UNIVERSITÀ DEGLI STUDI DI TRENTO

Distributed Systems
A.A. 2011-2012

Data consistency properties in Amazon SimpleDB and Amazon S3

Davide Gerhard, Davide Girardi

davide.gerhard@unitn.it
davide.girardi-2@unitn.it

Abstract

This report describes the experiments performed as a project for the Distributed Systems course during the 2011-2012 academic year. During our work, we focused on the Amazon SimpleDB distributed database and Amazon Simple Storage Service in order to analyze their consistency properties, in our case, the read your write consistency and the monotonic write consistency.

Contents

1 Introduction

The new services offered by the commercial cloud storage systems are attracting a lot of interest nowadays for their ability to offer scalability, availability and durability at a low cost. The other side of the medal of these systems is usually their weak consistency properties, considering the theorems behind distributed systems, such as the CAP theorem. During our work, we analyzed two of these services, Amazon SimpleDB and Amazon S3, from a consumer perspective in order to understand which kinds of consistencies they really offer and if they respect what is promised. As a starting point, we focused on the research done by Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee and Anna Liu, described in [2], trying to reproduce their results.
We will now make a survey of the theory behind NoSQL databases, Amazon SimpleDB and Amazon S3. We will then show how we made our analysis in the 2nd chapter. The description of how we implemented our benchmarks, all the results and our reflection on them will continue in the following 3rd and 4th chapters.

1.1 Not only SQL databases

The *Not only SQL database* (NoSQL), is a class of database management system (DBMS) that are designed to achieve high throughput and high availability at the cost of not being able to offer the other usual services of a normal DBMS, such as joins and ACID[1] transactions. If most of these systems provide weak consistency features, then it is possible to observe stale data returned by a query. This is mainly due to the CAP theorem, which states that it is impossible for a distributed computer system to offer all three of the following properties at the same time

- **Consistency:** all nodes see the same data at the same time.
- **Availability:** a guarantee that every request receives a response about whether it was successful or failed.
- **Partition tolerance:** the system continues to operate despite arbitrary message loss.

Since their principal focuses are availability and partition tolerance, they usually relax the data consistency constraints.

[1]Atomicity, Consistency, Isolation, Durability; a set of properties guaranteed by the usual DBMS's transactions

1.2 Amazon SimpleDB

Amazon SimpleDB[2] is a distributed database written in Erlang by Amazon. It is a part of the Amazon Web Services[3] and it is also a NoSQL database. The data is organized into domains which are collections of items described by attribute-value pairs. The system ensures high availability and partition tolerance storing multiple geographically distributed copies of each domain. Regarding its consistency options, there are two possible read options:

- **Eventual consistent read**
  If no new updates are injected after some time t, eventually all correct nodes will obtain the same copy of the database. This should guarantee better performances.

- **Consistent read**
  The consistent read option ensures that the data returned always comes from the most recent write operation

1.3 Amazon S3

Amazon Simple Storage Service (S3)[4] is another part of the Web Services offered by Amazon. It is an online storage system which is supposed to offer scalability, high availability, and low latency. The data unit is called object and all objects are organized into one or different buckets and identified by a key assigned by the user. S3 is designed to provide 99.999999999% durability and 99.99% availability of objects over a given year. It has two possible write options:

- **Standard redundancy**

- **Reduced redundancy**

The standard redundancy option guarantees that the probability of durability of an object is at least 99.999999999% while the other one aims to give at least 99.99% probability of durability. The documentation online[5] provided by Amazon states that choosing the EU (Ireland) Region the system offers read-after-write consistency for PUTS of new objects and eventual consistency for overwrite PUTS and DELETES.

[2]http://aws.amazon.com/simpledb/
[3]http://aws.amazon.com/documentation/
[4]http://aws.amazon.com/s3/
[5]http://aws.typepad.com/aws/2009/12/aws-importexport-goes-global.html

1.4 Read your writes consistency

A relevant consistency property that could be important for a consumer who is using one of these systems is the *read-your-writes consistency*. It states that when the most recent write is from the same thread as the reader, then the value should be fresh.

