Replicated Data Consistency Schemes: Implementation and Evaluation

Submitted By: Trivikram Bollempalli
Mentor: Ivo Jimenez
Professor: Carlos Maltzahn

Abstract:

Replication of data is done in distributed systems to increase availability and fault tolerance. Availability and Consistency are the two properties that are desirable from any data replication scheme. Most of the systems offer strong consistency and eventual consistency schemes, but there are other consistency schemes that can be achieved with a trade off between performance and availability. In this paper, we discuss about other consistency schemes, algorithms to implement them and performance results of these algorithms and try to compare the different consistency schemes.

1.Introduction:

Most of the distributed systems like windows Azure implemented strong consistency scheme. In recent times eventual consistency is also implemented on some systems like Dynamo. Both of these systems are two extreme points in terms of availability and consistency. Strong consistency offers full consistency to applications which makes its performance slow whereas eventual consistency scheme guarantees no consistency but it performs better in terms of time taken for a read or write operation. Related studies show that other consistency schemes are possible and we can achieve different trade offs between consistency and availability other than those offered by strong and eventual consistency. In particular, the paper written by Doug Terry, 'Replicated Data Consistency Schemes Explained Through Baseball' explains four other consistency schemes and explains through a baseball game that those schemes can be useful for some applications. In this paper, we introduce the different consistency schemes explained in [1], discusses the algorithms to implement them and their results.

2.Different Consistency Schemes:

We discuss six different consistency schemes from the perspective of read operations. There can be many readers and writers. The readers may choose different consistency scheme based on their needs of consistency, performance and availability.

Strong Consistency:

In strong consistency, the reader expects to see the effect of all the writes that happened. If a request to an object is made, the returned object should contain the effect of all writes in the order they happened until last modified.

Eventual Consistency:

In eventual consistency, the reader expects only one guarantee that it reads fully consistent data eventually. The read object may contain any of the effects due to the subset of write operations.

Bounded staleness:

In Bounded Staleness, the read expects that the values read are not too old but

need not be latest. The reader keeps a bound on the level of staleness it can accept. This staleness can be defined using time or number of write operations. For example, 'the read should see the effect of all writes happened before 1 minute' or 'the read should see the effect of all writes happened before last 'k' write operations'.

Monotonic Reads:

In Monotonic reads, the readers requires that whatever value read is later than or equal to the last read performed by the reader. The read data can be a stale value but it should be latest compared to the previous read.

Read My Writes:

In Read My Writes, the reader expects that the read value should see the effect of all previous write operations performed by it. If the client doesn't perform any writes, then this scheme is equivalent to eventual consistency. If the client is the only writer, it is easy to observe that it is equal to strong consistency scheme.

Consistent Prefix:

In consistent prefix, the reader does not need the latest value of an object, but whatever value it reads should contain the effect of ordered sequence of writes happened till that point of time.

The usefulness of these consistency schemes is not explained here. More details about these schemes can be referred from [1].

3 Architecture

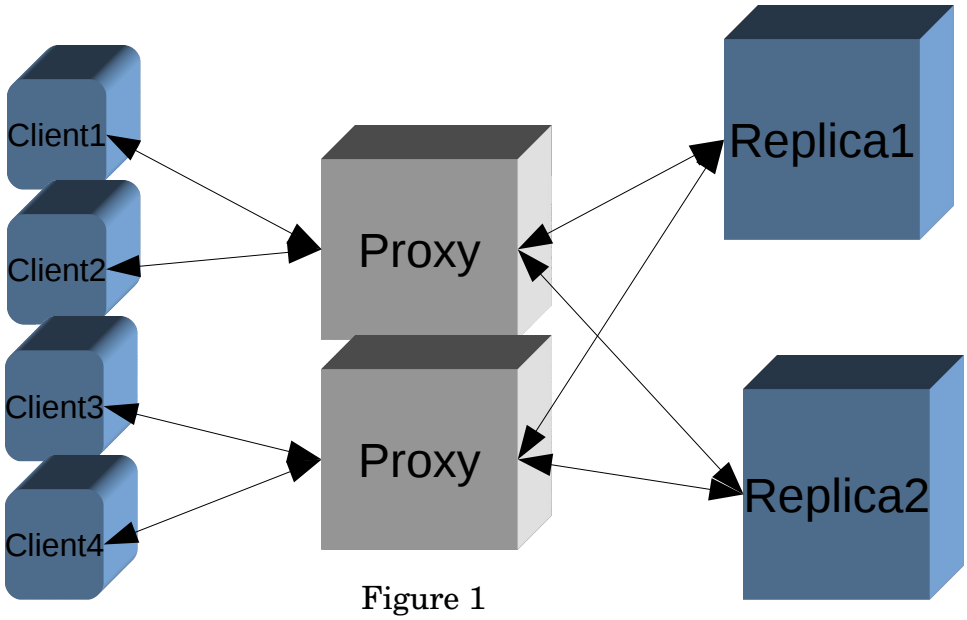


Figure 1

The schemes mentioned in section 2 are implemented on a system and evaluated. The system contains three components. Clients, Proxy and Replicas as shown in figure 1.

