JaneySpaces

Table of Contents

Introduction

JaneySpaces is a network abstraction built around shared objects floating in a distributed space. These objects are created, updated and destroyed by clients, who can subscribe to hear messages regarding these space modifications.

I started writing it once I found myself repeating code for sharing state and maintaining network object identity in a number of pet projects. I found that simple message passing or RPC mechanisms still required a lot of repetitive boiler plate code and interfered too much with the design of my applications. JaneySpaces therefore is meant to fit as closely as possible with traditional Object-Oriented ways of development, attempting to be non-intrusive with your plain old Java classes. It encourages decoupling of server and client implementations and simplifies the writing of distributed applications without allowing you to forget that the software runs over a fallible, laggy network. It is generally recognized that over simplified abstractions often leak, making them much harder to use.

JaneySpaces shares many features with JavaSpaces, while borrowing some principles from REST systems. One of these principles is having a limited set of actions or verbs, which can be executed on an infinite number of types of objects, the nouns which float in the space. However it places fewer restrictions on the nature of those objects than JavaSpaces, allowing greater flexibility in your business logic. For example, objects are not automatically removed from the space by clients when they want to make use of them, as they are permanently shared. In this respect JaneySpaces is similar to Terracotta, although the network abstraction is not hidden away in the JVM. Instead you still have to think about how your application is distributed, which can result in more efficient distributed systems which are more resilient to failure on laggy networks. It also avoids having to specify “roots” for your object graphs, since any object can be accessed in the space when needed, and does not require you to provide any object instance field for describing cross network object identity - the local identity of the object, and JaneySpaces network protocol, does this for you. Of course you are still free to create artificial root objects, or give all objects identifying value fields to index them if this suits your design, but it is your choice.

Architecture

The current architecture is the initial version I have written to support the principles outlined above. However, as long as these principles are not broken, the architecture may change in future and developers using JaneySpaces should not notice.

Currently there is a central "Router" server and a number of clients who connect to it. The clients can create/update/destroy objects and subscribe to events by sending messages to the Router, which in turn keeps track of the events each client is interested in and which objects they know of. The Router keeps the state of all objects in the space in memory, so can service all requests from clients without needing assistance from other clients.

The Router is therefore a single point of failure, and potentially a bottleneck on scalability. Please look at the TODO for how this may change in future.

Verbs

As mentioned, JaneySpaces has very few verbs. The main IDistributedClient interface looks roughly like:

```
public interface IDistributedClient {
        <T> IDistributedClient startInstanceListen(T object, IObjectEventHandler<T> listener);
        <T> IDistributedClient stopInstanceListen(T object, IObjectEventHandler<T> listener);
        <T> IDistributedClient startKeyListen(IObjectKey<T> listen, IObjectEventHandler<T>
listener);
        <T> IDistributedClient stopKeyListen(IObjectKey<T> listen, IObjectEventHandler<T>
listener);

        ITransaction startTransaction();
}
```

The first four methods are "local" in that they do not send any messages over the network. They express the types of object events we want to be informed of, where objects can be Created, Updated or Deleted. When a message arrives at the client, the local registered listeners will be called to inform them of relevant changes to objects. More on this later.

Please note that all methods return the IDistributedClient again, allowing simple chaining of method invocations.

The 5th method returns a transaction, which *does* result in network traffic. The ITransaction interface looks something like:

```
        public interface ITransaction {
            void commit();
            ITransaction startKeyListen(IObjectKey<?> listen);
            ITransaction stopKeyListen(IObjectKey<?> listen);
            ITransaction save(Object object);
            ITransaction remove(Object object);
    }
```

The *only* way to modify the distributed JaneySpace is through the last two methods. All objects in the store are inserted by a call to save() followed by a commit(). This is what I mean when I say there are very few verbs in JaneySpaces.