1.5 Monotonic write consistency

Another important consistency property is the *monotonic write consistency*. If this property is guaranteed by the system, the writes of a process are serialized.

2 Experimental Analysis

Our investigation was performed by executing a benchmark program and doing the same tests for both Amazon SimpleDB and Amazon S3. Every tests was run trough the cloud computing service Amazon EC2[6] to minimize the latency between our benchmark process and the data store. In addition we always used a the region EU-Ireland for all tests.
The base architecture of our benchmark program was the same described in [2] and shown below, but in our case, the writer and the reader process are executed only by a single thread.

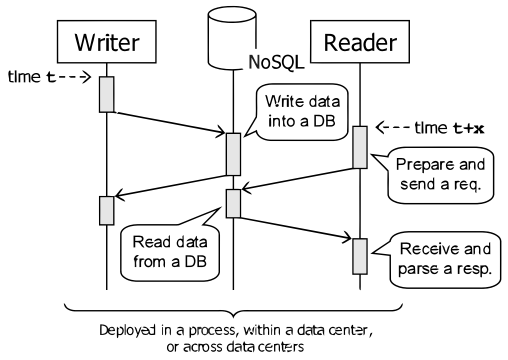

Figure 1: Architecture of the benchmark application. The use of a noSQL database is just an example.

[6] http://aws.amazon.com/ec2/

We will now describe how we performed our analysis of the consistency properties offered by Amazon simpleDB and Amazon S3.

2.1 Read your write consistency

To evaluate the *read your write consistency*, we proceeded following these steps (all steps were executed by a single thread):

1. a writer process writes the current timestamp on a data element on simpleDB or S3.
2. a reader process repeatedly reads the data contained in the data element. We save at which time the read is occurred and if we read the last written value or not.
3. We now calculate the probability of reading the freshest value after a specific time interval elapsed from when the write begins.

We ran these test several times either using the eventual consistent read or the consistent read for Amazon SimpleDB and using different redundancy options for Amazon S3. To calculate the probability, we considered it as a frequency, in particular the number of occurrences of a positive read (it reads the freshest value) per time interval (in our case $1ms$). Therefore, we used the following formula,

$$P_{pr}(t_0) = \frac{N_p}{N_r},$$

where P_{pr} is the probability that our read was positive, N_p is the number of positive reads in the interval t_0 and N_r is the total number of reads in the same interval.

2.2 Monotonic write consistency

To evaluate the *monotonic write consistency*, we proceeded following a procedure similar to what we showed before for the read your write consistency. The process follows these steps (again, all steps were executed by a single thread):

1. a writer process writes the current timestamp (call this vale v_0) on a data element on simpleDB or S3, we wait for 1 second to be sure that the data has been propagated and then it updates this value two times in a row (call these values v_1 and v_2).
2. a reader process read repeatedly the data contained in the data element. We save at which time the read is occurred and if we read v_0,v_1 or v_2.

3. We now calculate the probability of reading a value after a specific time interval elapsed from when the write begins, for each value.

We ran these test several times either using the eventual consistent read or the consistent read for Amazon SimpleDB and using different redundancy options for Amazon S3, as we did for the read your write consistency. To calculate the probability, we used the formula shown above for each of the three values v_0,v_1 or v_2.

3 Benchmark implementation

Our benchmark program was developed using the Java programming language. The main reason behind our choice is that this language is used in the reference [2] and is one of those best supported by Amazon. Indeed, they offer good API's which can be used by a developer to communicate with Amazon simpleDB or S3. The version used by us is the 1.2.15 [5]. We will now show some relevant portions of code to describe how we implemented the procedures summarized in the previous section.

3.1 Amazon SimpleDB's benchmark implementation

Amazon provides some methods to interact with a database on simpleDB. In particular we used `createDomain`,`putAttributes`,`getAttributes`[5] for, respectively, create a domain, put an attribute on a database or get an attribute from a database.
We developed two methods to perfom the read your writes and monotonic write consistency tests. The source code of these methods is described in the following two code listings.

