1 Introduction

This manual contains all the necessary information to write and run COMPSs-Redis applications. A COMPSs-Redis application is a program that is written with the COMPSs framework and uses Redis as the storage backend. In this manual **we will assume that COMPSs is already installed** and we will focus on how to install the Redis utilities and the storage API for COMPSs. Also, we will assume that the user understands the underlying programming model of COMPSs and that he or she is able to write applications with it. If some of these requirements are not met the user can refer to the COMPSs manuals[1]. Also, it is advisable to read the Redis Cluster tutorial for beginners[2] in order to understand all the terminology and words that are going to be used here.

2 Installing COMPSs-Redis

In this section we will list all the requirements and dependencies that are necessary to run a COMPS-Redis application.

2.1 Redis Server

`redis-server` is the core Redis program. It allows to create standalone Redis instances that may form part of a cluster in the future. `redis-server` can be obtained by following these steps:

1. Go to `https://redis.io/download` and download the last stable version. This should download a `redis-${version}.tar.gz` file to your computer, where `${version}` is the current latest version.

2. Unpack the compressed file to some directory, open a terminal on it and then type `sudo make install` if you want to install Redis for all users. If you want to have it installed only for yourself you can simply type `make redis-server`. This will leave the `redis-server` executable file inside the directory `src`, allowing you to move it to a more convenient place. By *convenient place* we mean a folder that is in your `PATH` environment variable. It is advisable to not delete the uncompressed folder yet.

3. If you want to be sure that Redis will work well on your machine then you can type `make test`. This will run a very exhaustive test suite on Redis features.

As a reminder, **do not delete the uncompressed folder yet**.

[1] https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar/documentation

[2] https://redis.io/topics/cluster-tutorial

2.2 Redis Cluster Script

Redis needs an additional script to form a cluster from various Redis instances. This script is called `redis-trib.rb` and can be found in the same tar.gz file that contains the sources to compile `redis-server` in `src/redis-trib.rb`. Two things must be done to make this script work:

1. Move it to a convenient folder. By *convenient folder* we mean a folder that is in your `PATH` environment variable.
2. Make sure that you have Ruby and `gem` installed. Type `gem install redis`.
3. If you want to use COMPSs-Redis with Python you must also install the PyPI packages `redis` and `redis-py-cluster`. It is also advisable to have the PyPI package `hiredis`, which is a library that makes the interactions with the storage to go faster.

3 COMPSs-Redis Bundle

`COMPSs-Redis Bundle` is a software package that contains the following:

1. A java JAR file named `compss-redisPSCO.jar`. This JAR contains the implementation of a Storage Object that interacts with a given Redis backend. We will discuss the details later.
2. A folder named `scripts`. This folder contains a bunch of scripts that allows a COMPSs-Redis app to create a custom, in-place cluster for the application.
3. A folder named `python` that contains the Python equivalent to `compss-redisPSCO.jar`

This package can be obtained as follows:

1. Go to `trunk/utils/storage/redisPSCO`
2. Type `./make_bundle`. This will leave a folder named `COMPSs-Redis-bundle` with all the bundle contents.

4 Writing COMPSs-Redis applications

4.1 Java

This section describes how to develop Java applications with the Redis-COMPSs storage. The application project should have the dependency induced by `compss-redisPSCO.jar` satisfied. That is, it should be included in you `pom.xml` if you are using Maven, or it should be listed in the dependencies section of the used development tool.

A COMPSs-Redis application is almost identical to a regular COMPSs application except for the presence of Storage Objects. A Storage Object is an object that it is capable to interact with the storage backend. If a custom object extends the Redis Storage Object and implements the Serializable interface then it will be ready to be stored and retrieved from a Redis database. An example signature could be the following:

```
import storage.StorageObject;
import java.io.Serializable;

  /**
  * A PSCO that contains a KD point
  */
  class RedisPoint
  extends StorageObject implements Serializable {

    // Coordinates of our point
    private double[] coordinates;
    /**
    * Write here your class-specific
    * constructors, attributes and methods.
    */
    double getManhattanDistance(RedisPoint other) {
      ...
    }
  }
```

The `StorageObject` object has some inherited methods that allow the user to write custom objects that interact with the Redis backend. These methods can be found in table 1.

Name	Returns	Comments
`makePersistent(String id)`	Nothing	Inserts the object in the database with the id. If id is null, a random UUID will be computed instead.
`deletePersistent()`	Nothing	Removes the object from the storage. It does nothing if it wasn't already there.
`getID()`	String	Returns the current object identifier. If the object is not persisted, returns null instead.

