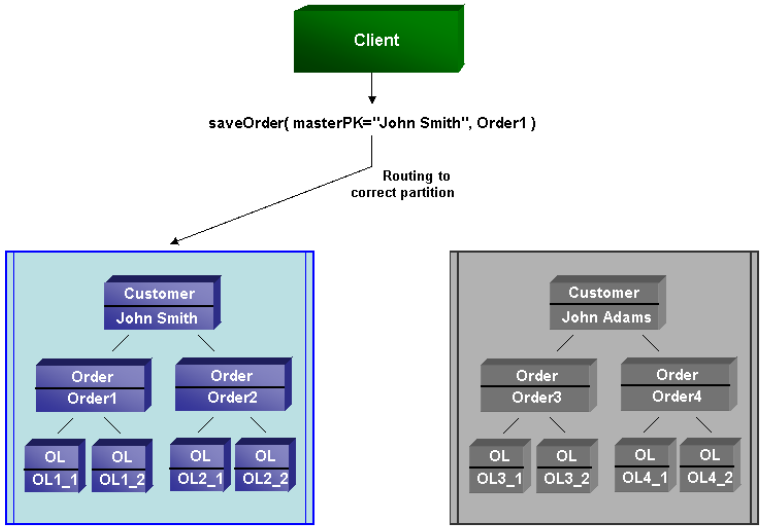
   Note that this applies even in small deployments, where the complete database is not memory resident. Depending on the proportion of reads to writes and the percentage of cache hits on read, this feature can significantly speed up the perceived performance of an application,

2. **Partitioning.** If an application needs access to more information than can fit in the memory of one machine but needs to run at the highest possible speed, then the answer is to place all of the database in memory, spreading tables across nodes and partitioning them where necessary. In this case, relations span node boundaries typically and a write to the in-memory image of the data is effectively a distributed transaction.

Of course, these points are relevant if they *might* be needed at some point in the future. A start-up will want the flexibility to start small (a few machines in the cloud) but be able to scale quickly to an architecture that can handle more transactions or more data. Using CloudSave as the object-store mapper gives this flexibility.

4.1 Entity placement strategy

Therefore, for CloudSave we assume that any entity will reside in a partitioned, backed-up cluster. The entity group gives us the best strategy for distributing changeable entities. (In a future version of CloudSave, we will add services to distribute standing data across a grid.) Other entities, which have no locality of reference with other entities, can be placed in spaces based on available memory capacity: there will be no speed benefit in general.



The diagram shows the strategy of grouping the Customer, Order and OrderLine entities in the same space, with the Customer as the master entity. The partition to hold the information relating to a particular customer is selected based on the Customer primary key; this is also the partition that holds all the Orders and OrderLines for the entity.

The central entity - Customer in this example - is called the **master entity**; the other entities in the group are called **subsidiary entities**. In order to guarantee co-location of the information, the routing of all related entities must be based on the primary key of the master entity. The splitting of entities into entity groups is an architectural decision -i.e. it needs some thought and experience of the domain - and should be based on the expected usage of the information. There is naturally a constraint that there is a required to-one relationship (or series of them) from the subsidiary entity to the master entity, e.g. orderline.getOrder(), then order.getCustomer().

Naturally, it is possible to have relationships from entities in an entity spaces to entities outside it. These accesses are programmed in the same way as for those inside the entity spaces - CloudSave takes care of the details - but the programmer should be aware of the extra effort required to access information.

Pat Helland in his "Life Beyond Distributed Transactions" talks about entities as the scope of atomic transactions. Although he is not explicit, we believe he had the same idea: his "entities" are actually CloudSave's "entity groups" - in general, mapping to many Java objects.

4.2 Time for Presumed Commit

Now the question becomes: how can you reliably persist changes across all these partitioned spaces, in the face of potential failures - machines going down, lost packets and possible timeouts due to JVM garbage collection? This cannot simply be palmed off on the database: in the midst of a distributed transaction, one of the machines involved will eventually go down and the "system of record" will be left in an inconsistent state. This is something that CloudSave must handle, and it is the key difference in functionality between CloudSave and standard mapping solutions like Hibernate.

Helland says "attempts to use distributed transactions are too fragile and perform poorly". Turning that round, the goal of CloudSave is to make distributed transactions robust and fast.