<u>Events</u>

As mentioned, the events in JaneySpaces are Create, Update and Delete. A ITransaction.save() invocation will cause Created or Updated messages, and Itransaction.remove() will cause Delete messages. These are sent to all clients in the space, depending on the routing policy described below.

JaneySpaces separates the routing of events from the local observation of events, which is very different to Terracotta and JavaSpaces, and so far we have only seen the local side of event management. The two listen methods on ITransaction specify which objects we want to be sent over the wire to our client. Note that we pass no listener into this method, since you register the actual listeners on the local level, not remotely. You can therefore specify a remote listen which brings in a wide range of objects, and then specify multiple specialist listeners for different types of object. It is clear that complex distributed caching logic becomes a lot easier to envisage when you have this control over routing, since routing decisions can be decentralized to the cache clients, who can automatically register regions of the "cache space" they are responsible for. Note this means that listening fully for an object requires *two* startKeyListen calls, one to have the objects routed to your client and once to register a listener with those objects. However, the separation of routing from local listener execution makes the system very powerful, and helper classes can be written for common use cases.

IObjectKeys are used to match objects. Types in JaneySpaces can declare "public keys", which will allow them to be indexed for fast lookups by clients and network routers. An IObjectKey<T> represents specific values for the public keys for a particular type T, so by listening for that IObjectKey your client and event handler will be informed of any events (creates, updates and deletes) to objects which match that key. From a routing point of view, once a client becomes informed about an object (as it happens to match a listening-key) the client remains informed of any updates until the object is deleted from the cache (even if it ceases to match the listening-key). From a local client listening point of view the listen methods are only called for keys which match the actual IObjectKey. As IObjectKeys can specify which public keys are "active", nulls are permitted values that you can index by and listen for, avoiding the necessity of creating meaningless "template" objects as in JavaSpaces.

IObjectKeys are also standard objects which JavaSpace recognizes, and must be saved in the space before you can start routing with them. It is therefore possibly to write your own IObjectKey to use in indexing, which in turn a client can listen for to monitor network routing!

Nouns

When you call “save” in a transaction, JaneySpaces does more than add one object to the distributed store. It follows all the references from that object and adds all referants as well. This is necessary otherwise another client on a remote JVM would not see the same object as your have locally as it would lack references to other objects.

JaneySpaces keeps weak references to all objects added to the store locally. Therefore if these objects are garbage collected your local client will automatically remove the objects from the JaneySpace, without you having to call remove(). However, relying on this is not recommended, since the garbage collector on your local machine may not run in time to prevent a remote machine hosting your objects from OOMing. It is therefore recommended you cleanup after yourself.

Note that JaneySpaces does not yet support leasing, like JavaSpaces, for object lifetime management. See the TODO for details.

If you make changes to a local object you will need to call save with that object again for the JaneySpace to find out about your change. However you need not call save on *every* object you modify, since JaneySpaces will follow the references from the object you call save on, and check other objects if they have changed since they were last saved. You may therefore want a “root” node which references many objects you have updated, on which you can call save and have JaneySpaces do the checking for you. See TODO for details.

Also, updates on objects will appear to other clients as exactly that! The local instance of the object will be modified, unlike Java spaces where the identity of the object is detached from the network's identity once you remove it from the space, and saving “updates” causes new objects to be instantiated across the network. In JaneySpaces the identity of objects is managed automatically, while timing distributed updates in state can be set by the developer calling save(). Thread synchronization is therefore very important in business-logic to guarantee that JaneySpace shared objects will not change underneath you while you are using them.

Serialization

JaneySpaces does not use Java Serialization. This is because Java Serialization is designed to send immutable object graphs from A to B, where the only mechanism to send updates is to resend the whole shebang. Since JaneySpaces is distributed over many machines, and only sends updates for objects that have changed, it supports its own serialization mechanisms which use unique id numbers to preserve identity over the network.

