IT 11 045
Examensarbete 15 hp
Juni 2011

Implementing a eventual consistency job distribution with CouchDB

Simon Unge

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Implementing a eventual consistency job distribution with CouchDB

Simon Unge

The implementation of a job distributer in a distributed system can be a complex task. There is no shared memory or a reliable global clock, and network failures can, and will, occur, making communication between the different nodes of the system complex.
To make life easier for the programmer, I have evaluated if CouchDB can be used as tool for reliable communication and synchronization between nodes. The results shows that CouchDB indeed can be used for this purpose. CouchDB takes care of the communication, and guarantees that the nodes will, eventually, get the jobs to execute.
With CouchDB as the backbone in a job distributer, the programmer can concentrate of the functionality of the job distributer, and not worry about the need to implement a reliable communication between nodes with all the complexity that comes with it.

Handledare: Thomas Lindgren
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 11 045
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

A distributed system is next to useless if one does not take advantage of the fact that jobs can be divided between the nodes[1] in the system. One way of doing this is to have a job distributer, a tool used to distribute and manage the jobs, making sure that the jobs are executed somewhere in the system and make sure that each job is not executed more than once.

In distributed system there is no shared memory[6]. The lack of a shared memory between nodes means that all communication is done via message passing, so a reliable communication between the nodes in the system is necessary so that the jobs can be distributed. One advantage of distributed system is that if the job burden is too much for the current setup, more nodes can be added to take some the burden.

In this thesis, I want to evaluate if it is possible to build a job distributer that uses CouchDB as a tool used for a reliable communication between nodes, synchronization of jobs and make the job distribution easy to scale.

1.1 Contributions

- Demonstrate that CouchDB can be used as a tool to distribute jobs in a network
- That it can be scaleable
- Guaranteeing eventual consistency
- That the task is performed only once.

[1] A node, in the context of this thesis, is a computational entity in the distributed system

2 Background

This part will give the necessary background information needed to grasp the content of this thesis.

Section 2.1 describes what consistency in a distributed system means(section 2.1.1), and one model of consistency called eventual consistency(section 2.1.2) as well as the CAP theorem which explains why a strong consistency is not always possible(section 2.1.3).

Section 2.2 describes CouchDB, a database management system used as the main tool in the thesis, which has one important replication feature which will be used to synchronize databases, described in section 2.2.4

2.1 Distributed systems

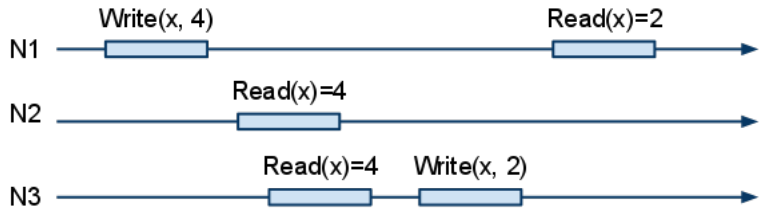
2.1.1 Consistency

A consistency model in a distributed system is a guarantee from the distributed system about the relation between an update to an object and the access to an updated object.

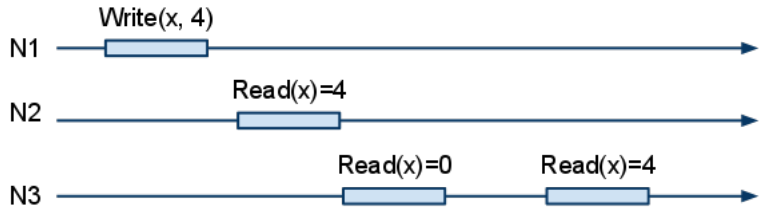
The strictest from of consistency is called strong, or atomic, consistency[6]. Strong consistency guarantees that when an update is made to an object, the update is immediately availably through out the system, and thus all attempts to access the object will return the updated version.

Figure 1a shows a simple example of strong consistency. In the figure, node N1 is making an update to object x with the value 4, and attempt to read object x by node N2 and N3 results in the updated value.

In a distributed system, the lack of a global clock - each node having the exact same time - makes it costly to achieve strong consistency[6]. Therefore there exists weaker models of consistency, where the system does not guarantee that any subsequent access to an updated object will result in the latest version. There are a number of weaker consistency models. Of particular interest in this thesis is the eventual consistency model.



(a) Strong Consistency



(b) Eventual Consistency

Figure 1: Consistency Examples. An update to an object x - the objects are simple one valued integers - is represented by "write(x, Value)" where Value is the a new value written to x. An access to the object x is represented by "read(x)=ReturnValue" where ReturnValue is the value returned when accessing the object.

2.1.2 Eventual Consistency

Eventual Consistency is a weaker form of consistency model[7]. It guarantees that if no new updates are made to an object, eventually the updated object will have replicated through the system to all affected nodes and all access to the local copy of the object will return the latest update. However, due to network delays, system load and the number of replicas needed, the time required for an update to become consistent may vary.

The time between performing an update and it being available on all nodes is called the inconsistency window[7]. During the inconsistency window replicas can be inconsistent and an attempt by a node to access the local copy of the object may not return the latest version of the object, with the consequence that accesses on two nodes can get two different results.

Figure 1b shows an example of an update made on an object in a distributed system that guarantees eventual consistency. In the figure, there are three nodes that replicate each other. Initially, x has the value 0. Node N1 is updating x with the value 4. Node N2 reads the value 4 from x, which is the latest value. Node N3 reads the value 0 from x, which is an old value of x. Subsequently N3 makes another read of x, this time the latest value, 4, is returned.

2.1.3 CAP Theorem

According to the CAP Theorem[4], it is impossible for a distributed system to uphold the following three properties at the same time:

- Consistency. Strong, or atomic consistency, described in section 2.1.1
- Availability - Every request received by a non failing node must be result in a response (although not necessarily immediately).
- Partition tolerance - To be able to split the database over several servers.

It can however, uphold two of them. In the case of the thesis problem, high availability as well as partition tolerance is of priority. According to the CAP theorem, this will be at the cost of strong consistency.

