PiCaS User Documentation

Jan Bot

9/1/2014

v0.4

Introduction

PiCaS has been developed to solve one specific problem: how to manage large scale job distribution across heterogeneous compute resources. More specifically, it does this for problems where the number of individual jobs vastly outnumbers the amount of computational resources.

It is a so-called pilot job framework, a method of working explained in sections to come, but which boils down to a class of applications which are submitted to compute nodes, start running and fetch pieces of work from some central server. The application keeps doing this until either all work has been done, it run out of time on the worker node or it crashes.

This documentation should be read as a how-to for setting up your own code to run with PiCaS.

Requirements

PiCaS is build around CouchDB, which is therefore required. The PiCaS client libraries use Python 2.x and the couchdb python module.

Code availability

The PiCaS client source code can be downloaded from github[1].

Document structure

The document explains the working of PiCaS from the ground up, beginning with an introduction to CouchDB and working through the client library and closing with an example. To get stated quickly, you might want to jump directly to the example, only reading the other parts when necessary.

1 https://github.com/jjbot/picasclient

Pilot Job Systems

When working with grid or other large scale compute facilities and working your way through thousands or millions of jobs, keeping track of which jobs succeeded or failed can become tedious. Furthermore, as you could be working on a heterogeneous system, *e.g.* combining both cloud & grid computing, the job scheduling software cannot keep track of all tasks.

Pilot job frameworks deal with this complexity by decoupling the tracking of parameters from the job submission system. In such frameworks, a central database keeps track of all the work that needs to be done and client software pulls in work when active. The process is illustrated in Illustration 1.

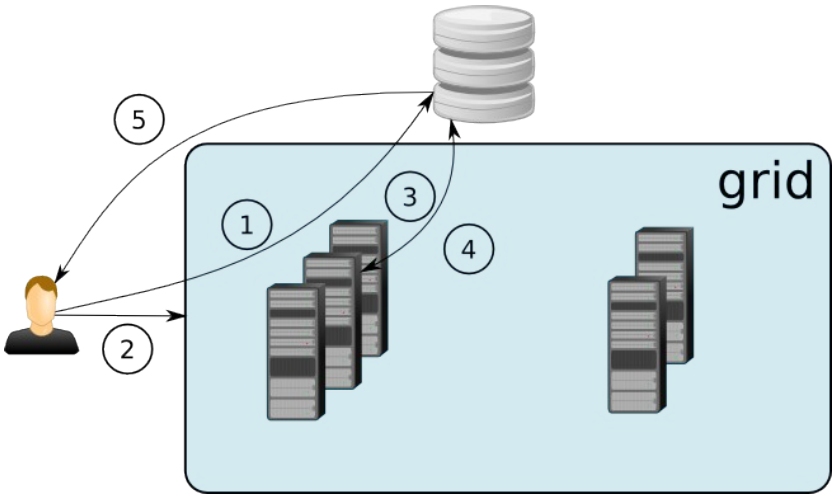

Illustration 1: Workflow of a pilot job system.

The workflow of pilot jobs systems: (1), the user uploads work to the central database. This could be a list of parameter settings that need to be executed using some application on a computational resource. The user then (2) submits jobs just containing the application to the grid, which handles retrieving from (3) and updating of (4) the job database. This process continues until all the work present in the database has been done, the application has crashed or the job has run out of time on the computational resource. When all work has been done, the user retrieves (5) the results from the database.

Server side

CouchDB

For PiCaS, CouchDB[2] is used as the central database. CouchDB is a document store, a type of NoSQL database which deals, as the name implies, with documents. These documents are self contained pieces of information and can contain anything, from arrays of number to binary data, or combinations thereof. These documents are structured as associative arrays, similar to maps in Java or dictionaries in Python, and are basically key-value pairs that can be nested arbitrarily. Every document has both a unique identifier and a unique version number, used to track changes made to a document. This also allows multiple clients to work on data simultaneously as the revision number can be used as a safeguard against conflicting document updates. Documents can be retrieved using their unique identifier or can be iterated over one by one. Each document can contain one or more tokens, depending on the preference of the user. Here, we assume that one document contains one token.

