dataClay

Manual

Barcelona Supercomputing Center

Contents

Getting started

1. Main Concepts

1.1 What is dataClay

dataClay [1, 2] is a next-generation object store that enables programmers to store objects using the same model they use in the application, thus avoiding time consuming transformation between persistent and non persistent models. In other words, dataClay enables applications to store objects in the same format they have in memory, just calling a makePersistent method, and thus accessing to persistent object or object in memory is the same (you just follow the object reference).
In addition, dataClay simplifies and optimizes the idea moving computation close the data (see Section 1.3) by enabling the execution of methods in the same node where a given object is located. dataClay also optimized the idea of sharing data and models (set of classes) between different users by means of storing the class (including method definition) together with the object.

1.2 Basic terminology

In this section we present a brief terminology that is used throughout the manual.

dataClay application is any application that uses dataClay to handle its persistent data.
Backend is a node in the system that is able to store persistent objects and handle execution requests. These nodes need to be running the dataClay platform on them. We can have as many as we need either for capacity or parallelism reasons.
Clients are the machines where dataClay applications run. These nodes can be very thin. They only need to be able to run Java or Python code and to have the dataClay lib installed.
dataClay object is any object stored in dataClay.
Objects with alias are objects that have been explicitly named by the owner (much in the same way we give names to files). Not all dataClay objects have an alias (a name). If an object has an alias, we can access it by using its name. On the other hand, objects without an alias can only be accessed by a reference from another object.
Dataset is an abstraction where many objects are grouped. It is indented to simplify the task of sharing objects with other users.
Data model consists of a set of related classes.

Namespace is a dataClay abstraction aimed at grouping a set of classes together. Namespaces have two objectives: i) grouping related classes to ease the task of sharing them with other users and ii) avoiding clashing of class names. A namespace is similar to a package in Java or module in Python.

1.3 Execution model

As we have mentioned, one of the key features of dataClay is to offer a mechanism to bring computation closer to data. For this reason, all methods of an object stored in dataClay, will not be executed in the client (application address space) but on the backend where dataClay stored the object. Thus, searching for an object in a collection will not imply sending all objects in the collection to the client, but only the final result because the search method will be executed in the backend.

1.4 Tasks and roles

In order to rationalize the different roles that take part in data-centric applications, such as the ones supported by dataClay, we assume three different roles.

Data provider is the owner of the data.

Model provider is an expert in data modeling that builds models matching data provided by the data providers. These models can integrate data from different providers or provide better structure or management methods than the ones offered initially by the data providers.

Application developers are the traditional application developers. The idea is that when developing a dataClay application, you use one of the optimized models developed by a model provider.

Although dataClay encourages these roles in the cycle of applications, they do not have to be declared as such and, of course, they can be assumed by a single person.

1.5 Garbage collection

Given that dataClay stores language objects (currently Java and Python) in a distributed manner, dataClay also implements a global garbage collection to guarantee that objects that cannot be referenced any longer get removed and do not waste persistent space. The rules to decide when an object is not reachable are the same as in the original languages with a small addition: objects with name (alias) are also placed into the set of root objects.

2. My first dataClay application

In order to better understand what dataClay is and how it is used, we present a very simple example (HelloPeople) where data is stored using dataClay. Sections 2.1.1 and 2.1.2 present this example both in Java and Python respectively.

2.1 HelloPeople: a first dataClay example

HelloPeople is a simple application that stores a set of people into a persistent collection. Every time this application is executed, it first tries to load the collection identified by the given name, and if such collection does not exist the application creates it. Once the collections has been retrieved, or created, the new person is added to the collection and the whole set of people is displayed. HelloPeople receives the following parameters:

- a string that identifies the name of the collection.
- a string with the name of the person to be inserted into the collection.
- an integer with the age of the person to be inserted into the collection.

2.1.1 Java

Example: HelloPeople, my first dataClay applicataion in Java

Person.java

```
package model;

public class Person {
  String name;
  int age;

  public Person(String newName, int newAge) throws Exception {
    name = newName;
```

```
    age = newAge;
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }
}
```

People.java

```
package model;

import java.util.ArrayList;

public class People {
   private ArrayList<Person> people;

   public People() {
      people = new ArrayList<>();
   }

   public void add(final Person newPerson) {
      people.add(newPerson);
   }

   public String toString() {
      String result = "People: \n";
      for (Person p : people) {
         result += " - Name: " + p.getName();
         result += " Age: " + p.getAge() + "\n";
      }
      return result;
   }
}
```

HelloPeople.java

```
package application;

import api.DataClay;
import java.util.Iterator;
import model.People;
import model.Person;

public class HelloPeople {
  private static void usage() {
    System.out.println("Usage: application.HelloPeople <peopleAlias>
        <personName> <personAge>");
    System.exit(-1);
```

```
  }

  public static void main(String[] args) throws Exception {
    // Check and parse arguments
    if (args.length != 3) {
      usage();
    }
    String peopleAlias = args[0];
    String pName = args[1];
    int pAge = Integer.parseInt(args[2]);

    // Init dataClay session
    DataClay.init();

    // Access (or create people object with alias)
    People people = null;
    try {
      people = People.getByAlias(peopleAlias);
      System.out.println("[LOG] Collection " + peopleAlias + " found.");
    } catch (Exception ex) {
      System.out.println("[LOG] Creating NEW peopleAlias");
      people = new People();
      people.makePersistent(peopleAlias);
    }

    // Create person
    Person person = new Person(pName, pAge);
    person.makePersistent();
    people.add(person);

    // Iterate through people remotely
    System.out.println(people);

    // Finish dataClay session
    DataClay.finish();
  }
}
```