2.2 CouchDB

The main tool used in this thesis is CouchDB, a document based database server implemented in Erlang. In a document based database, data is represented by a collection of documents, each of which can contain several fields and values[5]. The documents are self-contained, meaning that there are no relation between any two documents - they are independent from each other - and unlike SQL databases there is no strict schema to follow when

```
{
        "_id" : "2737419288317x134agu771h98",
        "_rev" : "3-ff2e79d0625ee249bet2003o98",
        "Name" : "John Doe",
        "Occupation" : "Programmer",
        "Interests" : ["Climbing", "Walking", "Crawling"]
}
```

Figure 2: CouchDB Document

creating or altering a document. Intuitively a document can be thought of as a business card. Two different business cards can have totally different content, one representing a person, containing fields like "Name: John Doe" and "Occupation: Drug dealer" and one representing a company, containing fields like "Company name: Foo AB" and "Motto: We love our customers". Both are business cards, but with no other relationship or common content other than being business cards.

In CouchDB the documents are JSON objects[10] that consists of field/-value pairs. Field values can be strings, numbers, or lists containing datatypes, or even nested JSON objects. Each document must have a identity field, "_id", which is unique inside the database and a revision field, "_rev", which is maintained by CouchDB, and is used for conflict handling and revision control (see also section 2.2.1).

Figure 2 shows an example CouchDB document. In the example, the document has five fields. The fields "Name" and "Occupation" have strings as values, and the field "Interests" has a list of strings as its value, ["Climbing", "Walking", "Crawling"].

CouchDB is an HTTP server. Each CouchDB server can host several databases, and each database can store many documents. The server is accessible though a RESTful[2] JSON API. To access documents residing in a database, the client issues HTTP requests, such as 'GET', 'PUT', 'PUSH' and 'DELETE' to a URI[3] along with additional JSON data in the request

[2]Representational State Transfer[3]
[3]Unified Resource Identifier[12]

```
First create a database called 'fruits':
$ curl -X PUT http://127.0.0.1:5001/fruits

CouchDB confirms the creation with a JSON object:
{"ok":true}

Then add a document to the database 'fruits', in this case data about
a fruit and its characteristics to the newly created 'fruits' database,
a JSON object is passed as data:
$ curl -X PUT http://127.0.0.1:5001/fruits/banana -d '{"colour": "yellow"}'

CouchDB respond with a confirmation and the id and revision of the document.
The document banana resides on address 'http:127.0.0.1:5001/fruits/banana':
{"ok":true, "id":""banana, "rev":"1-oeff2t43d998o770ou"}

To retrieve the document just created, use a GET request to the address of
the document:
$ curl -X GET http://127.0.0.1:5001/fruits/banana

CouchDB response with the JSON representation of the document:
{"_id":"banana", "_rev":"1-oeff2t43d998o770ou", "colour":"yellow"}

And finally delete the document. For this we need the id and rev of the
document:
$ curl -X DELETE http://127.0.0.1:5001/fruits/banana?rev=1-oeff2t43d998o770ou
```

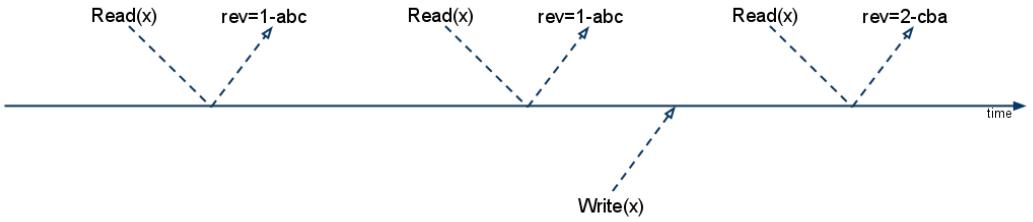
Figure 3: Document creation. In the example, an instance of CouchDB is assumed to be running and is located on 'http://127.0.0.1:5001'. The communication is done via a command-line utility called 'curl'[8] which lets you issue HTTP requests via the terminal.

body. The server processes the request and responds accordingly, with a JSON object or error message.

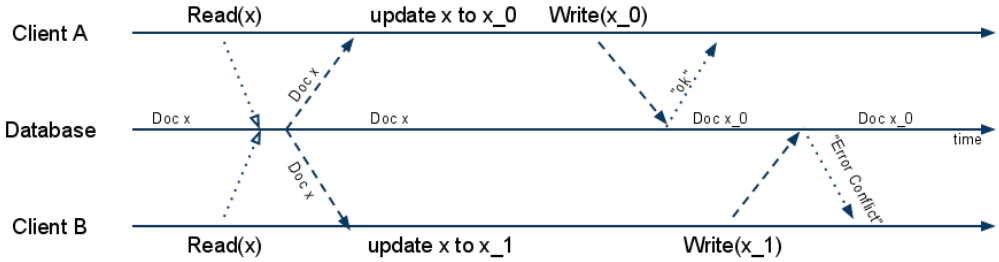
Figure 3 shows how to create a database, add a document to it, request the document, and then delete the document.

2.2.1 Document Revisions.

CouchDB uses Multi-Version Concurrency Control(MVCC) to manage access to the documents[1]. MVCC is a method that provides non-blocking read access to a database. The idea is to let the clients get a snapshot of the database, instead of working directly on the database. The data is versioned and a read request will return the latest current version of the data. An



(a) CouchDB Revision Timeline.



(b) Update Conflict

Figure 4: Revision examples.

update to data does not overwrite the current data, but instead adds newer version of the data, while the old version is kept unchanged. This way there is always a consistent, latest version of the data, and there is no need to block read operations to make sure there is not a write in progress, and no need for a write operation to wait for a read to be ready since the write will always create a new revision. When the write operation has completed, the new data will be seen as the latest version.

CouchDB implements MVCC by giving each document a revision field that is maintained by CouchDB. The revision value is on the form "N-xxxxxxxxx" where the N is an integer denoting the number of times a document has been updated, and the x's is a MD5 hash made from the contents of the document JSON data. As explained above, when updating a document, CouchDB does not overwrite the document, but instead makes a new version of the document, with a new revision value, with N incremented by one, and

archiving, keeping it as an older revision of the document, the old version.

Figure 4a shows how CouchDB works in a non-locking manner. In the figure, three read-operations to document 'x' are issued. The first two reads results in the same version of the document, i.e the same revision-value. The third read is performed after an update to the document, and the returned document thus has a new revision-value.

