Distributed Objects aka BigObjects

Distributed objects are called *BigObjects*. They provide a unified view of a data structure over the cluster as the union of all instances of the data structure on cluster nodes. BigObjects also provide *services* as computations that can be performed on one or several cluster nodes where the BigObject is stored.

BigObjects are always distributed among all slave nodes of a cluster, the master node does not store data associated to BigObjects. All cluster nodes have a representation of the BigObject as a container for the data of the BigObject that has to be stored on the node. This part of the data is called the *local data* of the BigObject on a node, and the union of all these local data is the complete data associated with the BigObject on the cluster.

As explained in the document about distribution the distribution of data is governed by objects that are instances of subclasses of `Allocation` and `DHTAllocation`.

BigObject names

BigObjects have an important property: their name. This name is a Java String and there is no rule about the content of this String. The only constraint is that the name must be unique among all the BigObjects, because it is used to identify each BigObject.

Even if there is no rule enforced inside the BigGrph code, there is a kind of convention in use in all the BigGrph programs. This convention advocates the use of a hierarchical naming scheme, where the character `/` is used as a separator, as in the filesystem. In the implementation of some algorithms, the default naming scheme uses this convention: if the BigObject representing a graph is named `graph1`, then the default name of the `LongDHT` (a BigObject) containing the Strongly Connected Components identifiers is `graph1/sccFinalIds`.

On all cluster nodes, there is a registry that keeps track of local instances of BigObjects and store them with their name as index. To retrieve a BigObject from his name, use the following code sample:

```
import bigobject.BigObjectRegistry;

    //
    String name = ... ;
    bo = BigObjectRegistry.defaultRegistry.get(name);
```

BigObject creation: deployment

All BigObjects are subclasses of the abstract class `bigobject.LiveDistributedObject<LocalData, SharedData>`. This class takes two type arguments, only the first one is actually used by BigGrph, in most cases the second type argument is `Serializable`. This allows to write correct code without

The first type argument is the actual class that is used to store data of the BigObject on each slave node of the cluster. For example, the `BigAdjacencyTable` is actually a `LiveDistributedObject<LongObjectMap<LongSet>, Serializable>` which means that on all cluster nodes a map where keys are long numbers and values are sets of long numbers is the representation of the local part of the overall adjacency table of a graph.

Any class of BigObject is a subclass of `LiveDistributedObject`, and its constructor must use the `super()` call with the correct arguments in order to actually call the `LiveDistributedObject` constructor with the required arguments.

The constructor of LiveDistributedObject is defined as

```
LiveDistributedObject(String name, Allocation alloc, Object... extraParameters)
```

This means that two arguments are mandatory: the name of the BigObject and an `Allocation` object which is used to know the cluster nodes on which the BigObject is to be created. All other arguments are optional and depend only on what is necessary for the subclass of `LiveDistributedObject` to properly initialize its instances according to the application requirements.

A subclass of `LiveDistributedObject` will define its constructor so that it will call (through the `super()` call) at the end the base class constructor with the same arguments as the subclass one. The reason is that on the node that initiates the creation of a BigObject, the same constructor that is called at this time will be called with the same arguments on all the other nodes of the cluster.

This also means that all the arguments on a BigObject constructor will be serialized in order to be transfered and used on all the nodes of the cluster.

As an example, suppose that we want to define a type of BigObject that store some information in a list of integer numbers on each cluster node. Such class will be defined with a constructor that takes an initial capacity for the union of all lists.

```
import java.util.ArrayList;
import bigobject.LiveDistributedObject;

public class DistributedLists
       extends LiveDistributedObject<ArrayList<Integer>, Serializable>
{
   public DistributedLists(String name, Allocation allocation, int initialCapacity)
   {
        super(name, Allocation, initialCapacity);
        int localCapacity = initialCapacity / allocation.getNodes.size();
        setLocalData(new ArrayList<Integer>(localCapacity));
   }

   // ...
}
```

When such a class is instanciated (for example on the master node), the constructor of `LiveDistributedObject` is called with the name and the `Allocation` object, plus an `extraParameters` array of size 1 that contains an `Integer` with the value given as the `initialCapacity` argument.

The deployment routine of `LiveDistributedObject` will arrange a computation request on all the other nodes of the cluster to call the constructor of `DistributedLists` with the same set of arguments. Serialization of arguments is done at this point. Thus, on all nodes of the cluster the `DistributedLists` is called with the same arguments. The deployment code takes care to avoid infinite recursion through itself.