2.1.2 Python

Example: HelloPeople, my first dataClay applicataion in Python

person.py

```
from dataclay import DataClayObject, dclayMethod

class Person(DataClayObject):
    """
    @ClassField name str
    @ClassField age int
    """
    @dclayMethod(name='str', age='int')
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age
```

```
from dataclay import DataClayObject, dclayMethod

class People(DataClayObject):
    """
    @ClassField people list<model.classes.Person>
    """
    @dclayMethod()
    def __init__(self):
        self.people = list()

    @dclayMethod(new_person="model.classes.Person")
    def add(self, new_person):
        self.people.append(new_person)

    @dclayMethod(return_="str")
    def __str__(self):
        result = ["People:"]

        for p in self.people:
            result.append(" - Name: %s " % p.name)
            result.append(" - Age: %d " % p.age)

        return "\n".join(result)
```

hellopeople.py

```
#!/usr/bin/env python2
import sys

from dataclay.api import init, finish

# Init dataClay session
init()

from model.classes import Person, People


class Attributes(object):
    pass


def usage():
    print "Usage: hellopeople.py <colname> <personName> <personAge>"


def init_attributes(attributes):
    if len(sys.argv) != 4:
        print "ERROR: Missing parameters"
        usage()
```

```
        exit(2)
    attributes.collection = sys.argv[1]
    attributes.p_name = sys.argv[2]
    attributes.p_age = int(sys.argv[3])


if __name__ == "__main__":
    attributes = Attributes()
    init_attributes(attributes)

    # Create people object if it does not exist
    try:
                # Trying to retrieve it using alias
        people = People.get_by_alias(attributes.collection)
    except Exception:
        print "[LOG] Creating people's object with alias %s" \
            % attributes.collection
        people = People()
        people.make_persistent(alias=attributes.collection)

        # Add new person to people
    person = Person(attributes.p_name, attributes.p_age)
    person.make_persistent()
    people.add(person)

    print "[LOG] People object contains:"
    print people

        # Close session
    finish()
```

3. Application cycle

Now that we have created our first application (HelloPeople) by defining the class model (person.java) and main program (HelloPeople.java), we can detail the steps that need to be done in order for this application to run using dataClay to store its data in a persistent state. A graphical view of these steps is presented in Figure 3.1 and they are detailed in the following sections.

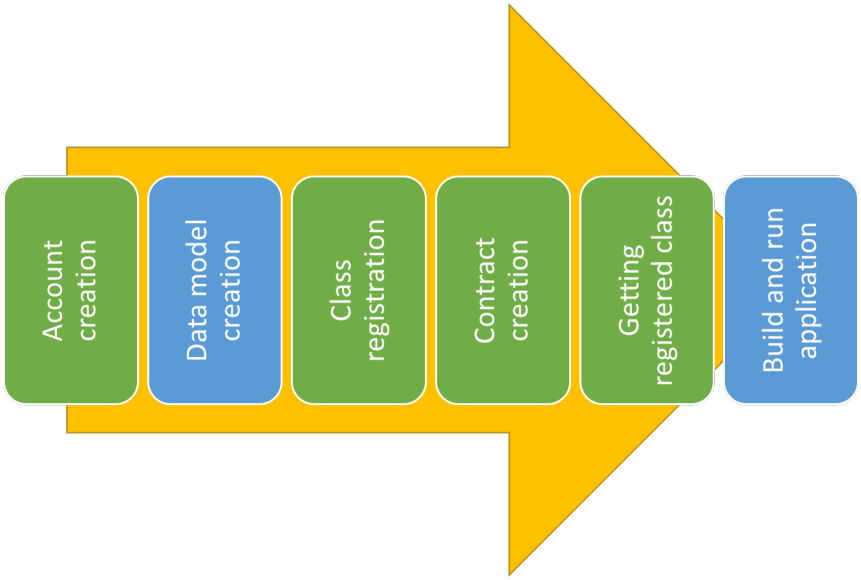


Figure 3.1: Application life cycle

3.1 Account creation

Everybody using dataClay needs to have its own account regardless of the role they play (data providers, model provider, or application programmer). Currently accounts are identified by a

string and are protected by a password. Accounts are the abstraction used to grant/deny privileges to create/use/access models and/or data.
Although dataClay foresees three different roles with respect to data, they can all be assumed by the same person and this is that case in the HelloPeople example: a single person (you) creates the model, creates application, and inserts the data.
Details on how accounts are created are presented in Chapter 6.

3.2 Namespaces and class models

The model provider is in charge of implementing the Java/Python set of classes that define a specific class model. These classes are designed and implemented like always, ignoring that these classes will then be used to store persistent objects in dataClay. Once the classes have been created and tested, the model provider needs to register them into dataClay to enable objects of these classes to be stored persistently.
Registering classes is important for three reasons: i) enables dataClay to automatically offer an optimal serialization of the objects instantiating any of these classes, ii) enables dataClay to execute class methods over the objects inside the dataClay infrastructure without having to move the data from dataClay to the application, and iii) enables the sharing of classes among application developers in an easy and effective way.
Registering classes only implies, from the model provider point of view, uploading their files (e.g. .class files or .py files) into dataClay and defining into which namespace they should be added.
A **namespace** is a dataClay abstraction to group a set of classes together. Namespaces have to objectives: i) grouping related classes to ease the task of sharing them with other users and ii) avoiding class name clashing. A namespace is similar to a package in Java or module in Python.
Registering a class model can be easily performed by using the available tools described in Chapter 6 and has to be performed before any application tries to store objects of this class model into dataClay.

3.3 Datasets and data contracts

In the same way we grouped classes into namespaces to ease the task of sharing them, dataClay also has the dataset abstraction. A **dataset** is a set of objects that will be shared with other users as a whole. Objects inside a dataset can be of any class and there is no restriction to the number of objects or their size inside a datasets. Datasets are identified by a string defined by the creator of the dataset.
In order to be able to use a dataset, its owner has to explicitly grant permission to another user to access it. This permission granting is acheived by means of data contracts. Contract creation can be easily performed by using the available tools described in Chapter 6. Once a data contract is created, dataClay will make sure that the user receiving the contract will have access to the objects included in the datasets of the contract. An application can use as many data contracts as needed.

3.4 Using a registered class: getting its stubs

As we mentioned in Section 3.2, when we implement a class model we do not take into account anything about persistence, but when using it (as seen in the code in Section 2.1), persistent objects have methods such as `makePersistent` that have not been defined as part of the class. In order to be able to use such methods, and thus enable persistence of objects, the application needs to be linked with an automatically modified version of the classes. This modified version is what we refer as class **stubs** in dataClay. A stub is a class file containing the modified version of the original

class in order to be compatible with dataClay. It is important to understand that, besides the newly added methods, the rest of class behaves just like the original one.
The stubs of a class are obtained using the available tools described in Chapter 6.