The revision field is also used to detect update conflicts. When a client wants to update a document, the revision-value of the document needs to be included in the request to CouchDB to check if the client was working on the latest version of the document. If two users are trying to make an update to the same revision, say "1-abc", CouchDB will only accept one of the updates, the one that CouchDB gets first, and increase the revision-value of the document to "2-cda". When the second user tries to commit its update, still including the old revision-value "1-abc", CouchDB will reject the update, since the user is probably overwriting data the user did not know existed.

Figure 4b illustrates an update conflict. In the figure, two clients request the same document, x. Then they both makes changes to the document, resulting in two new versions of the document, x_0 and x_1. Client 1 writes its x_0 to the database first, making it the latest revision of the document in the database. Then client 2 writes its update x_1 to the database, the update is rejected since there is a newer version of x in the database.

CouchDB also uses the revision number when detecting conflict in replication between databases, as described in section 2.2.5.

2.2.2 Changes.

Clients may need to view the revision timeline. For this purpose, CouchDB comes equipped with a 'changes'-API[1]. A 'change' is a JSON object containing information about a when document was updated. It contains three fields: "seq", "id" and "changes". The "seq" value is an integer which represents an update sequence number of the change made to the database. The update

```
To request continiuous changes from a database:
$ curl -X GET http://127.0.0.1:5001/fruits/_changes?feed=continuous

Example of change object.
{"seq":3, "id":"banana", "changes":[{"rev":"1-oeff2t43d998o770ou"}]}
```

Figure 5: CouchDB Changes

sequence number is a integer value stored internally by CouchDB which is incremented each time a change is made to the database. The "id" value is the document id that was altered and the "changes" value is list containing the documents revision value and possible additional information about the document such as if the document has been deleted.

Changes can be polled by the client or they can be given to the client in a continuous stream. When polling for changes to a database, CouchDB will return list of changes-objects, one for each document in the database that has been changed since a given update sequence number, along with a "last_seq" number, indicating the last update sequence number. The purpose of the sequence number is for a client to be able to ask "What changes has been made to the database since sequence number X?".

When a continuous stream is used, an HTTP connection between the client and CouchDB is opened "forever"(until closed by the client or due to network failure). Whenever a change is made to the database, the client gets a notification in the form of a changes-object.

Figure 5 shows examples of how the changes-API is used and what a change object looks like.

2.2.3 Views.

CouchDB uses map and reduce functions, combined called 'views', to implement queries. These views are Javascript functions defined and stored in documents called design documents. Map functions are functions that take each document in the database as the argument, and either ignores the

```
function(doc) {
    if(doc.colour == "yellow") {
        emit(doc._id, null);
        }
}
```

Figure 6: View function

document, or emits one or more view rows as key/value pairs.

Figure 6 shows a that function would emit all documents in the database which are yellow fruits, with the document id as the key and "null" as value.

2.2.4 Replication.

One important feature of CouchDB is replication. Replication is the process of synchronizing two copies of the same database, either within an CouchDB server, or between two different CouchDB servers[1]. A replication is uni-directional, having a source and a target database. When a replication is triggered, explained in more detail below, CouchDB will compare the source and target databases to find out which documents from source differ from which documents on target, and transfer the documents that differ. The documents that differ could either be new documents that do not exist on the target database, or newer versions of documents(See section 2.2.1).

A replication can be either triggered or continuous. A triggered replication replicates all changes made to the source database since the last replication, while continuous replication periodically replicates changes made on the source database to the target database (CouchDB has a complex algorithm to determine when to replicate during continuous replication[1], the details of the algorithm are beyond the scope of this thesis).

It is with this replication system that CouchDB achieves eventual consistency between databases.

A replication can be made bi-directional be creating two replications, one from database X to database Y and one from Y to X. The replication can

```
To trigger a replication between two databases on the
same CouchDB server:
$ curl -X GET http://127.0.0.1:5001/_replicate -d \
  '{"source":"database", "target":"database-replica"}'
```

Figure 7: Replication

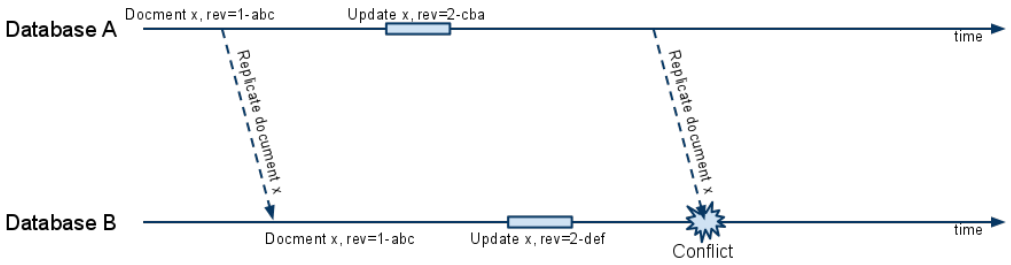


Figure 8: Replication Conflict.

either be "pushed", where the source database is located on the server which gets the replication request, or "pulled", where the target database is located on the server which gets the replication request.

Figure 7 shows examples of how to set up replications between databases. A JSON object containing information about source and target is passed as additional data to the server.

2.2.5 Conflicts and resolution between databases.

When databases replicate each other (see section 2.2.4), conflicts may arise. This happens when CouchDB notices that the latest source and target version of a document X have different revisions(see section 2.2.1).

Figure 8 shows how a conflict can arise during replication. In the figure, database A contains one document, x, with revision value "1-abc". Database A triggers a replication with Database A as source and Database B as target, transferring document x. Database A and B updates document x, resulting in new revisions of x, "2-cda" on database A and "2-def" on database B.

Database A triggers a second replication, transferring the new revision of document x. Database B detects a conflict between the replicated revision of x and the local revision of x.

CouchDB does not resolve conflicts, but chooses one of the versions as the latest version. When CouchDB discovers a conflict, it marks the document with a conflict flag, and then decides which version is the winning one, saves the winning version as the latest revision, and the losing version as the previous revision. The algorithm to decide the winning version is deterministic[1]: each revision comes with a revision history list, where all previous revisions values of the document are stored, and the revision with the longest revision history list becomes the winner. If both revisions have equally long history lists, the _rev fields of the conflicting documents are compared, and the one with the highest ASCII sort order becomes the winning version. The deterministic decision ensures different CouchDB servers will come to the same decision given the same data.