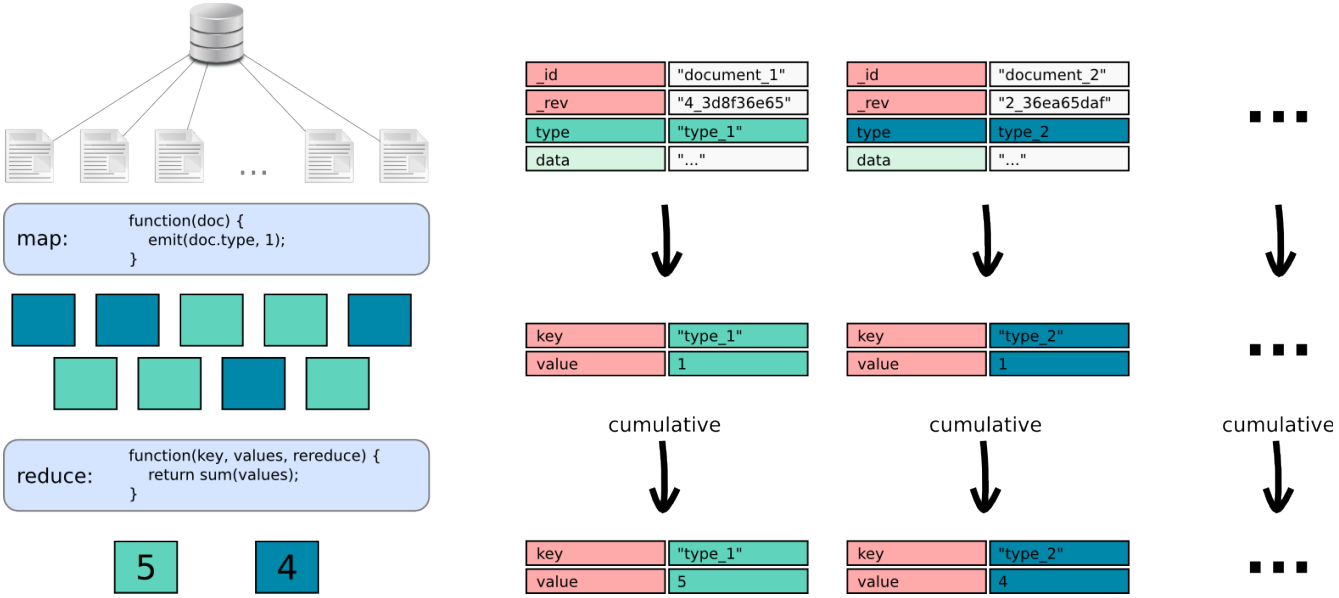

Illustration 2: The CouchDB view system.

To facilitate the easy retrieval of information, CouchDB implements 'views', predefined indexes which return parts or aggregations of individual documents. These views are constructed using a MapReduce model, which a user can implement using *e.g.* Javascript. In the Map phase, all documents are analyzed and emit key-value pairs which contain a subset of the information stored in a document. As the unique id and revision number are always part of this subset, the information can easily be traced back to the original document. The emitted key is used to sort and, when needed, group the results. The value contains the relevant information for the end user-application. For a complete description of CouchDB views, refer to the online documentation[3] or to the CouchDB guide[4].

CouchDB views are used to create queues, this prevents jobs from trying to update (overwrite) the

2 http://couchdb.apache.org/
3 http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views
4 http://guide.couchdb.org/draft/views.html

same document, which would lead to a significant amount of overhead as the update request will fail and the client needs to retrieve and re-update the document.

In the optional Reduce phase, all emitted pairs are bundled based on their key and are processed further to return one item per key, which subsequently means that the links to the documents that lead to this result are lost. The Reduce phase is only used when creating overviews, not in the queue code listed below.

Tokens

Tokens are the basic unit of work when using PiCaS. It should contain all the information needed to execute your target application. This can range from a single parameter setting to a range thereof, depending on *e.g.* the execution time, parameter granularity and application output size. The standard fields of a token are as follows:

- lock: a number (int) representing the time at which a token gets locked.
- done: a number (int) representing the time at which processing of the token was completed.
- hostname: the name of the machine that last modified the token.
- scrub_count: a number which keeps track of the number of times a token has been 'reset'.
- type: the token type, this is used to be able to have different tokens with different jobs within the same database.

Tokens can be modified using the PiCaS client library, the often occurring operations are shown in Illustration 3.