3.5 Build and run the application

To run a dataClay application, we just need i) to make sure that it is using the stubs instead of the original class files and ii) to create a configuration file that specifies account information, datasets, etc. The details about this file are described in Section 7.1

3.6 Easier than it looks

Let's see how we can execute the examples in sections 2.1.1 and 2.1.2. First, we define a configuration file containing the basic information to open a client session with dataClay. By naming this file as *session.properties*, *DataClay.init()* automatically processes it (more details in Section 4.1). An example of *config.properties* is presented here:

```
Account=Alice
Password=AlicePass
DataSets=HelloPeopleDS
DataSetForStore=HelloPeopleDS
StubsClasspath=./stubs
DataClayClientConfig=./cfgfiles/client.properties
```

File named *client.properties* which path is specified in variable *DataClayClientConfig*, contains the basic information for the initial network connection with dataClay:

```
HOST=127.0.0.1
PORT=11034
```

Now we can start using the dataClay tool for management operations. In this way, our class models can be registered, datasets with specific access rights can be defined, and our application can interact transparently with dataClay when using downloaded stubs (more details about the tool in Chapter 6.

```
// To begin with, create an account
dClayTool.sh NewAccount Alice AlicePass

// Create a dataset (with granted access) to register stored objects on it
dClayTool.sh NewDataContract Alice AlicePass myDataset

// Register the class model in a certain namespace
// Assuming Person.class or person.py is in ./modelClassDirPath
dClayTool.sh NewModel Alice AlicePass myNamespace ./modelClassDirPath <java |
    python>

// Download the corresponding stubs for your application
dClayTool.sh GetStubs Alice AlicePass myNamespace stubsDirPath
```

II Java: Programmer API

4. Java API

In this chapter we will present the Java API that can be used by applications. The description of the available methods has been divided into the following sections. First, in Section 4.1 we present the API that is not linked to objects directly, but is intended to initialize and finish applications, as well as, to gather information about the system. Secondly, in Section 4.3 we introduce the most basic extensions to the class methods that will be used by all dataClay applications to continue in Section 4.4 with more advanced extensions that will only be needed by a subset of applications. Finally, exception handling is also introduced.

4.1 Global API

In this section, we present a set of calls that are not linked to any given object, but are general to the system. In Java, they can be called through *api.DataClay* class.

```
public static void  finish () throws DataClayException
```

Description:
Finishes a session with dataClay that has been previously created using init.
Exceptions:
If any error occurs while finishing the session, a DataClayException is thrown.

```
public static Map<BackendID, Backend>  getBackends ()
```

Description:
Retrieves the available backends in the system.
Returns:
A hash map with the available backends in the system indexed by their backend IDs.

```
public static void  init () throws DataClayException
```

Description:
Creates and initializes a new session with dataClay.
Environment:
session.properties: The configuration file. Location of this file and its contents are detailed in Section 7.1.
Exceptions:
If any error occurs while initializing the session, a DataClayException is thrown.

Example: Using global api - distributed people

```
// Open session with init()
DataClay.init();

List<Person> people = new ArrayList<>();
people.add(new Person("Alice", 32));
people.add(new Person("Bob", 41));
people.add(new Person("Charlie", 35));

// Retrieve backend information with getBackends()
Map<BackendID, Backend> backends = DataClay.getBackends();

BackendID[] backendArray = backends.keySet().toArray();
int numBackends = backends.size();
for (int i = 0; i < people.size(); i++) {
  people.get(i).makePersistent(backendArray[i % numBackends]);
}

// Close session with finish()
DataClay.finish();
```

4.2 Static stub class methods

These methods can be called from any downloaded stub as static or class methods. In this way, the user is allowed to reference persistent objects by using their aliases (see Section 4.3 to see how objects are persisted with an alias assigned). Aliases prevent objects to be removed by the Garbage Collector, thus an operation to remove the alias of an object is also provided. More details on how the garbage collector works can be found in Section 1.5. Notice that the examples provided assume the initialization and finalization of user's session with methods described in previous section Section 4.1.

```
public static void  deleteAlias  (String alias)
throws DataClayException
```

Description:
Removes the alias linked to an object. If this object is not referenced starting from a root object, the garbage collector will remove it from the system.
Parameters:
alias: alias to be removed.

Exceptions:
If no object with specified alias exists, a DataClayException is raised.

Example: Using deleteAlias

```
Person newPerson = new Person("Alice", 32);
newPerson.makePersistent("student1");
...
Person.deleteAlias("student1");
```

```
public static void <T>  getByAlias  (String alias)
throws DataClayException
```

Description:
Retrieves an object instance of the current stub class (from the method is being called), that references the corresponding persistent object with the alias provided.
Parameters:
alias: alias of the object.
Exceptions:
If no object with specified alias exists, a DataClayException is raised.

Example: Using getByAlias

```
Person newPerson = new Person("Alice", 32);
newPerson.makePersistent("student1");
Person refPerson = Person.getByAlias("student1");
assertTrue(newPerson.getName().equals(refPerson.getName())
```

4.3 Object extended methods: basic

In this section we present the basic extension to the object methods. This methods are intended to be used by standard programmers to handle the persistence of their objects.

```
public void  makePersistent  () throws DataClayException
public void  makePersistent  (BackendID backendID)
throws DataClayException
public void  makePersistent  (String alias, [BackendID backendID])
throws DataClayException
public void  makePersistent  (String alias, [boolean recursive])
throws DataClayException
public void  makePersistent  (BackendID backendID, [boolean recursive])
throws DataClayException
public void  makePersistent  (String alias, BackendID backendID,
[boolean recursive]) throws DataClayException
```

Description:

Stores an object in dataClay and assigns an OID to it.

Parameters:

alias: a string that will identify the object in addition to its OID. Alias need to be unique per class. If no alias is set, then this object will not have an alias, will only be accessible though other object references.

backendID: identifies the backend where the object will be stored. If this parameter is missing, then a random backend is selected to store the object. **LOCAL** can be set as a constant for backendID. When LOCAL is used, the object is created in the backend specified as local in the client configuration file (as detailed in Section 7.1).

recursive: when this flag is TRUE, all objects referenced by the current one will also be made persistent (in case they were not already persistent) in a recursive manner. When this parameter is not set, the default behavior is to perform a recursive makePersistent.