3 Problem Description

The goal of this thesis is to evaluate the possibility to use CouchDB, and its replication functionality, as a mean to distribute jobs in a distributed system in a fault tolerant and consistent manner. The thought of distributed system will consist of several nodes located of different geographical parts of the world.

3.1 Nodes

The distributed system will consist of several nodes. On each node, a CouchDB server will be running. The database server will contain a database which purpose is to distribute jobs (see section 3.2) in the system. The database needs to be replicated bi-directionally (see section 2.2.4) with one or several

of the CouchDB servers in the system, so that they are strongly connected, in order to distribute and synchronize jobs (see section 3.2.1).

3.2 jobs

A job is something that the system needs to execute. Each job contains one or several discrete job steps. A job step will contain a Unix shell command for the system to perform. A job step could be specified to be executed on a certain node, or set of nodes, or it can be specified to be executed on any available node.

Each job step must be executed in strict order - step 1 is followed by step 2 etc - and a step must finish successfully before the next step can be executed. Each step must be executed once, and only once.

A job step can either fail or succeed. If a job step fails, the exit status must be caught and alternative actions should be able to execute instead. If the job step succeeds, the next job step is allowed to be executed.

3.2.1 Synchronization of jobs

As mentioned in section 3.2, a job contains job steps. To guarantee that the steps are executed in order, and only executed once, the nodes needs to synchronize their databases with each other and somehow ensure that only one nodes executes a job step, and that they are executed in the right order.

3.3 Diino Case study

Where could a system like this be useful? The idea behind this thesis was created at Diino AB, a company who gives a service handling backup of data towards companies and individuals.

The users of Diinos services are spread out in the world. To give the users a faster service, Diino has nodes in different geographical locations, so that the distance between a user and a node is as short as possible. Diino has

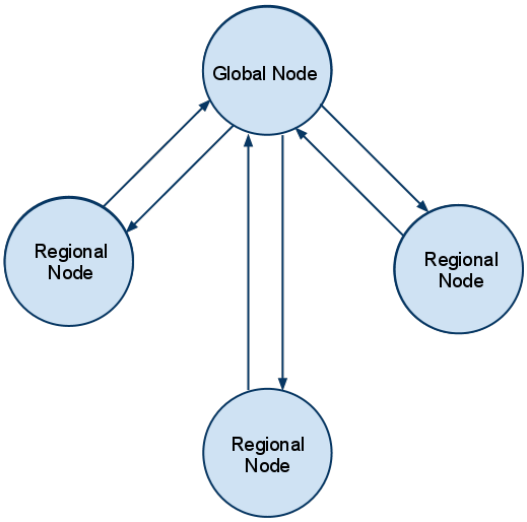


Figure 9: Exampel of Diino node layout

jobs that needs to be executed on one or more of the nodes. Some of the jobs will have steps that needs to be executed on different nodes.

3.3.1 Diino node layout

Diino has two kinds of nodes, a Global node and Regional Node. The Global node, of which there is only one, is the main node in the system. It is on the Global node that most of the jobs (see section 3.2) that the system needs to perform will be created. It is be possible, however, for jobs to be created on the regional nodes. Jobs are created on the global node according to the scheme described in this thesis. After a job is created, the job is propagated through the system to the regional node(s), so the Global node will need to synchronize with every Regional node for each job.

There may be several regional nodes, installed in different geographical locations.

Figure 9 shows a possible synchronization layout between Diino nodes. In the figure, there is one Global node and three Regional nodes. The arrows between the nodes denote bi-directional synchronization.

3.3.2 Diino specific job

"Create User" is a Diino specified job that exemplifies the use of parts of the requirements of jobs described in section 3.2. The "Create User" job is a job with three steps with specific targets on each step.

The first step is to be executed on the global node. In this step the information about the user (username, password etc) is collected and then sent to a regional node, where the information will reside.

The second step is executed on a specific regional node. The information from sent from the global node is stored on the regional node.

The third step is executed on the global node, where the job is confirmed to be finished. This type of job shows the use of the option to specify a target for each step.

4 Design

In this section, I describe how I, with assistants of Diino, have designed a job distributor which hopefully fulfills the requirements stated in section 3. The the design is divided into two parts.The first part, 4.1, describes how the job document is designed, and the second part, 4.2, describes the different parts of the job distributer which uses the job document to execute the job.

4.1 The Job document

The job document will be a single CouchDB document, and will contain all information about the job. By this I mean that the job document will be the only source of information about the job; the current state of the job; which step that is executing; which node that will execute the step, and which worker that will do the work; if the step executed correctly or if it did not. The contents of the document is explained in detail in this section.

```
{
    "_id": "a9f92b386d2258109a5ee57b191e728c",
    "_rev": "8-391e15a84e5d1d2eace2a1d1dd3727bd",
    "creator": "global_node",
    "step": 1,
    "job": [
        {
            "target": "reg_a",
            "do": "sleep 5",
            "alt_do": null,
            "claimed_by": "reg_a",
            "winner": "reg_a",
            "executioner": "Node: reg_a, worker: <0.64.0>",
            "step_status": "Finished",
            "exec_time": 5.006715,
            "start_time": "14:45:4",
            "finish_time": "14:45:9",
            "retry_strategy": {
                "max_retries": 3,
                "sleep": null,
                "sleep_factor": 2,
                "sleep_max": 10
            }
        },
        {
            "target": "any",
            "do": "sleep 5",
            "alt_do": null,
            "claimed_by": "reg_b",
            "winner": null,
            "executioner": null,
            "step_status": null,
            "exec_time": null,
            "start_time": null,
            "finish_time": null,
            "retry_strategy": {
                "max_retries": 3,
                "max_time": null,
                "sleep": null,
                "sleep_factor": 2,
                "sleep_max": 10
            }
        }
    ],
    "job_status": null
}
```


Figure 10: Job document

Job document fields

_id: This is a the unique identifier of the job.

_rev: The current revision.

creator: The node id on which the job was created.

step: The index of the current job step, starting from 0, used to track the current step in the job array to guarantee the steps are performed in strict order.

job: An array, or list, of all the steps the job contains. Each step contains several fields of its own.

job_status: This is the finish status of the job. It can be "null", "failed" or "success".