Read your write consistency test implementation

```
    /**
     * read your write consistency test
     */
    private static void readYourWrite() {
        // define the array with results
        // tArray[count][0]=currentTimestamp
        // tArray[count][1]=1/0 ; 1 = equal to baseTimestamp, 0 ow.
        long tArray[][] = new long[readLoop][2];

        // [A][B]:
        // A = millisecond
        // B = 1: true(1) if good read; 0: total reading in A millisecond
        int probArray[][] = new int[maxMS][2];
```

```java
        initializeArray(probArray);
        long baseTimestamp;
        int times, count;
        Random casual = new Random();

        // run test N times
        for (times = 0; times < testTimes; times++) {
            E_INFOQ("test number: "+times);
            // write the timestamp into simpledb
            if (writeTimestamp(baseTimestamp=System.nanoTime())) { System.
                exit(1); }
            // read the timestamp M times
            for (count = 0; count < readLoop; count++) {
                // save the diff from nanoTime and the starting time
                // tArray[count][0] = System.nanoTime()-baseTimestamp;
                // if it is equal to the starting time put 1 to tArray
                if (readTimestamp(readConsistency) == baseTimestamp)
                    tArray[count][1]=1;
                // else write 0
                else tArray[count][1] = 0;
                tArray[count][0] = System.nanoTime()-baseTimestamp;
            }
            // don't save the first round - high result
            if (times > 0 )
                // write the result to a file
                if(writeResult(tArray,times,"readYourWrite")) E_ERR("error
                     while saving the data");
                calculateProbability(tArray,probArray);
        }
        if(writeProbability(probArray,resultFile))  E_ERR("error while
            saving the result");
    }
```

Monotonic write consistency test implementation

```java
    /**
     * monotonic write consistency test
     */
    private static void monotonicWriteConsistency() {
        final int baseInt = 0;
        // define the array with results
        // tArray[count][0]=currentTimestamp
        // tArray[count][1]=1/0 ; 1 = equal to baseTimestamp, 0 ow.
        long tArray0[][] = new long[readLoop][2];
        long tArray1[][] = new long[readLoop][2];
        long tArray2[][] = new long[readLoop][2];

        // [A][B]:
        // A = millisecond
        // B = 1: true(1) if good read; 0: total reading in A millisecond
        int probArray0[][] = new int[maxMS][2];
        int probArray1[][] = new int[maxMS][2];
        int probArray2[][] = new int[maxMS][2];
        long baseTimestamp, baseTimestamp1, baseTimestamp2, readData, time
            ;
        int times, count;

        initializeArray(probArray0);
        initializeArray(probArray1);
        initializeArray(probArray2);

        // run test N times
        for (times = 0; times < testTimes; times++) {
            E_INFOQ("test number: "+times);

            //write the v0 value
            if (writeTimestamp(baseInt)) {
                E_ERR("error on writing the timestamp"); System.exit(1);
            }
            // sleep for 1 seconds = stabilize old value = V0
            try { Thread.sleep(1000); } catch (InterruptedException i) {
                E_ERR("sleep error"); }

            baseTimestamp = System.nanoTime();

            // write the timestamp into simpledb
            if (writeTimestamp(baseTimestamp1 = System.nanoTime())) {
                E_ERR("error on writing the timestamp"); System.exit(1);
            }

            // get second nanoTime
            if (writeTimestamp(baseTimestamp2 = System.nanoTime())) {
                E_ERR("error on writing the timestamp"); System.exit(1);
            }

            // read the timestamp M times
            for (count = 0; count < readLoop; count++) {
                time = System.nanoTime()-baseTimestamp;
                readData = readTimestamp(readConsistency);

                tArray0[count][0] = time;
                tArray1[count][0] = time;
                tArray2[count][0] = time;
                if ( readData == baseInt ) tArray0[count][1] = 1;
                else tArray0[count][1]=0;
                if ( readData == baseTimestamp1 ) tArray1[count][1] = 1;
                else tArray1[count][1] = 0;
                if ( readData == baseTimestamp2 ) tArray2[count][1] = 1;
                else tArray2[count][1] = 0;
            }

            // don't save the first round - high result
            if (times > 0 )
            // write the result to file
            if(writeResult(tArray0,times,"monitonicWriteConsistencyV0"))
                E_ERR("error during save the data");
            if(writeResult(tArray1,times,"monitonicWriteConsistencyV1"))
                E_ERR("error during save the data");
            if(writeResult(tArray2,times,"monitonicWriteConsistencyV2"))
                E_ERR("error during save the data");
            calculateProbability(tArray0,probArray0);
            calculateProbability(tArray1,probArray1);
            calculateProbability(tArray2,probArray2);
        }

        if(writeProbability(probArray0,"resulV0.dat"))  E_ERR("error
            during save the result");
        if(writeProbability(probArray1,"resulV1.dat"))  E_ERR("error
            during save the result");
        if(writeProbability(probArray2,"resulV2.dat"))  E_ERR("error
            during save the result");
        if(writeProbabilityMono(probArray0,probArray1,probArray2,
            resultFile)) E_ERR("error during save the result");
    }
```