The clients can perform read and write operations. Clients can send a write request to store (key, value) pair to proxy. The proxy acquires locks for the key from different replicas and proceed with the write. Similarly, clients can send a read request for a key to proxy and the proxy gets the value of the key from the replicas from possible set of values satisfying the constraint imposed by the client.

4. Algorithms For Different Schemes:

This section deals with algorithms used to implement different read consistency schemes on the system shown in figure 1. Each component is multi-threaded.

Design:

All algorithms are designed by maintaining that the proxy does not store any information to perform future operations. All information about the operations, if needed, is stored in replicas. The proxy function is to only maintain the constraints required by a scheme. This condition helps us to design the algorithms suitable for real systems without a proxy. This condition is further discussed while talking about the algorithms for different schemes.

Write Operations:

For the write operations, we use strong consistency scheme. When a client thread submits a write request for a key, proxy acquires lock for that key from all replicas, performs the write operation on all replicas and then releases the lock. The following is the pseudo code in proxy for write operations.

```
Write( key, value){

    for all replicas{
        obtainLock();
    }

    for all replicas{
        performWrite(key, value);}
    }

    for all replicas{
        releaseLock();
    }
}
```

Pseudo code for write operations

Strong Consistency Read:

For strong consistency read, when a client thread submits a read operation for a

key, proxy accepts the request, acquires lock from all the replicas, perform the read operation on any replica and releases the lock. The following is the pseudo code in proxy for strong consistency read operation.

```
strongRead(key){

    for all replicas{
       obtainLock();
    }

    value = replica[random].performRead(key);

    for all replicas{
       releaseLock();
    }
    return value;
}
```

Pseudo code for strong consistency read operations

Eventual Consistency Read:

For eventual consistency read, when a client thread submits a read operation for a key, proxy accepts the request, submits a read request to a random replica without acquiring key and receives a response from the replica. The response is returned to the client. The following is the pseudo code in proxy for eventual consistency read operation.

```
eventualRead(key){
        return replica[random].performRead(key);
}
```

Pseudo code for eventual consistency read operations

Monotonic Read:

For Monotonic reads, when a client thread submits a read operation for a key, proxy accepts the request. Proxy has to determine what is the timestamp of the last read performed by the client.

If Proxy is allowed to store, then a session information is maintained at proxy, and timestamp of last transaction can be obtained which is easy to implement. Then the proxy can choose a random replica, sends a request and check whether that replica satisfies the constrains.

Since proxy is not allowed to store, we can't do that anymore. If a proxy is not there, then the replicas has to communicate the session information stored in them, and then identify the latest read operation performed by the client and then the replica which has data later than that will respond to the request. The proxy should also do the same thing. The proxy collects the last timestamp of the read performed by the client for that

key,from the replicas and also the latest timestamp of the key being requested. The proxy then determines the timestamp of the last read performed and sends a request randomly to the replica selected randomly from the replicas whose timestamp is later than the last read's timestamp. Following is the pseudo code.

```
MonotonicRead(key){
        for all replicas{
                // collects timestamps of last reads and latest data for key
                ts[replica][] = getTimeStamps();

                //last reads performed by client on replica
                sessionTS[replica] = ts[replica][0];

                //latest timestamp of requested key on replica
                latestDataTS[replica] =  ts[replica][1];
        }

        // compares the last read performed at each replica and find the maximum
        lastRead = findMaxSessionTS(sessionTS);

        while(true){
                i = random number between (0, replica.length -1);
                if(latestdataTS[i] >= lastRead){
                        break;
                }
        }
        replica[i].performRead();
}
```

Pseudo code for Monotonic reads

Bounded Staleness:

The proxy should also do the same thing. The proxy collects the last timestamp of the read performed by the client for that key,from the replicas and also the latest timestamp of the key being requested. The proxy then determines the timestamp of the last read performed and sends a request randomly to the replica selected randomly from the replicas whose timestamp is later than the last read's timestamp. Following is the pseudo code.