Job step fields

target: The target on which the step should be executed. This could be a specific node, a set of nodes or, or all nodes.

do: The command that should be executed. In this thesis, the commands are limited to unix shell commands, but could in theory be anything. For example RESTful commands.

alt_Do: The command to execute if the Do-command fails, if any.

claimed_by: The node identifier of the node that attempts to claim the job step

winner: The node identifier of the node that is the winner of the claim, and thus the node that will execute the step

executioner: The node identifier and process identifier of the step

step_status: The current status of the step. The status is the state of the step, and could for example be "running", "finished" or "failed".

exec_time: The time in seconds it took to execute the step.

start_time: The time stamp of when the node started executing the step. The time stamp is the local time of the node.

finish_time: The time stamp of when the node is finished executing the step. The time stamp is the local time of the node.

retry_strategy: The retry strategies for the step. This field contains the following several fields of its own.

Retry strategy fields

max_retries: The number of retries that should be performed if the do-command fails.

sleep: An integer of seconds between each retry. The sleep time is either fixed, or increases between each retry. The increase size is determined by the sleep_factor field.

sleep_factor: An integer to multiply the sleep field with on each retry.

sleep_max: The maximum allowed seconds between each retry.

Figure 10 shows an example of a CouchDB representation of a job document used in the thesis.

4.2 The Job Distributer

4.2.1 Listener

Each instance of CouchDB has a "Listener". The Listener is a process which role is to set up a continuous changes feed(see section 2.2.2) to a CouchDB

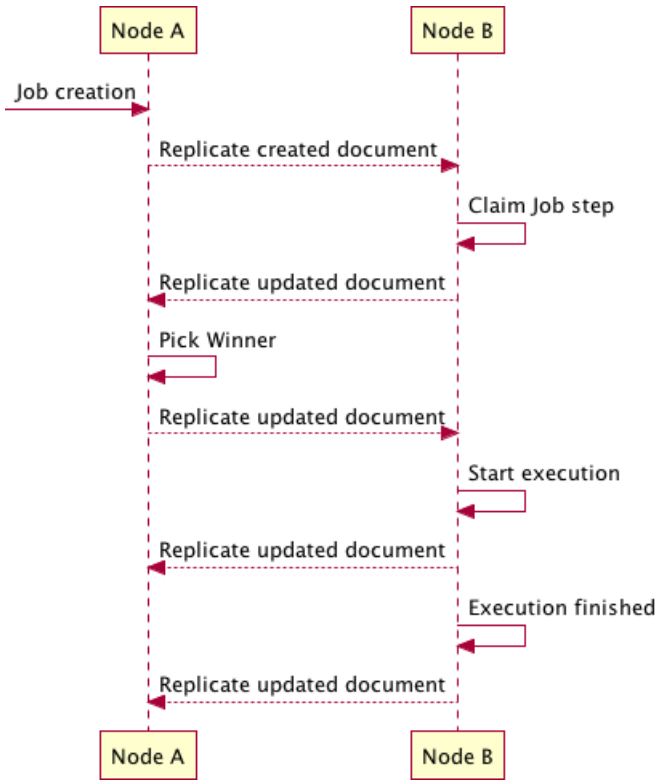


Figure 11: Update flow. The filled in arrow lines represent a save of a document. The dotted arrow lines represents a replication of the document.

database, and handle the changes information it receives from CouchDB. Upon receiving a change on a document, the listener sends the change information to a work manager (See section 4.2.2), and then waits for the next change to arrive. The contents of the message is the document id of the document that has been changed, and on which database the document resides.

4.2.2 Work manager

The work manager is the core process of the system. The work manager is a process which receives messages from the listener and the workers, and makes, most, of the decisions (the workers will also, to some extent, make

decisions regarding retry strategies, as explained in section 4.2.3) based on the messages that it receives.

The work manager receive messages from the listener. These messages contains the id of the affected document, and information about the database on which the document resides. The database name works as the node identifier.

Upon receiving a message from from the listener, the work manager retrieves a copy of the affected document from the database, and goes through its content. Based on the content of the document (see section 4.1), the work manager takes different actions.

First of all, the work manager checks whether or not the job still has steps that has not been executed. Then if the node it is affected by the current job step. This is done by examine the "target" value of the current job step. The node can be affected in two ways; either directly by being the specific target of the job step, or indirectly by being part of a subset of nodes that can handle the step.

If the node is the target, the work manager gives the job step to one of the workers, if there are free workers (described below), and update the document with the current status of the job, incrementing the "step" value with one, and save it to the local database. Each time a document is changed, or updated, in any way, it is saved to the database so any observer can follow the work flow of the job. If there are no free workers, the work manager will ignore the document, and await the next change.

If the node is part of the subset of targets, the work manager will try to claim the job. The work manager claims the job by updating the "claimed_by" value of the current step, if no other node has already claimed the step. Before making a claim, the work manager will reserve a worker for the step. If there are no free workers, the work manager will not try to claim the step. After a claim is made, the work manager will save the document on the database, and wait for a decision.

Each job will have a creator. The creator is simply the node on which the job was created. It is the creator of a job who decides which node that will execute a claimed step. The work manager on the creator node does this by examine the "claimed_by" value of a document, and simply set the claimer as the winner, and saves the document to the local database. Although it is possible that several nodes tried to claim the job step, the creator node will only see one of these claims, thanks to CouchDB replication conflict management (see section 2.2.5).

When a node sees that it is the winner of a step, it will give the job step to the reserved worker, and updating the document with the current status of the job, incrementing the "step" value with one, and save it to the local database.

Each work manager has a fixed number of workers under its deposal. As explained above, the work manager needs to know the state of the workers to be able to give job steps to free or reserved workers. The work manager is always able to get information about the current state of its workers. The workers can be one of the following states: free, booked, busy.

When the work manager claims a step, it changes the state of one of its free workers to booked, and when the work manager gives a job step to a worker, the state is changed to busy. If the node did not win a claim, the reserved(booked) worker for the claim is freed. When a worker is done with a step, the work manger frees the worker, handles the result of the step, and updates the document to the local database.