3.2 Amazon S3's benchmark implementation

The implementation of the benchmark for S3 is heavily based on the simpleDB's one. The methods for the read your write and monotonic write consistency tests are the same shown above. We used different methods to communicate with the system, in particular, `createBucket`,`putObject`,`getObject`[5].

4 Results

To show our results, we aggregated our data and built some graphics which evince the probability of reading the freshest or the right value against time.

4.1 Amazon SimpleDB

4.1.1 Read your write consistency

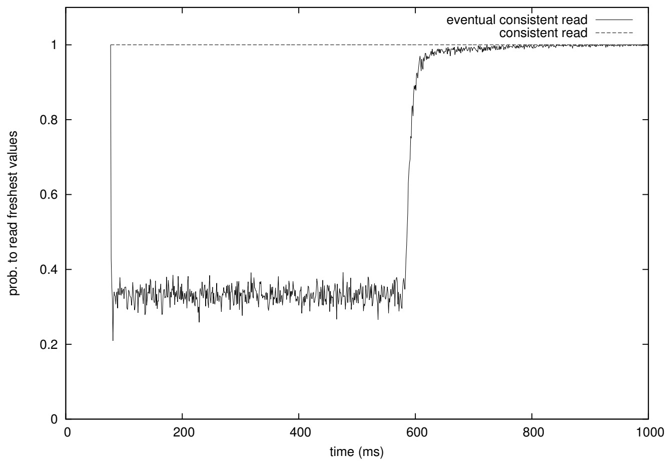

Figure 2: Results of the read your write consistency tests performed on SimpleDB

This graphic shows the probability of reading the fresh value against the time interval that elapsed from when the write begins to the time when the read is submitted.

4.1.2 Monotonic write consistency

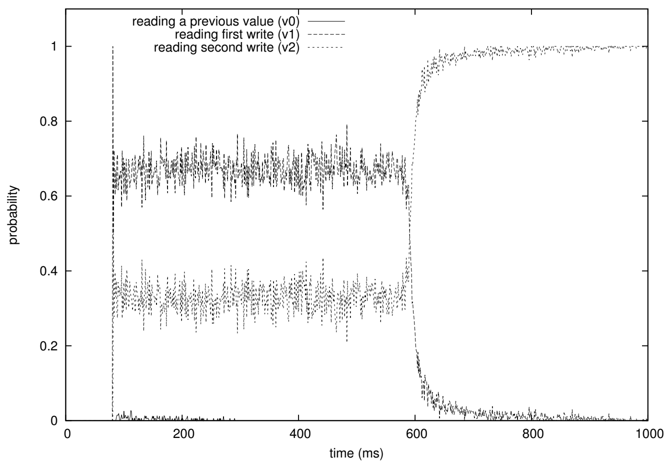

Figure 3: Results of the monotonic write consistency tests performed on SimpleDB

In this case, the graphic shows the probability of reading v_0,v_1,v_2 against the time from the start of the cycle.