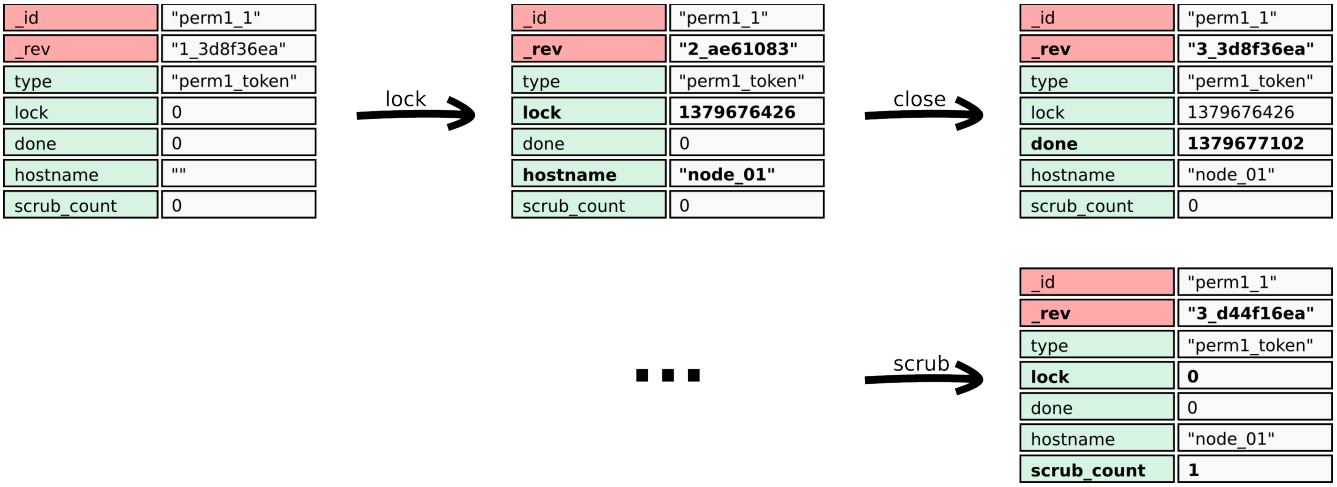

Illustration 3: PiCaS tokens encoded as CouchDB documents, changes to fields are in bold. The fields in red are required by CouchDB. The _id field stores the name of the document, the _rev field stores the revision number. This number is automatically incremented each time a document is changed. It contains the revision number itself and a random number to ensure the document isn't overwritten when multiple processes try to update it. The modification processes are straightforward: locking adds the current time and hostname to the token, closing the token updates the 'done' field and scrubbing changes the lock field to zero and increments the scrub_count by one.

Views

We somehow need to keep track of the status of tokens and, as we do not want to iterate over many tokens before finding an unclaimed one, we need some type of queue. These queues can be created using the CouchDB view system: we'll write some Javascript which takes the documents from the database and returns only unclaimed tokens. The tokens all have the structure as depicted in Illustration 3.

Three views are needed to keep track of the different states the tokens can be in: 'todo', 'locked' and 'done'. As to what they do: the names should be quite self explanatory. The following is an example of a 'todo' view written in Javascript. For these functions only the map phase needs to be filled in.

```
function(doc) {
 if(doc.type == 'token') {
  if(doc.lock == 0 && doc.done == 0) {
   emit([doc._id],[doc._id]);
  }
 }
}
```

Similar code needs to be written for the 'locked' and 'done' views. This can easily be done by changing the third line in the example above (keep in mind, the 'lock' and 'done' fields are time stamps represented by integers).

Client side

To make writing a grid application easier we have developed the PiCaS client library. This library allows you to use CouchDB as a token pool server. The main assumption for PiCaS is that the amount of jobs vastly exceeds the amount of compute instances (worker nodes).

The client library is written in Python, and although knowledge of this programming language is not required, it does help. For basic grid jobs you'll just have to implement one method, which we'll help you do in the following sections.

The most important class you'll have to deal with is the RunActor class from the PiCaS client `actors` module. This class makes sure that tokens are fetched and processed correctly. It can also contain any Python code you need to solve your problem. A basic version of this class is available as an example, for many grid applications it's enough to just modify one method (`process_token`) to make it execute your application.

PiCaS modules

Here, an overview will be given of the client modules of the PiCaS system.

Actors

Contains the main RunActor class, which needs to be overwritten when using the PiCaS system. This class, that the user needs to provide, contains all the code that is needed to get the application running.

Clients

Contains the class that connects to CouchDB and handles the document updates.

Executers

Contains some helper code to easily execute commandine applications.

Generators

Used to easily create tokens.

Iterators

Provides different iterators which can be used to walk through token queues.

Modifiers

Code that modifies tokens in such a way that they are compatible with the CouchDB views.

An example

Here, we will create an almost trivial example which still does something useful: calculating the square for a list of numbers.