Figure 11 illustrates the flow of a job documents updates in the system. In the figure, the job is created in node A and replicated to node B. Node B claims the job step, and saves the updated document on its local database. The document is then replicated back to node A. Node A decides that node B is the winner of the step and saves the document to its local database. The document is replicated back to Node B. Node B sees that it is the winner of the claim, updates the document, with information about which process

is executing the step etc, saves the document to its local database, and then starts executing the job step. The document is replicated back to node A. When node B is finished executing the job step, it updates the document and saves it to its local database, and the document is replicated to node A. The total number of updates to the document for one step was four. And the document was replicated five times.

The work manager handle failures. A failure could be that the work manager for some reason was restarted, along with all the workers. As mentioned in section 4.1, the job document will be the only place where information about a job is stored. This means that the work manager always needs to check if the promises made by the work manager still is true.

If the node is the winner of a job step, the work manager checks if it really has a booked worker for the step. If it has not, the "claimed_by" made on the step will be set to null on the document, as well as the "winner" value, and then the document is saved to the database, making it possible for nodes to claim the step.

If the work manager sees that it is currently the executioner of a step, by examine the "executioner" value, it checks if it really has a busy worker handling the step. If not, it means that the worker that executed the step died for some reason. The work manager will then, as with the booked worker failure, set the "claimed_by" value and "winner" value to null, as well as the "executioner" value.

Another type of failure is when the job step does not succeed. The worker handling the step will send the result of the attempts to execute the step to the work manager. If the step did not succeed, the whole job is seen as a failure. The work manager sets the "step_status" field in the document to the result received from the worker, then sets the "job_status" field to "failed".

4.2.3 Workers

A worker is a process which executes the job steps. The worker will receive the job step to execute from the work manager, and try to execute the command. The command is a unix shell command stored in the "do" field of the job step. If the command succeeds, a succeed message is sent to the work manager. If the command fails, the worker will check if it should retry the command, and how many times it should retry the command, how long it should wait between retries, and if it should try to do some alternative command if all retries fail. This information is stored it the job steps "retry_strategy" field. If none of the retries succeed, the worker will send the result status of the command as well as the result status of the alternative command, if any.

4.3 Practical job distribution design.

The design described in section 4.2 is not practical in a larger node system. Depending on the node replication setup, the number of replications can be very costly. If all nodes replicate each other, the number of replications per update to a document is, in the worst cases, approximately $(N-1)^2$, where N is the the total number of nodes. If the job step can be claimed by all nodes, the number of times the document will be updated is four times(see section 4.2.2), giving a total of $4*(N-1)^2$ replications per job step. Clearly the job distributer is just not scalable in that type of setup.

To make the design more scalable, the number of replications must be reduced drastically. One solution is to let one node be the only node where jobs can be created, and let all other nodes only replicate that node. .This would drastically reduce the number of replications.

Another action to reduce the number of replications is to use a replication filter. This filter should make sure that only affected nodes receives a replicated document

Another weakness in the design is fault tolerance. If the job creator nodes

CouchDB server fails for some reason, the job is stalled until the server is restored. A solution to this could be to have multiple CouchDB instances per node, so that when a CouchDB server fails, another server will take over. If there is only one node that is the job creator, as suggested above, only this node needs to have several instances.

5 Implementation

I implemented all the functionality in Erlang[9]. I choose Erlang because of Erlangs process handling and message passing system. With Erlang, implementing the design was pretty straight forward.

To be able to communicate with a CouchDB server from within Erlang, I used a erlang/otp application called "couchbeam"[2], which is an application that lets me set up a continuous changes stream from within an erlang process as well as retrieve and save documents.

5.1 A first approach

In order to make the system as concurrent as possible, and spread out the points of failures, I decided to separate as much functionality as possible into separate processes.

As a first approach to implement the design, I created four types of processes. A supervisor process, a listener process, a work manager process and a worker process.

The supervisor process, called listener_sup, is the process used to start up the whole system. When listener_sup is started, it creates one listener process and one work manager process. The listener process is given the process identifier, pid, of the work manager so that it can send changes messages to the work manager, as well as needed information about the CouchDB instance (The host address to the local CouchDB instance, as well as the

database name) and the work manager process is given needed information about the CouchDB instance, and the number of workers it should create.

When listener_sup creates the listener and the work manager, it creates a link to them, so that it will receive a message form them if they crash. Upon a crash, the listener_sup simply restarts the failed process.

The listener process, upon creation, creates a continuous changes stream towards a CouchDB database and then waits for messages from the change stream. Upon receiving a change(see section 2.2.2) the listener just forwards the message to work manager process, and then awaits the next change.

The work manager process, upon creation, creates a fixed number of worker processes, and saves their pid and status in a list. The work manager is the supervisor process of the workers, and, just like the listener_sup process, restarts workers that crashed. The work manager waits for messages from the listener and the workers. Upon receiving messages, it handles the messages as described in section 4.2.2.

The worker processes, upon creation, starts a receiver loop, awaiting messages from the work manager. When it receives a message, it creates an external process using the built in function "open_port", and gives the external process the shell command contained in the message from the work manager. It then awaits the result from the external process, which is the integer 0 if the command succeeded, and not 0 if the command failed. The worker will act on the message from the external process as described in section 4.2.2.

The implementation seemed to work, at first. But I missed one very important characteristic in the design about CouchDB changes API. Only when a document is actually is changed, may that be saved or deleted, will the listener receive information about the document. This is all good and well if the work manager always can handle a document. But if the work manager has no free workers, it simply ignores the document, and if this is the case on all nodes, the document will be "forgotten". One approach to solve this could be for the work manager to always save the document, even though no

changes was made to the document, i.e there was no claim or no execution of the step etc. This approach, however would lead to unnecessary updates of a document, since an update to a document actually is a new revision of the document and the old version is kept(see section 2.2.1), and each node that does not have free workers will re-save documents, possible the same documents. Each update is replicated throughout the system, and if the update does not contain any new information, the network communication space is wasted. So I needed a way to minimize the updates on documents for the sole purpose to keep them alive.