Table 1: Available methods from StorageObject

As an important observation, **Redis Storage Objects that are used as INOUTs must be manually updated**. This is due to the fact that COMPSs does not know the exact effects of the interaction between the object and the storage, so the runtime cannot know if it is necessary to call `makePersistent` after having used an INOUT or not (other storage approaches do live modifications to its storage objects). The following example illustrates this situation:

```
/**
* A is passed as INOUT
*/
void accumulativePointSum(RedisPoint a, RedisPoint b) {
  // This method computes the coordinate-wise sum between a and b
  // and leaves the result in a
  for(int i=0; i<a.getCoordinates().length; ++i) {
    a.setComponent(i, a.getComponent(i) + b.getComponent(i));
  }
  // Delete the object from the storage and
  // re-insert the object with the same old identifier
  String objectIdentifier = a.getID();
  // Redis contains the old version of the object
  a.deletePersistent();
  // Now we will insert the updated one
  a.makePersistent(objectIdentifier);
}
```

If the last three statements were not present, the changes would never be reflected on the `RedisPoint a` object.

4.2 Python

COMPSs-Redis is also available for Python. As happens with Java, we first need to define a custom Storage Object. Let's suppose that we want to write an application that multiplies two matrices A, and B by blocks. We can define a `Block` object that lets us store and write matrix blocks in our Redis backend:

```
from storage.storage_object import StorageObject
import storage.api

class Block(StorageObject):
  def __init__(self, block):
      super(Block, self).__init__()
      self.block = block

  def get_block(self):
      return self.block

  def set_block(self, new_block):
      self.block = new_block
```

Let's suppose that we are multiplying our matrices in the usual blocked way:

```
for i in range(MSIZE):
  for j in range(MSIZE):
    for k in range(MSIZE):
      multiply(A[i][k], B[k][j], C[i][j])
```

Where A and B are `Block` objects and C is a regular Python object (e.g: a Numpy matrix), then we can define `multiply` as a task as follows:

```
@task(c = INOUT)
def multiply(a_object, b_object, c, MKLProc):
    c += a_object.block * b_object.block
```

Let's also suppose that we are interested to store the final result in our storage. A possible solution is the following:

```
for i in range(MSIZE):
    for j in range(MSIZE):
        persist_result(C[i][j])
```

Where `persist_result` can be defined as a task as follows:

```
@task()
def persist_result(obj):
    to_persist = Block(obj)
    to_persist.make_persistent()
```

This way is preferred for two big reasons: we avoid to bring the resulting matrix to the master node, and we can exploit the data locality by executing the task in the node where last version of `obj` is located.

5 Launching COMPSs-Redis and using an existing Redis Cluster

If there is already a running Redis Cluster on the node/s where the COMPSs application will run then only the following steps must be followed:

1. Create a `storage_conf.cfg` file that lists, one per line, the nodes where the storage is present. Only hostnames or IPs are needed, ports are not necessary here.
2. Add the flag `--classpath=${path_to_COMPSs-redisPSCO.jar}` to the `runcompss` command that launches the application.
3. Add the flag `--storage_conf=${path_to_your_storage_conf_dot_cfg_file}` to the `runcompss` command that launches the application.
4. If you are running a python app, also add the `--pythonpath=${app_path}:${path_to_the_bundle_folder}/python` flag to the `runcompss` command that launches the application.

As usual, the `project.xml` and `resources.xml` files must be correctly set. It must be noted that there can be Redis nodes that are not COMPSs nodes (although this is a highly unrecommended practice). As a requirement, **there must be at least one Redis instance on each COMPSs node listening**

to the official Redis port 6379[3]. This is required because nodes without running Redis instances would cause a great amount of transfers (they will **always** need data that must be transferred from another node). Also, any locality policy will likely cause this node to have a very low workload, rendering it almost useless.

6 Integrating COMPSs-Redis on queue system based environments

`COMPSs-Redis-Bundle` also includes a collection of scripts that allow the user to create an in-place Redis cluster with his/her COMPSs application. These scripts will create a cluster using only the COMPSs nodes. Some parameters can be tuned by the user via a `storage_props.cfg` file. This file must have the following form:

```
REDIS_HOME=some_path
REDIS_NODE_TIMEOUT=some_nonnegative_integer_value
REDIS_REPLICAS=some_nonnegative_integer_value
```

There are some observations regarding to this configuration file:

1. `REDIS_HOME` must be equal to a path to some location that is **not** shared between nodes. This is the location where the Redis sandboxes for the instances will be created.
2. `REDIS_NODE_TIMEOUT` must be a nonnegative integer number that represents the amount of milliseconds that must pass before Redis declares the cluster broken in the case that some instance is not available.
3. `REDIS_REPLICAS` must be equal to a nonnegative integer. This value will represent the amount of replicas that a given shard will have. If possible, Redis will ensure that all replicas of a given shard will be on different nodes.

In order to run a COMPSs-Redis application on a queue system the user must add the following flags to his or her `enqueue_compss` command:

1. `--storage-home=${path_to_the_bundle_folder}` This must point to the root of the COMPSs-Redis bundle.
2. `--storage-props=${path_to_the_storage_props_file}` This must point to the `storage_props.cfg` mentioned above.
3. `--classpath=${path_to_COMPSs-redisPSCO.jar}` As in the previous section, the JAR with the storage API must be specified.

[3] https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

4. If you are running a Python application, also add the `--pythonpath=${app_path}:${path_to_the_bundle_folder}` flag

As a requirement, the supercomputer must not have any kind of zombie-killer mechanisms. That is, the system should not kill daemonized processes running on the given computing nodes.