CloudSave uses a "Presumed Commit" approach: in other words, if a transaction is not explicitly aborted, it will commit. This approach is based on three observations:

- the **overwhelming reliability of operations** in memory-based systems. For example, say a machine does 20 transactions per second throughout the working day and over the course of 5 years fails with 20 transactions in flight. Then in that time, around 2 billion operations have succeeded and 20 have failed: so the probability of failure is one in 100 million - so it's a pretty good bet that the hardware will handle the transaction.
- **retries can get out of hand** and cause problems in large systems. The Amazon S3 "Availability Event" is the best documented (and when they say "availability", we know what they mean!). A less well-known example was the meltdown of the NatWest branch systems in 1998, where the systems could not be restarted because of the amount of retrying in the Windows NT servers
- it is impossible for two parts of a distributed system to have the same knowledge simultaneously, because confirmations can get lost. If system A doesn't get told that an operation on system B has succeeded, this can be because system B has failed or because the acknowledgement was lost. Retrying doesn't help - System A still won't know what has happened if there is no response; all that has happened is potentially another connection tied up waiting for a response.

So in CloudSave, we use **presumed commit**: a transaction will be automatically committed if it is not aborted. The assumption that transactions commit means we do not have to "chase the acknowledgement", so the logic is simpler and most operations complete very quickly. Furthermore, this approach means that we avoid the fast positive-feedback loops that can turn a temporary perturbation at heavy load into a meltdown.

The corollary of not chasing the acknowledgement is that CloudSave aborts transactions at the first failure, even if it is retriable - e.g. when a system is busy, causing a timeout on a call into a space. This may seem harsh! However, the alternative of pressing on with retry mechanisms is likely to exacerbate instabilities. To make it easier for the programmer to do some limited retry, we provide a framework for calling transactions at the outermost level that

- Tries once
- In the case of a retriable failure
    - logs it
    - backs off for a random amount of time (e.g. between one and two seconds)
      [The back-off is important to reduce the chances of a positive-feedback loop.]
    - then retries a second time only, using a slightly longer timeout this time.

This is only used at the outermost level - where the business transaction is created. Transaction management in the entity spaces always fails without retry. The sort of retry shown above is worth doing once because we expect that transient timing issues like slow-downs due to JVM garbage collection will be a common source of transaction failure. After a second retry, the likelihood is that the transaction will never commit in a reasonable time, and it makes sense to try other correction mechanism or ask the eventual user if he wants to retry.

4.3 Services

We have not given the prominence to services and business logic that they deserve, simply because the conceptual challenges of CloudSave are around the management of data.

However, if the entity spaces hold the system of record and services have an affinity with an entity group, then the service should naturally reside in the entity spaces. Furthermore, we can expect that, as the functionality of a system grows, applications will increasingly look to the cloud for more services - services in the cloud can become the new point of integration, rather than using an RDBMS as the integration point.

Business services have the following characteristics:

- be the first port of call for the business transaction
- will provide the usual hooks for business activity monitoring and logging of transactions initiation and results for audit purposes - architects will probably want to keep these operations out of client code
- provide enforcement of security policies
- can call other business services and transactional persistence services through the appropriate DAO.

5. Conclusion

This brief overview may have given the impression that solving this problem is easy. If so, let us hurriedly assure you it is not. Ensuring that transactions distributed across large partitioned, backed-up clusters are robust and consistent in the face of failures and reconfiguration requires a lot of careful analysis and testing.

Optimistic programmers tend to think about the happy path - but for distributed transactions, the devil is in the detail of failure scenarios. So there is definitely a case for a product at this layer:

- it is more cost-effective, higher quality, lower risk to use a product than "roll your own"
- separating this layer from the business application will make development easier.

In summary, the key points are:

- distributed transactions. with persistence, in Clouds and Grids can be fast and reliable
- the logic to do this can and should be separated from the business logic
- entity groups are a simple strategy for structuring spaces for best performance
- presumed commit, rather than chasing acknowledgements, is the key to optimal speed and improving stability
- plug-ins can be invoked for different persistence stores, so the solution is not just limited to databases; multiple stores can be accessed
- by using the GigaSpaces infrastructure, the system will scale to arbitrarily large data sets and performance requirements.