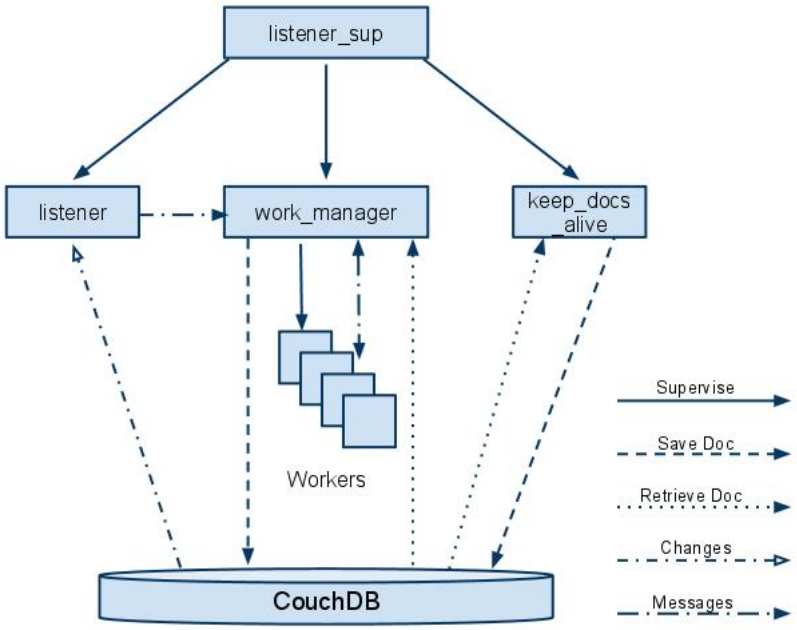


Figure 12: Process layout

5.2 A second, better, approach.

In my second approach, the listener_sup, listener, work manager and worker processes are unchanged. I just added one more process, a "keep docs alive"

process. This process is created and monitored by the listener_sup process.

This process uses CouchDBs view functionality (see section 2.2.3) to periodically retrieve all unclaimed jobs on the nodes local database that the node was the creator of, and then re-saves them, one by one. This way, the work manager will not be responsible for keeping document alive, and documents are only kept alive by their creator node.

Figure 12 shows how the final process layout looks like.

6 Evaluation

To evaluate the implementation, I have set up a network containing one to three nodes, on two different computers but with in the same local network. The nodes all replicate each other. Each node will have 2 workers. The computer with one node is in all test the node where I create the jobs, and is thus the creator of the job, which is an important property since this will be the node that decides a winner when the nodes makes claims on steps (see section 4.2.2). The "keep docs alive" process (see section 5.2) will check for forgotten documents every five second, if nothing else is said.

The job documents contains three steps each, and each step is a five second sleep command. This means each job would, at best, take fifteen seconds the execute.

The way the work manager is implemented, each step will in the best cases, when there is always free workers available, result in two updates to a document for steps with a specific target, and four updates to steps with unspecified target (see section 4.1 and 4.2.2).

When there are specific targets, the work manager first saves the document before executing a step, and after executing a step.

When there is no specific target, the work manager first saves the document in an attempt to claim the step, the a work manager (on the node where the job was created) saves the document when selecting winner. After

that, a work manager saves the document before executing a step, and after executing a step.

Each test evaluation is based on the time taken for a number of jobs to be executed, and the number of revisions that was needed before the job finished (There are, probably, more replications done than the revision value shows. The revision value only shows accepted revisions, not the rejected ones due to conflicts).

6.1 Test 1: 3 nodes, 2 workers each, 1 job, no specific target.

In this test, I used three nodes on two computers. Each node is allowed to execute the job steps. There is just one job, and the time taken for this job would, if there is no cost for replication etc, be 15 seconds.

- Execution time: 15 seconds (15 seconds)
- Revisions: Min: 12, Max: 12

The result shows that the time taken is 15 seconds, and that the total updates made to the document is 12, 4 updates per step. The result reflects the best case for this type of job. This is not surprising, since even though three nodes were involved, only one node actually did the work. The creator node. The reason for this is that the creator node is always the first one to see the changes made to the document, the other nodes will have to wait for the changes to be replicated throughout the network, and thus will always be the node that claims steps first. This leads to the creator nodes claiming the first step, winning the first step, performs the first step, claiming the second step, winning the second step and so forth. Since the steps are executed in strict order, only one worker is used by the creator node, meaning that there was never a lack of workers, and the "keep alive process" did not have to do anything.

6.2 Test 2: 1-3 nodes, 2 workers each, 6 jobs, no specific target.

In the second test, I measure the time it takes for one node to perform 6 jobs, then the same amount of jobs on 2 nodes, and finally on 3 nodes. I have a "keep docs alive" process that re-saves forgotten documents every five second. I then redo the tests with the "keep alive process" set to re-save forgotten document every 1 second. After the execution time, inside the parenthesis, is the optimal execution time[4], when there is no cost for replication, network delays etc.

1-3 nodes, 5 seconds "keep docs alive":

1 node:

- Execution time: 51 seconds (45 seconds)
- Revisions: Min: 12, Max: 20

2 nodes:

- Execution time: 31 seconds (22.5 seconds)
- Revisions: Min: 12, Max: 16

3 nodes:

- Execution time: 17 seconds (15 seconds)
- Revisions: Min: 12, max: 13

1-3 nodes, 1 second "keep doc alive":

[4]The optimal execution time is based on how long 1 job should take, which is 15 seconds, times how many jobs there are in total, divided in the total number of workers available: $\frac{15*jobs}{workers}$

1 node:

- Execution time: 46 seconds (45 seconds)
- Revisions: Min: 12, Max 41

2 nodes:

- Execution time: 31 seconds (22.5 seconds)
- Revisions: Min: 12, Max: 28

3 nodes:

- Execution time: 17 seconds (15 seconds)
- Revisions: Min: 12, Max: 13

The above test shows that adding more nodes will give a performance boost (This is only true up to an extent. If there are more nodes and thus workers than available steps to execute, there will not be any gain on adding more nodes.). Comparing the results from the tests from the 5 second "keep docs alive" and 1 second "keep docs alive" shows that lowering the time the "keep docs alive" process waits between updates will give a better execution time, but a higher revision number due to all the re-saves of documents and each time a document is re-saved, it will be replicated to all nodes resulting in more network strain.