It is the responsability of the `DistributedLists` to actually initialize its local list of integers with the relevant size. In this example this done by dividing the overall capacity by the number of cluster nodes.

BigObject services

BigObjects can provide remote computation services to each other. A service is declared to be provided by a BigObject by initializing a field with an instance of the class `bigobject.Service`. Reflection will be used by BigGrph to discover the services provided by a BigObject.

As an example, suppose that we want to provide a service in the previous class `DistributedLists` that returns the actual size of the list of integer stored on each cluster nodes. We define a field that we initialize with an instance of `Service`:

```
import bigobject.Service;
import toools.io.FullDuplexDataConnection2;

   // ...

   final public Service sizeService = new Service() {
       @Override
       void serveThrough(FullDuplexDataConnection2 connection)
           throws ClassNotFoundException, IOException
       {
           connection.out.writeInt(getLocalData().size());
       }
   };
```

The purpose of the `serveThrough()` is to actually handle the request and, if needed, send the response. The `FullDuplexDataConnection2` argument is an object that has both an `in` and `out` fields that are respectively instances of subclasses of `java.io.DataInputStream` and `java.io.DataOutputStream`. These two objects are used to receive a request and to, possibly, send a response.

Invocation of a service need to specify the BigObject instance, the relevant Service in this object and the cluster node on which the request will be executed.

```
import toools.net.TCPConnection;
import octojus.OctojusNode;
import bigobject.Service;

   // ...

   public int getLocalSize(OctojusNode clusterNode)
   {
       TCPConnection connection = connectTo(sizeService, clusterNode)
       int size = connection.in.readInt();
       return size;
   }
```

The `connectTo()` function returns a `TCPConnection` object which has a `in` and `out` field that are Streams to communicate with the other side of the connection. `TCPConnection` is a derived class of `FullDuplexDataConnection2`.

Service requests are processed on each cluster node on a dedicated thread. This means that concurrent invocations of services on the same BigObject are possible and that care should be taken to avoid errors because of concurrency.

BigObject disk serialization

BigObjects can be serialized on disk and deserialized if needed. The basic principles are the following:

- Each cluster node writes on disk its own part of the BigObject (as returned by getLocalData())
- The filenames are derived from the name of the BigObject, including a number that identifies the cluster node.

BigGrph has a global parameter to set the base directory in which such files are created. This parameter, called the *BigObject local directory*, is set as:

- the value of the environment variable `BGRPH_GRAPH_LOCALDIR`;
- the value of the JVM property `biggrph.graph.localdirectory`.

In both cases, this parameter has the same value on all cluster nodes , and the associated directory is the same for all nodes. The default value, if not set through either the environment variable or the JVM property is `~/biggrph/local_data/`.

The serialization/deserialization operations are implementd in the `save()` and `load()` functions of the class `LiveDistributedObject`. The implementation of these functions rely on functions that have a default definition in `LiveDistributedObject` and can be redefined in subclasses of this class.

- `bigobject.LiveDistributedObject.__getLocalFile()` will return the filename into which the local part of the BigObject is to be written to or read from. Default behaviour is to use the BigObject local directory, then to append the name of the BigObject, a path separator, then the number of slave nodes and a path separator, and finally a number from 0 to the the number of slaves minus one as the final filename.
- `bigobject.LiveDistributedObject.__local__saveToDisk()` this function is supposed to write the object returned by `getLocalData()` into the file returned by `__getLocalFile()`.
- `bigobject.LiveDistributedObject.__local_loadFromDisk()` this function restores the object stored in the file returned by `__getLocalFile()` and sets this object as the object returned by `getLocalData()`.

Because the BigObject name is used to build the name of the file, it is sometines interesting to use a hierarchical scheme to build the name of objects and to use the path separator character inside these names.

For example, suppose that the BigObject named `graph1/sccFinalIds` is deployed on a cluster of 4 slave nodes. then calling `save()` on this object will create 4 files: `0.biggrph`, `1.biggrph`, `2.biggrph` and `3.biggrph` all located inside the directory `~/biggrph/local_data/graph1/sccFinalIds/4/`.

Examples of BigObjects

LongDHT, IntDHT and ByteDHT

BigAdjacencyTable