Exceptions:

If the alias already exists or the BackednID is not a valid one, a DataClayException is raised.

Example: Using makePersistent method

```
Person newPerson("Alice", 32);
newPerson.makePersistent("person1", DataClay.LOCAL);
assertTrue(newPerson.getLocation().equals(DataClay.LOCAL));
```

4.4 Object extended methods: advanced

In this section we present the advanced extension to the object methods. This methods are not intended to be used by standard programmers, but by runtime and library developers or expert programmers.

```
public BackendID  getLocation () throws DataClayException
```

Description:

Retrieves the location of the object.

Returns:

Backend ID in which this object is stored. If the object is not persistent (i.e. it has never been persisted) this function will fail.

Exceptions:

If the object is not persistent, a DataClayException is raised.

Example: Using getLocation

```
Person p1 = Person.getByAlias("person1");
p1.makePersistent(DataClay.LOCAL);
assertTrue(p1.getLocation().equals(DataClay.LOCAL));
```

```
public BackendID  newReplica () throws DataClayException
public BackendID  newReplica (boolean recursive) throws DataClayException
```

```
public BackendID  newReplica (BackendID backendID
[,boolean recursive]) throws DataClayException
```

Description:
Creates a replica of the current object.

Parameters:
backendID: ID of the backend in which to create the replica. If null, a random backend is chosen.
recursive: when this flag is TRUE, all objects referenced by the current one will also be replicated (except those that are already present in the destination backend). When this parameter is not set, the default behavior is to perform a recursive replica.

Returns:
The ID of the backend in which the replica was created. It is very important to realize that dataClay does not take care of replica synchronization. Details on how such synchronization can be achieved are described in Section 4.6

Exceptions:
If the object is not persistent or the location ID do not represent a valid location, a DataClayException is raised.

Example: Using newReplica

```
Person p1 = Person.getByAlias("person1");
// replicating object and subobjects, from one of its locations to LOCAL
p1.newReplica(p1.getLocation(), DataClay.LOCAL);
```

```
public void  federate (String dataClayName
[,boolean recursive]) throws DataClayException
```

Description:
Federates current object with external dataClay.

Parameters:
dataClayName: Name of the external dataClay. It must be previously registered.
recursive: when this flag is TRUE, all objects referenced by the current one will also be federated (except those that are already present in the destination dataClay). When this parameter is not set, the default behavior is to perform a recursive federation.

Exceptions:
If the object is not persistent or the external dataClay is not registered, a DataClayException is raised.

Example: Using federate

```
Person p1 = Person.getByAlias("person1");
// federating object and subobjects from our dataClay service to dataClay2
p1.federate("dataClay2");
```

4.5 Error management

DataClayException is the dataClay exception class that can be thrown from Java object operations. Currently it extends directly from java.lang.Exception providing the same API.

4.6 Replica management

In this section we present how to manage consistency for objects that instantiate Java classes and are replicated among different backends.
Let us suppose that we have our class Person:

```
public class Person {
  String name;
  int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Once this class is registered and with the proper permissions and stubs, an application that uses it might look like this:

```
public class App {
  public static void main(String[] args) {
    DataClay.init();

    Person p = new Person("Alice", 42);
    p.makePersistent("person1");

    p.newReplica();

    p.setAge(43);

    System.out.println(p.getAge());
  }
}
```

With no consistency policies, the printed message would show an unpredictable age for Alice, since methods *setAge* and *getAge* are executed in a random backend among the locations of the object. In order to overcome this problem, dataClay provides a mechanism to define synchronization policies at user-level. In particular, class developers are allowed to define three different annotations to customize the behavior of attribute updates:

@DataClayAnnotations.InMaster
@DataClayAnnotations.BeforeUpdate(method="...", clazz="...")
@DataClayAnnotations.AfterUpdate(method="...", clazz="...")

The *InMaster* annotation forces the update operation to be handled from the master location.
On the other hand, *BeforeUpdate* and *AfterUpdate* define extra behavior to be executed before or after the update operation. The *method* argument specifies an operation signature of a static class method. The *clazz* argument refers to the class where such a static class method is implemented. In this way, the developer is allowed to define an action to be triggered before the update operation,

and an action to be taken after the update operation.

Let us resume our previous example. Assuming that the *name* attribute is never modified (e.g. private setter), we want, however, that every time the age is updated the change is propagated to all the replicas. Empowering Person class with the proper annotation, we can intervene updates of attribute *age* to perform the update synchronization:

```
public class Person {
  String name;

  @Inmaster
  @DataClayAnnotations.AfterUpdate(method="replicateToSlaves",
                                   clazz="model.SequentialConsistency")
  int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Following the example, an as part of the class model, the proposed *SequentialConsistency* class could be implemented as follows:

```
package model;

import java.util.Set;

import api.BackendID;
import serialization.DataClayObject;

public class SequentialConsistency {
  public static void replicateToSlaves(DataClayObject o, String setter,
      Object[] args) {
    Set<BackendID> locations = o.getAllLocations();
    locations.remove(o.getMasterLocation()); // synchronize with non-master
        replicas
    for (BackendID replicaLocation : locations) {
      o.runRemote(replicaLocation, setter, args);
    }
  }
}
```

In this example, the master replica leads a sequential consistency model by synchronizing the contents with secondary replicas.

Few considerations merit the attention of model developers:

The master location of an object is the backend where the object was originally created. It can be checked with the method *getMasterLocation()*.

The method name specified in the annotations is always implemented as a *public static void* operation, which receives the context info about the original method that triggered the action. This context info consists of:

- A dataClay object reference. Object in which the original method is being executed. In our example, a reference to Person object.

- The method itself. An identifier that dataClay can manage. In our example, the *setAge* method identifier.
- The arguments received by the method. In our example, the new age to be set.