6.3 Test 3, 2 nodes, 2 workers each, 3 jobs, specific targets.

In this test I want test that the system can handle specific targets for the job steps. The test consists of 3 identical jobs. The first step of each job is to be executed on node A, the second on node B, and the third on node A again.

The nodes are located on separated computers, and each has two workers. The "keep docs alive" re-saves documents every five second.

The best possible execution time, presented in the parenthesis next to the test execution time, is the time it would take if there is no replication or network communication costs. Since each job consists of 3 steps, and each step has a target, in the best case a document would only be updated 2 times per step (see section 6).

- Execution time: 23 seconds (20 seconds)

- Revisions: Min 6, Max 8

In almost all of the above tests, the execution time is higher than the best possible execution time. The main factor for this is, in my opinion, not due to the fact that replications between nodes takes time but based on the time between the re-saves of the forgotten documents.

6.4 Test 4, 3 nodes, 2 workers each, 90 jobs, no specific targets.

I wanted to test how well the system could handle a lot of jobs on the same time. So I created 90 jobs, at the same time, and started up the system. The system did handle all the jobs, but my listener process crashed several times during the test. The crashes seems to be caused by some functionally in couchbeam(see section 5) when too many changes are sent from CouchDB to the listener. The test was still a success in some way, because of the my listener_sup process who restarted the listener each time if crashed.

- Execution time: 3 minutes, 45 seconds (4 minutes, 30 seconds)

7 Conclusion and future work

7.1 conclusion

The goal of this thesis was to see if CouchDB could be used as the backbone in a job distributer, and in short, my conclusion is yes, it can.

CouchDB takes care of most of the more difficult issues when working in a distributed environment. Thanks to its replication mechanism it is easy to synchronize databases, and you will have the guarantee that each node will, eventually, have the latest version of a job document.

CouchDBs Multi version Concurrency Control(MVCC) ensures high availability, so that each node can always make reads to the database without the need to wait for the a write to be executed. A feature I took advantage of when having several processes working against a local database; I did not need to figure out when it was safe to make a read. It always is.

CouchDBs RESTful interface makes it easy for the programmer to communicate with the database from any programming language.

The JSON document structure that CouchDB uses makes it easy to alter the job document, adding and removing fields as one sees fit for ones application. CouchDB does not mind, it just replicates without questions.

With all of the above headaches taken care of, the programmer can concentrate on the actual job distributer.

The job distributer I developed successfully shows that it is possible to scale with CouchDB . Just add another node, set up replication and start the job distributer, and that node will start to claim job steps to ease the burden on the other nodes. Note that in the current design, the scalability is up for debate, as seen section 4.3.

In the code that I wrote, the job distributer can only handle shell commands. This, however is just a limitation in my job distributer, and not CouchDB. CouchDB is just a the reliable messenger, what you want in the messages or how you want to handle the content is up the programmer.

One of the biggest downsides of CouchDB is that since it uses MVCC, all old versions of documents are saved, which can lead to the databases growing rather large if not taken care of.

7.2 Future work

There are several areas where the job distributer could be improved. The most important thing to improve is to make sure the job distributer can handle a lot of database changes arriving at the same time without failing. As seen in section 6.4, currently the job distributer can not handle too many changes arriving at the same time.

The job distributer can only handle unix shell commands. Adding different kinds of job action to perform would be useful, and quite easy. In each job step, an additional field can be added which identifies what kind of job it is, be it a shell command or a RESTful command etc, and the worker process will act accordingly.

Adding better failure handling. In the current implementation, the failure handling for job steps is very black or white, a step is successful or it is not and the job is aborted. When only shell commands are used, this kind failure handling could be enough. But if one extends the type of jobs the to include RESTful commands as well, the job distributer should be able to react differently on different kinds of HTTP status codes.

Adding job priorities. As it is now, all jobs have the same priority. There could be that some jobs are more important than other, and that these jobs should be performed before jobs with less priority. When there are sufficient workers to handle all jobs concurrently, this is not a problem, but when there are no free workers available, it could come in use to have priorities. The priorities could just be a integer value in each job document, and the "keep docs alive" processes could then use that value to decide in which order it should re-save documents.

In contrast to the problem description, let some job steps be performed

on multi nodes. In my design, I wanted to guarantee that a job step is only performed once, but there should be possible to have the option to create jobs with steps that should be performed on multiple nodes.

Minimize the amount of replications. Each time a job document is changed and saved, it will be replicated throughout the system. But there are situations when not all nodes needs to get the updated document. When, for example, there is a specific target for a job step, there is no need for unaffected nodes to get the document replicated to its local database. Another example is when a node claims a step. The only node who is interested in the claim is the node that will select the winning node. To solve this, some kind of filters should be implemented, making sure no unnecessary replication is made.

Control the job distributer. In its current state, it is not possible to control the job distributer after it has been started. If you want to add or remove workers, you need to restart the job distributer. It should be possible to control the job distributer during run time, through a RESTful interface for example; adding and removing workers; change the wait time for the "keep docs alive" process; change the database; get information about memory and CPU usage etc.

References

[1] Anderson, J., Lehnardt, J. & Slater, N. (2010) *CouchDB: The Definitive Guide,* O'Reilly Press

[2] Chesneau, B. (2011) *Couchbeam,* http://benoitc.github.com/couchbeam, Accessed 2011-06-15

[3] Fielding, R. (2000) *Architectural Styles and the Design of Network-based Software Architectures,* chapter 5

[4] Gilbert, S., Lynch, N. (2002) *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services.* ACM SIGACT News 33(2)

[5] Lennon, J. (2009) *Beginning CouchDB,* Apress

[6] Kshemkalyani, A. and Singhal, B. (2008) *Distributed Computing Principles, Algorithms, and Systems*, Cambridge University Press

[7] Vogels, W. (2008) *Eventually Consistent - Revisited*, http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, Accessed 2011-06-15

[8] Curl, http://curl.haxx.se/, Accessed 2011-06-15

[9] Erlang Programming Language, http://www.erlang.org, Accessed 2011-06-15

[10] JSON-RFC, http://tools.ietf.org/html/rfc4627, Accessed 2011-06-15

[11] MD5-RFC, http://tools.ietf.org/html/rfc1321, Accessed 2011-06-15

[12] URI-RFC, http://tools.ietf.org/html/rfc3986, Accessed 2011-06-15