These mechanisms are independent of messaging protocol and format. The default format is human readable XML messages, but it would be trivial to support JSON, SOAP, or compact byte formats. The communication protocol is entirely orthogonal to the explanations of how events are propagated and object identity is managed, which are the issues the developer is really concerned about.

Examples

A number of examples are provided in the Test package, including:

1. CleanupTest – saves and removes 100 objects from the store.
2. CollectionsTest – Saves a Arrays.ArrayList in the store to test serialization/deserialization
3. ComplexTypes – Tests serialization of objects with reference fields to (effectively) value types, i.e. Object integer = 5;
4. DistributedListen – A listening and saving clients send messages to each other, through the central router.
5. DistributedUpdates – A client saves objects in the space and modifies then while another client watches.
6. RequestResponseTest – A client sends a message to another client who replies. A simple example of how RPC can be achieved in the space.
7. SimpleSave – Two clients save objects and modify them
8. ThroughputTest – Two clients save 5000 objects each into the space, while another two clients listen. This test is a benchmark for the space of JaneySpaces (i.e. currently not fast enough!)

TODO

JaneySpaces is very new. There are gaps in functionality, undiscovered bugs and serious performance issues which need addressing. Here is a non-exhaustive, non-ordered list:

1. Need more unit tests.
2. Better testing of state changes on objects. Currently we compute hashes each time an object is updated, and compare these to test for updates. This is inefficient since it requires analysing each object, and is ineffective since two object states can have the same hashcode. We need byte code instrumentation to detect updates to objects, so we can have the ease of use of Tangosol but with the update control we already have.
3. Better support for cleaning up objects. If a client dies his objects will stay on the router forever. At the very least we need to purge objects based on ownership. We may also want time-based leases.
4. Allow delegation of routing away from the central router. This will improve performance for larger datasets, as clients can send messages directly to each other, and increase the potential size of the space for a given set of hardware.
5. Have automatic failure over to a new router on the failure of the central one. This should be easy to arrange since the routing metadata can be stored in the JaneySpaces as normal objects.
6. Support for some higher level abstractions on the space:
   a) We need DependencyAdaptors which combine update/delete events from many objects which reference each other into one handler. For example, I would like to add an event

handler to a collection which gets “created” messages when an object is added to the collection (but not when the object is original created). This is possible using instance listeners, and should be separate from the simple IDistributedClient interface.

   b) Asynchronous RPC like interfaces, which insert and listen for request/response objects
   c) “Object binder” which automatically pulls single instances of objects from the store for ease of use.
   d) Data loading api with support for caching.

7. Add support for “ranges” on IObjectKeys, so instead of direct matches we can use the IComparable interface to define a range of acceptable objects. This would be very useful for distributed caching.
8. Add support for the central router to act as a client to another router. I'm thinking about having the IObjectKey routing acting like ip routing, with default gateways and routing tables.
9. Put more thought into synchronization issues – currently a global lock is held during updates on local objects, and when the event handlers are fired. Therefore you can be certain that objects wont be updated beneath you as you operate on them, but this might be unnecessary inefficient, and could potentially deadlock.
10. The allocation of id numbers over the network needs much more thought. Each client currently randomly generates numbers, so clashes are possible. We need a better distributed algorithm.
11. Support for other messaging formats. I am especially interested in a router which only produces Created/Delete messages for immutable objects in JSON format, which would provide excellent AJAX support.
12. Support for propagating changes on fields (like Tangosol) instead of entire objects.
13. Support for “optional references” where the client is not automatically sent an object but can request it if they need it by id number.
14. Support for distributed types in the space which are automatically propagated with object instances. These are in Java byte code and automatically loaded on remote machines, an absolute requirement for many business grid processing scenarios.
15. Support for disk caching on clients/routers of objects which are rarely updated.
16. Write a C# interface! It is possible! Can even use IKVM to support distributed Java byte code loading! Imagine a C# client running on windows, with a nice GUI, talking to a grid of distributed Linux calculation engines running dynamically loaded code in Java!