4.7 Object federation

In this section we present how to manage federation of objects that instantiate Java classes.
It is assumed a scenario with several dataClays running at the same time. The process of discovery to make a dataClay to be aware of other dataClays can be achieved by using dClayTool as exposed in section 6.5.
Notice that a current requirement for federating an object is that the class model of the object is registered in the involved dataClays within the same Namespace.
Let us suppose that we have our class Person:

```
public class Person {
  String name;
  int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Once this class is registered in both dataClays in the same Namespace, and with the proper permissions and stubs, an application that uses it might look like this:

```
public class App {
  public static void main(String[] args) {
    DataClay.init();

    Person p = new Person("Alice", 42);
    p.makePersistent("person1");

    p.federate("dataClay2");
  }
}
```

At this point, an application in a secondary dataClay named *dataClay2* can execute the following code:

```
public class App {
  public static void main(String[] args) {
    DataClay.init();

    Person p = Person.getByAlias("person1");

    System.out.println(p.name);
  }
}
```

The secondary dataClay has actually performed a replica of Person object aliased *person1*. From

now on, this replica can be used in the execution environment of any of the backends of the secondary dataClay.

III Python: Programmer API

5. Python API

In this chapter we will present the Python API that can be used by applications. The description of the available methods has been divided into the following sections. First, in Section 5.1 we present the API that is not linked to objects directly, but is intended to initialize and finish applications, as well as, to gather information about the system. Secondly, in Section 5.3 we introduce the most basic extensions to the class methods that will be used by all dataClay applications to continue in Section 5.4 with more advanced extensions that will only be needed by a subset of applications. Finally, exception handling is also introduced.

5.1 Global API

In this section, we present a set of calls that are not linked to any given object, but are general to the system. In Python, they can be called through *dataclay.api* or *dataclay.runtime*.

```
def finish ():
```

Description:
Finishes a session with dataClay that has been previously created using init.
Exceptions:
If any error occurs while finishing the session, a DataClayException is thrown.

```
def get_backends ():
```

Description:
Retrieves the available backends in the system.
Returns:
A map with the available backends in the system indexed by their IDs.

```
def  init (config_file='./cfgfiles/session.properties'):
```

Description:
Creates and initializes a new session with dataClay.
Environment:
session.properties: The configuration file can be optionally specified. Location of this file and its contents are detailed in Section 7.1.
Exceptions:
If any error occurs while initializing the session, a DataClayException is thrown.

Example: Using global api - distributed people

```
from dataclay.api import finish, init, get_backends

# Open session with init()
init()

from model import Person

person1 = Person(name='Alice', age=32)
person2 = Person(name='Bob', age=41)
person2 = Person(name='Charlie', age=35)
people = [p1, p2, p3]

# Retrieve backend information with getBackends()
backends = list(get_backends().keys())

# Round robin of persons in backends
numbackends = len(backends)
for i in range (len(people)):
  people[i].make_persistent(backend_id=backends[i % numbackends])

# Close session with finish()
finish()
```

5.2 Stub class methods

These methods can be called from any downloaded stub as class methods. In this way, the user is allowed to reference persistent objects by using their aliases (see Section 5.3 to see how objects are persisted with an alias assigned). Aliases prevent objects to be removed by the Garbage Collector, thus an operation to remove the alias of an object is also provided. More details on how the garbage collector works can be found in Section 1.5. Notice that the examples provided assume the initialization and finalization of user's session with methods described in previous section Sectionr̃efsec:PythonGlobalAPI.

```
def  delete_alias (cls, alias):
```

Description:
Removes the alias linked to an object. If this object is not referenced starting from a root object, the garbage collector will remove it from the system.

Parameters:
`alias:` alias to be removed.
Exceptions:
If no object with specified alias exists, a DataClayException is raised.

Example: Using delete_alias

```
newPerson = Person(name='Alice', age=32)
newPerson.make_persistent("student1")
Person.delete_alias("student1")
```

```
def get_by_alias (cls, alias):
```

Description:
Retrieves an object instance of the current stub class (from the method is being called), that references the corresponding persistent object with the alias provided.
Parameters:
`alias:` alias of the object.
Exceptions:
If no object with specified alias exists, a DataClayException is raised.

Example: Using get_by_alias

```
newPerson = Person(name='Alice', age=32)
newPerson.make_persistent("student1")

refPerson = Person.get_by_alias("student1")
assert newPerson.get_name() == refPerson.get_name()
```

5.3 Object extended methods: basic

In this section we present the basic extension to the object methods. This methods are intended to be used by standard programmers to handle the persistence of their objects.

```
def make_persistent (self, alias=None, backend_id=None, recursive=True):
```

Description:
Stores an object in dataClay and assigns an OID to it.
Parameters:
`alias:` a string that will identify the object in addition to its OID. Alias need to be unique per class. If no alias is set, then this object will not have an alias, will only be accessible though other object references.
`backend_id:` identifies the backend where the object will be stored. If this parameter is missing, then a random backend is selected to store the object. `api.LOCAL` can be set as a constant for backendID. When `api.LOCAL` is used, the object is created in the backend

specified as local in the client configuration file (as detailed in Section 7.1).
recursive: when this flag is TRUE, all objects referenced by the current one will also be made persistent (in case they were not already persistent) in a recursive manner. When this parameter is not set, the default behavior is to perform a recursive makePersistent.

Exceptions:
If the alias already exists or the BackednID is not a valid one, a DataClayException is raised.

Example: Using make_persistent

```
from dataclay import api
p1 = Person(name='Alice', age=32)
p1.make_persistent("student1", api.LOCAL)
assert p1.get_location() == api.LOCAL
```

5.4 Object extended methods: advanced

In this section we present the advanced extension to the object methods. This methods are not intended to be used by standard programmers, but by runtime and library developers or expert programmers.