Part 1: Setting up the database

1. Create a new database called 'square' in CouchDB, this can be done through the web interface, using cURL or through the use of one of the APIs. Make sure you have admin privileges otherwise this won't work.
2. Add tokens, when using the Python picasclient library you can use the generateToken function of the TokenGenerator class. To test, this can be done manually from the CouchDB web interface. A simple example of how to create tokens using Python is given below. Here, a list of 100 tokens is created. By setting the '_id' field you specify the name of the token in stead of letting CouchDB generate a random string.

```
from picas.clients import CouchClient
from picas.generators import TokenGenerator

tokens = []

for i in range(100):
  token = TokenGenerator.get_empty_token()
  token['_id'] = 'token:' + str(i)
  token['input'] = i
   token['type'] = 'token'
  tokens.append(token)

client = CouchClient()
client.db.update(tokens)
```

3. Create the necessary views under the design document 'example_app': todo, locked & done. You can also create an 'overview' view, which lists the total number of documents that are present in the other views. This can all be done using the CouchDB web interface. The default Javascript code for creating the 'todo' view is given below.
   Let's go through the lines one by one: `function(doc)` gives the function header CouchDB expects. Each document in the database is handed to this function in turn. `if(doc.type == 'token')` requires the type of the document to be of 'token', which are the only documents that we are interested in here. `if(doc.lock == 0 && doc.done == 0)` requires that no one else is working on the token (the lock is zero) and that the token also hasn't been processed before (done is zero). The `emit([doc._id], [doc._id])` line is more complicated: emit gives the output of the function, but can be called multiple times when necessary. It always returns two things, the key, used for ordering and grouping the results of a view and the value, which carries the content. Here, they are the same because we're only interested in the value. The value for PiCaS should be the location of the token in the database. As the token is the document itself, we only give the reference to the document, which is conveniently stored in doc._id.

```
function(doc) {
 if(doc.type == 'token') {
  if(doc.lock == 0 && doc.done == 0) {
   emit([doc._id],[doc._id]);
  }
 }
}
```

Part 2: Writing the client code

1. To get started with the PiCaS client you'll need some boilerplate code. Make sure to have the PiCaS client libaries in your Python library path.
   First, create the Python module '`square.py`' that will contain all the code needed for the application. Add a '`main`' section and create the CouchDB client, create a token modifier that will handle token updates and an Iterator that will allow you to receive tokens from the database. The code below shows you how to do this.

```
def main():
  client = CouchClient(url="http://localhost:5984", db='square')
  modifier = BasicTokenModifier()
  iterator = BasicViewIterator(client, "example_app/todo", modifier
   actor = ExampleActor(iterator, modifier)
   actor.run()

if __name__ == '__main__':
  main()
```

2. Create a class that overwrites RunActor (to be found in the Actors module). Here, the bulk of the work will be done. The process_token method should contain all the code you need to execute your application.

```
class ExampleActor(RunActor):
  def __init__(self, iterator, modifier):
    self.iterator = iterator
    self.modifier = mofifier
    self.db = iterator.client.db
```

3. You'll also need to implement the `process_token` function, this is where the code for your application should be. For now, you can forget about the key parameter in the function you need to implement.

```
def process_token(self, key, token):
   output = token['input'] ** 2
   token['output'] = output
   self.modifier.close(token)
   self.db[token['_id']] = token
   print str(token['input']) + '^2 = ' + str(output)
```

4. You will also need a bootstrap script which sets up the environment that you'll be working in on the grid worker nodes. This is usually a bash script which adds some libraries to your Python path, *e.g.* the couchdb & PiCaS modules.

Part 3: Dealing with failed jobs

In an environment with thousands to millions of jobs and with many machines working simultaneously on a problem it becomes very likely that a single compute job will fail. In the PiCaS system that means that a tokens is locked, but never completed. The amount of time that you allow before deciding that a job has failed depends on the application. Resetting tokens so they can be handed out again is a manual process, this means more work for you (the end user) but also allows for more control and gives you the opportunity to inspect the failed tokens, maybe discovering a pattern in the parameter settings that lead to failures.

Resetting tokens can easily be done in a small script. An example is given below.

```
def reset():
  client = CouchClient(url="http://localhost:5984", db="square")
  modifier = BasicTokenModifier()
  iterator = BasicViewIterator(client, "example_app/todo", modifier)
```

```
actor = ExampleActor(iterator, modifier)
status = actor.scrub_all("example_app/locked")
```