4.2 Amazon S3

We will not show any graphics related to our experiments on S3. Even if performing the monotonic write consistency test, we should get results similar to what we have seen for SimpleDB, we never found stale data in all our tests.

4.3 Comparison with the original paper

As we can notice from the graphics, the results derived from our experiments are really similar to those described in [2]. The only difference is that our graphics are shifted by 100ms. This is mainly due to the latency of the read/write operations and it has been probably omitted by the authors because it is not so meaningful for our purposes.

5 Conclusions

Based on our results, we can state that from the client view of the system S3 seems consistent regardless of what Amazon states to provide and all the redundancy options. On the other hand, all the tests performed on simpleDB confirmed the theory (the CAP theorem and noSQL database structure) in case of eventual consistent reads.
Consider now the following figure, as probable architecture of both systems.

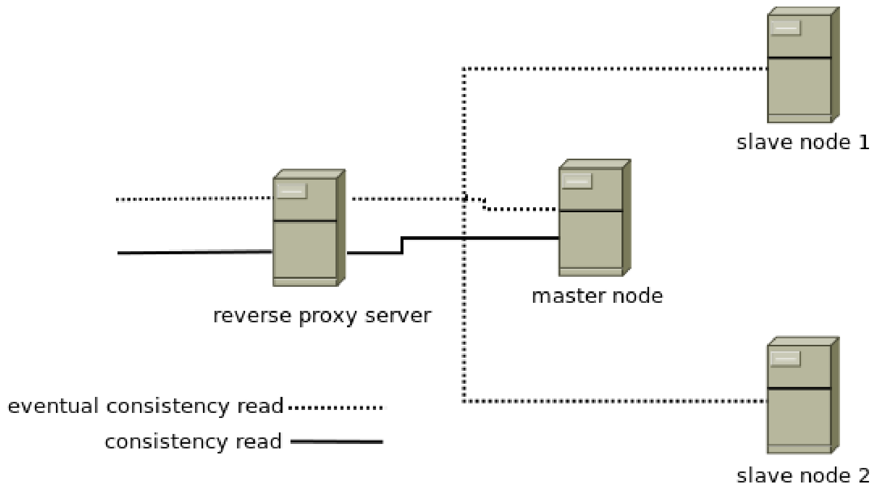

Figure 4: Probable architecture of Amazon SimpleDB and Amazon S3

The reverse proxy accepts the request of a client and translates it to a request for the actual distributed database/datastore which could be composed by a master and slave nodes. We made some hypotheses based on this architecture.
During a write operation the element is firstly written on the master node and then propagated on all other slaves (passive replication). In the simpleDB's case, the element is fully upgraded on all nodes after about 500ms or after each new update. If we want to get an element from simpleDB, it is retrieved from the master node if we chose the consistent read, or from any node if we chose the eventual consistent read, as if the system would be based on the active replication scheme. In the S3's case, an element is propagated on the slave nodes when it is overwritten and retrieved from any node.

References

[1] **H. Wada, A. Fekete, L. Zhao, K. Lee, A. Liu**, *Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers Perspective* , 5th Biennial Conf. on Innovative Data Systems Research (CIDR), pages 134143, Asilomar, CA, USA, January 2011.

[2] **Werner Vogels**, *Eventually consistent*, Comm. of the ACM, 52(1):4044, 2009.

[3] **Amazon.com**, *Amazon Simple Storage Service Developer Guide*, API version 2006-03-2001.

[4] **Amazon.com**, *Amazon SimpleDB Developer Guide*, API version 2006-03-2001.

[5] **Amazon.com**, *Amazon AWS Java API v. 1.2.15*.