```
def get_location (self):
```

Description:
Retrieves the location of the object.

Returns:
Backend ID in which this object is stored. If the object is not persistent (i.e. it has never been persisted) this function will fail.

Exceptions:
If the object is not persistent, a DataClayException is raised.

Example: Using get_location

```
newPerson = Person(name='Alice', age=32)
newPerson.make_persistent("student1", api.LOCAL)
assert newPerson.get_location() == api.LOCAL
```

```
def new_replica (self, backend_id=None, recursive=True):
```

Description:
Creates a replica of the current object.

Parameters:
backend_id: ID of the destination location in which the object should be replicated. If not provided, a random backend is chosen.
recursive: when this flag is TRUE, all objects referenced by the current one will also be replicated. When this parameter is not set, the default behavior is to perform a recursive

replication.

Returns:

The ID of the backend in which the replica was created. It is very important to realize that dataClay does not take care of replica synchronization. Details on how such synchronization can be achieved are described in Section 5.6

Exceptions:

If the object is not persistent or the location ID do not represent a valid location, a DataClayException is raised.

Example: Using new_replica

```
from dataclay import api
p1 = Person.get_by_alias("person1")
# replicating object and subobjects, from one of its locations to LOCAL
p1.new_replica(api.LOCAL)
```

```
def  federate (self, dataclay_name=None, recursive=True):
```

Description:

Federates current object with external dataClay.

Parameters:

dataclay_name: Name of the external dataClay. It must be previously registered.
recursive: when this flag is TRUE, all objects referenced by the current one will also be federated (except those that are already present in the destination dataClay). When this parameter is not set, the default behavior is to perform a recursive federation.

Returns:

Exceptions:

If the object is not persistent or the external dataClay is not registered, a DataClayException is raised.

Example: Using federate

```
from dataclay import api
p1 = Person.get_by_alias("person1")
# federating object and subobjects, from current dataClay to dataClay2
p1.federate("dataClay2")
```

5.5 Error management

DataClayException is the dataClay exception class that can be thrown from Python object operations. Currently it extends directly from java.lang.Exception providing the same API.

5.6 Replica management

In this section we present how to manage consistency for objects that instantiate Python classes and are replicated among different backends.
Let us suppose that we have our class Person:

```
class Person(DataClayObject):
    @dclayMethod(name="str", age="int")
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Once this class is registered and with the proper permissions and stubs, an application that uses it might look like this:

```
# Initialize dataClay
from dataclay.api import init, finish
from dataclay.DataClayObjProperties import DCLAY_GETTER_PREFIX

init()

from model.classes import *

if __name__ == "__main__":
    p = Person('foo', 100)

    execution_environments = p.get_execution_environments_info().keys()

    p.make_persistent(backend_id=execution_environments[0])

    p.new_replica(backend_id=execution_environments[1])

    p.age = 1000

    print(p.age)

    finish()
```

With no consistency policies, the printed message would show an unpredictable age for Alice, since getters and setters are executed in a random backend among the locations of the object.

In order to overcome this problem, dataClay provides a mechanism to define synchronization policies at user-level. In particular, class developers are allowed to define three different annotations to customize the behavior of attribute updates:

@dclayReplication(inMaster='boolean'
@dclayReplication(beforeUpdate='method'
@dclayReplication(afterUpdate='method'
@ClassField name type

The *inMaster* annotation forces the update operation to be handled from the master location if set to true.

On the other hand, *beforeUpdate* and *afterUpdate* define extra behavior to be executed before or after the update operation. The *method* argument specifies an operation signature of the class of current class. In this way, the developer is allowed to define an action to be triggered before the update operation, and an action to be taken after the update operation.

The *ClassField* annotation allows the user to define which fields of the class have to apply the defined behavior.

Let us resume our previous example. Assuming that the *name* attribute is never modified (e.g.

private setter), we want, however, that every time the *age* is updated the change is propagated to all the replicas. Empowering Person class with the proper annotations, we can intervene updates of attribute *age* to perform the update synchronization:

```
class Person(DataClayObject):
    """
    @dclayReplication(afterUpdate='replicateToSlaves', inMaster='False')
    @ClassField age int
    """

    @dclayMethod(name="str", age="int")
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @dclayMethod(attribute="str", value="anything")
    def replicateToSlaves(self, attribute, value):
        from dataclay.DataClayObjProperties import DCLAY_SETTER_PREFIX
        for exeenv_id in self.get_all_locations().keys():
        #if not exeenv_id is master_location:
            self.run_remote(exeenv_id, DCLAY_SETTER_PREFIX + attribute, value)
```

Notice that the class has now implemented the behavior to synchronize replicas through the method *replicateToSlaves*. That is, in this example, the master replica leads a sequential consistency model by synchronizing the contents with secondary replicas.

5.7 Object federation

In this section we present how to manage federation of objects that instantiate Python classes.
It is assumed a scenario with several dataClays running at the same time. The process of discovery to make a dataClay to be aware of other dataClays can be achieved by using dClayTool as exposed in section 6.5.
Notice that a current requirement for federating an object is that the class model of the object is registered in the involved dataClays within the same Namespace.
Let us suppose that we have our class Person:

```
class Person(DataClayObject):
    @dclayMethod(name="str", age="int")
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Once this class is registered in both dataClays in the same Namespace, and with the proper permissions and stubs, an application that uses it might look like this:

```
# Initialize dataClay
from dataclay.api import init, finish

init()
```

```
from model.classes import *

if __name__ == "__main__":
    p = Person('Alice', 42)

    p.make_persistent('person1')

    p.federate('dataClay2')

    finish()
```

At this point, an application in a secondary dataClay named *dataClay2* can execute the following code:

```
# Initialize dataClay
from dataclay.api import init, finish

init()

from model.classes import *

if __name__ == "__main__":
    p = Person.get_by_alias('person1')

    assert p.get\_name() == 'Alice'

    finish()
```

The secondary dataClay has actually performed a replica of Person object aliased *person1*. From now on, this replica can be used in the execution environment of any of the backends of the secondary dataClay.

IV dClayTool: dataClay management tool

6. dataClay tool

In this chapter we present *dClayTool*, the dataClay tool intended to be used for management operations such as accounting, class registering, or contract creation. It is very important to understand that this tool offers a friendly (and quite limited) version of the management API that should be enough for most dataClay users.

6.1 Accounts

In this section we present the options offered by the dataClay tool in order to manage user accounts.

```
dClayTool.sh  NewAccount  newaccount_name newaccount_pass
```

Description:
Registers a new account in the system.
Parameters:
`newaccount_name`: name of the new account to be created.
`newaccount_pass`: password of the new account.

6.2 Class models

In this section we present the options offered by the dataClay tool in order to manage classes such as registering a class model or obtaining the corresponding class stubs.

```
dClayTool.sh  NewModel
user_name user_pass namespace_name class_path language
```

Description:
Registers all classes contained in the class path provided. It is assumed that the class path is the main directory of the data model to be registered, and in case of containing

subdirectories these represent different packages/modules.
If the namespace where we want to include the class does not exist, it is created.
One of the supported languages must be chosen to specify which classes of the class path provided will be registered as part of the model.

Parameters:

user_name: user registering the model.
user_pass: user's password.
namespace_name: namespace where the model will be registered.
class_path: class path where classes are registered.
language: one of the supported languages (currently, *python* or *java*).

```
dClayTool.sh GetStubs
user_name user_pass namespace_name stubs_path
```

Description:

Retrieves the stubs of a certain class model registered in a namespace.

Parameters:

user_name: user requesting the stubs.
user_pass: user's password.
namespace_name: namespace of the class model to be retrieved.
stubs_path: folder where the downloaded stubs will be stored.

```
dClayTool.sh NewNamespace
user_name user_pass namespace_name class_path language
```

Description:

Registers a new namespace in the system.

Parameters:

user_name: user registering the namespace.
user_pass: user's password.
namespace_name: name of the new namespace.
language: one of the supported languages (currently, *python* or *java*).

```
dClayTool.sh GetNamespaces
user_name user_pass namespace_name class_path language
```

Description:

Registers a new namespace in the system.

Parameters:

user_name: user registering the namespace.
user_pass: user's password.
namespace_name: name of the new namespace.
language: one of the supported languages (currently, *python* or *java*).

6.3 Data contracts

In this section we present the options offered by the dataClay tool in order to manage datasets and data contracts.

```
dClayTool.sh NewDataContract
user_name user_pass dataset_name beneficiary_name
```

Description:
Registers a data contract to grant a user access to a specific dataset.
Only the owner of the dataset may grant users access to it.
If the dataset does not exist, it is created as a new dataset owned by the user registering this contract.

Parameters:
user_name: user that owns the dataset.
user_pass: user's password.
dataset_name: name of the dataset that will be shared through this contract.
beneficiary_name: user account that will benefit from this contract.

6.4 Misc

In this section we present other utilities that can be used to retrieve information from dataClay.

```
dClayTool.sh GetBackends user_name user_pass language
```

Description:
Retrieves backend names and hosts. Backend name can be used for LocalBackend in config file Section 7.1.

Parameters:
user_name: user requesting backend info.
user_pass: user's password.
language: one of the supported languages (currently, *python* or *java*).

6.5 Federation

In this section we present different commands to manage dataClay federation.

```
dClayTool.sh GetDataClayID
```

Description:
Retrieves the ID of the current dataClay. Current dataClay is the one the tool is connecting to.

```
dClayTool.sh RegisterDataClay dc_id dc_name dc_host dc_port
```

Description:
Registers an external dataClay with the provided info on current dataClay. Current dataClay is the one the tool is connecting to.

Parameters:
dc_id: id of the external dataClay
dc_name: name of the external dataClay
dc_host: hostname of the external dataClay
dc_port: port of the external dataClay

```
dClayTool.sh  GetDataClayInfo  dc_name
```

Description:

Retrieves the info related to the dataClay with the provided name. Info printed is: id, name, hostname and port.

Parameters:

dc_name: name of the external dataClay

```
dClayTool.sh  CheckDataClay  dc_name
```

Description:

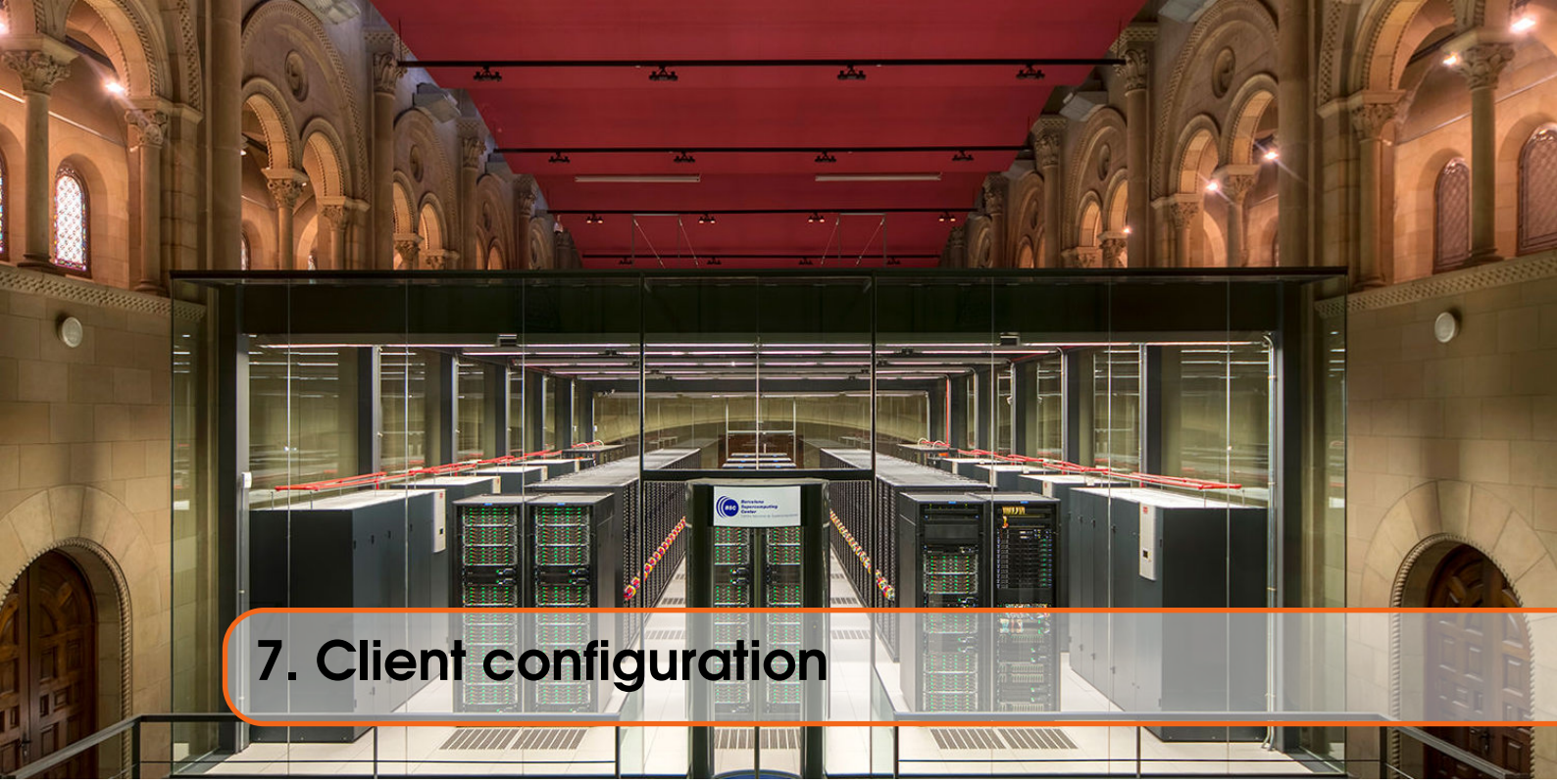
Checks that a external dataClay with provided name is reachable. The external dataClay must be previously registered in current dataClay.

Parameters:

dc_name: name of the external dataClay

V

Installation



7. Client configuration

7.1 Client configuration files

The basic configuration client for an application is the minimum information required to initialize a session with dataClay. In particular, and is shown in Tables 7.1 and 7.2, the file is a key-value text file to define certain properties. This file is automatically loaded during the initialization process (init()) if it can be found either in `./cfgfiles/session.properties` or in the path defined by the environment variable `DATACLAYSESSIONCONFIG`. Notice that *LocalBackend* will be referenced from the application by using either *DataClay.LOCAL* in Java or *api.LOCAL* in Python.

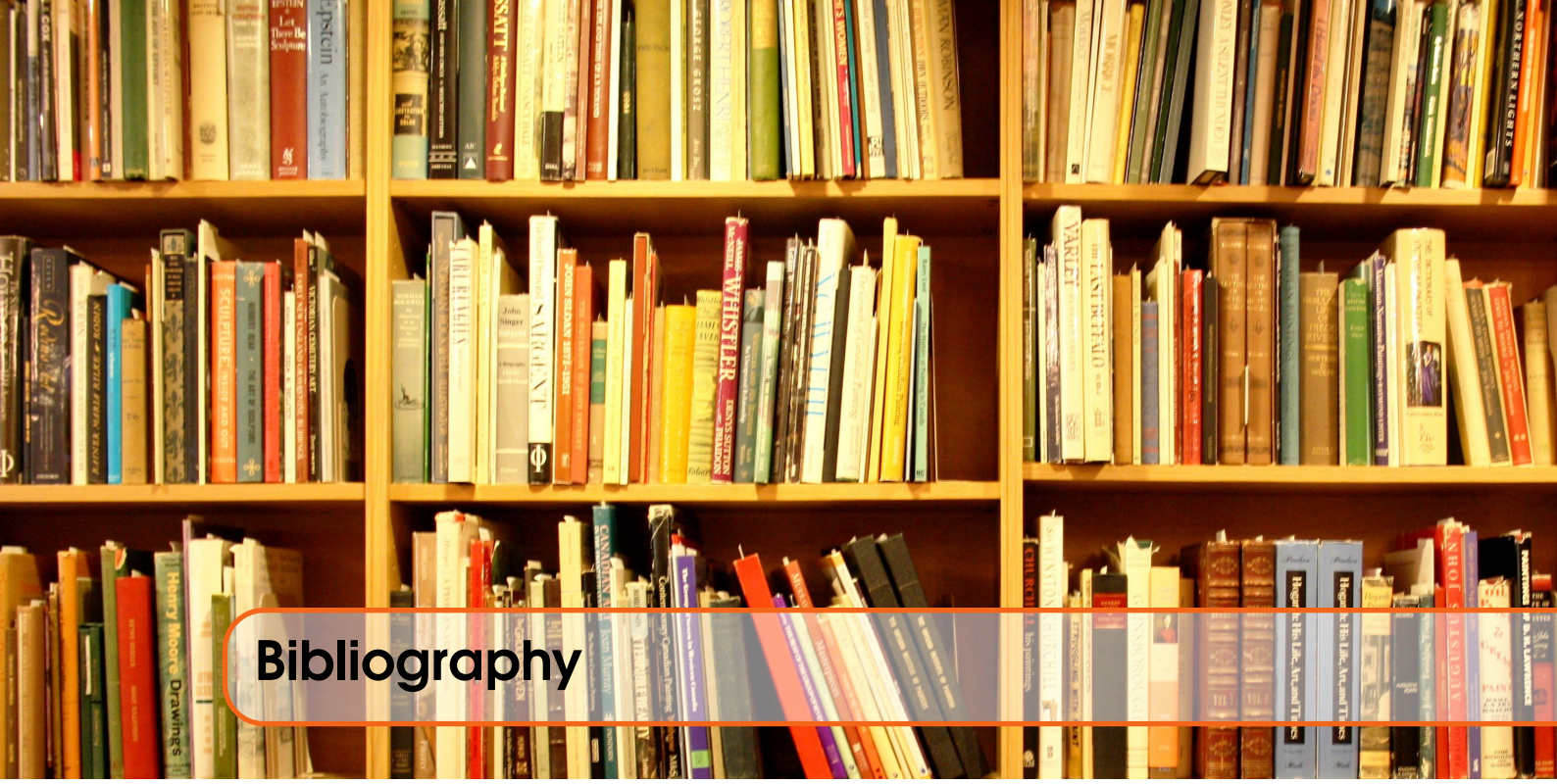
field	description
Account	User account
Password	User password
StubsClasspath	Directory where the class stubs are stored
DataSets	List of datasets that the application intends to access
DataSetForStore	Default dataset where data will be written
LocalBackend	Backend name that application may use with LOCAL reference.
DataClayClientConfig	Path to network configuration file (see Table 7.2)

Table 7.1: Init configuration file

field	description
HOST	IP address to connect with dataClay server
TCPPORT	Port to be used in the connection to dataClay

Table 7.2: Client network configuration file

VI Bibliography and index



Bibliography

[1] Jonathan Marí. “dataClay : next generation object storage”. PhD thesis. Universitat Politècnica de Catalunya, Mar. 2017 (cited on page 9).

[2] Jonathan Martí, Anna Queralt, Daniel Gasull, Alex Barcelo, Juan Jose Costa, and Toni Cortes”. “dataClay: a distributed data store for effective inter-player data sharing”. In: *Journal of Systems and Software* (May 2017) (cited on page 9).

Index