For Bounded stale reads, when a client sends a read request to the proxy, proxy accepts the request. It gets the latests timestamp for that key from the replicas and compares the timestamps and obtain the latest timestamp for that key. It randomly chooses a replica and compare the latest timestamp obtained with the replicas timestamp for the required key. If the difference between them is below certain threshold 'k', then a read operation is performed on that replica. Pseudo code is given below.

```
boundedStaleRead(key){
        for all replicas{
// collects timestamps of  latest data for key (gives no. of operations performed on key)
                sessionTS[replicaNo] = getTimeStamps();
        }

        totalOperations = findMaxSessionTS(sessionTS);

        while(true){
                i = random number between (0, replica.length -1);
                if(totalOperations – sessionTS[replicaNo] < 'k'){
                        break;
                }
        }
        replica[i].performRead();
}
```

Pseudo code for Bounded stale read

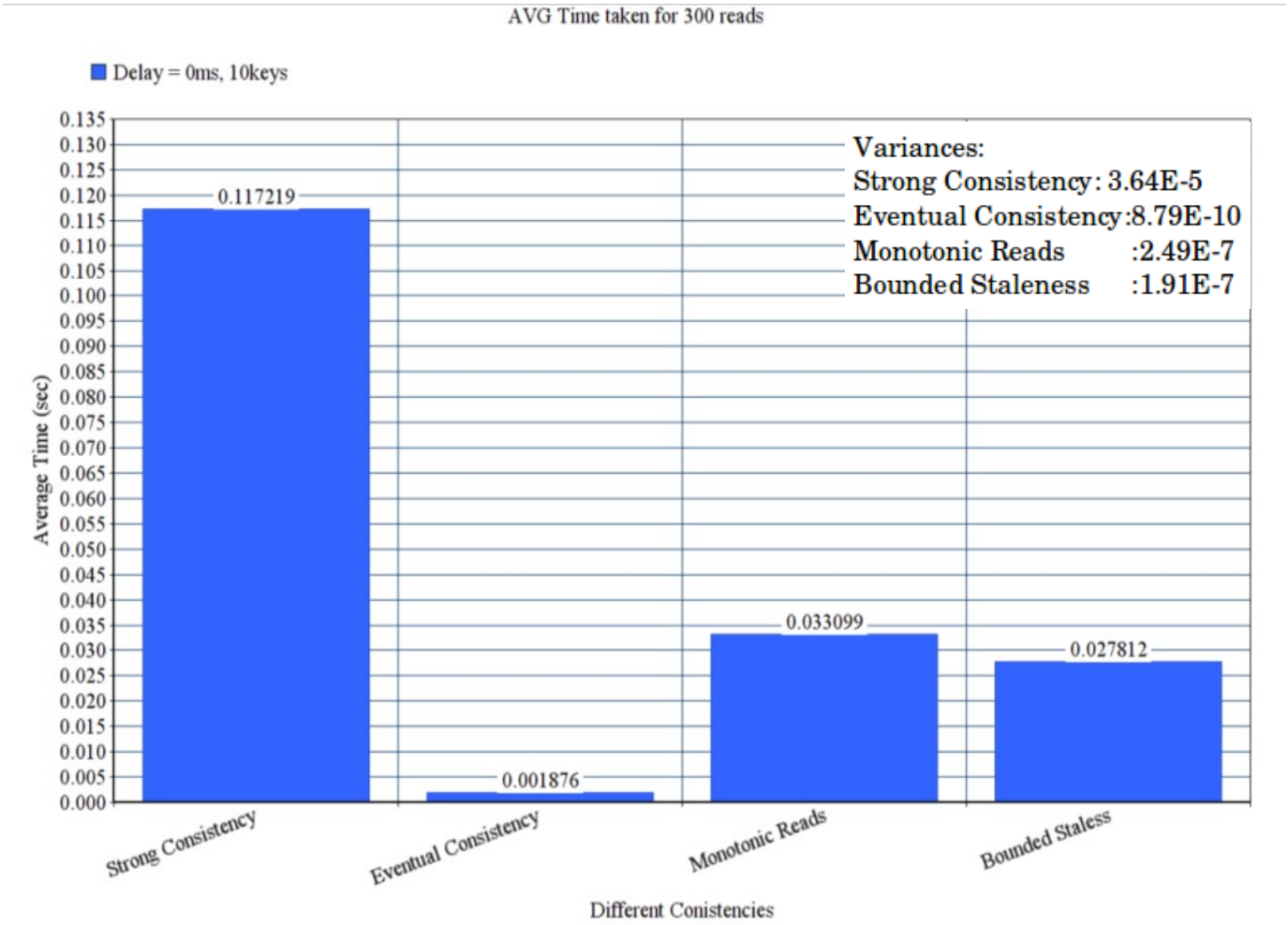
Other Approaches:

There can be other approaches to the algorithms. In this paper we considered the writes to be strongly consistent. We can assume writes to be eventually consistent and can implement different read consistency schemes. For example, amazon's shopping cart uses eventually consistent writes and strong reads. In this case, strong reads take more time as they have to go to every replica and need to merge them. Monotonic and bounded reads also go to every replica, but need not merge them. They need to perform the same computation discussed in the algorithms in this report.

5. Evaluation

The above algorithms are implemented in a system and evaluated for performance. Total number of replicas assumed for the experiment is 2 for all the experiments. The values shown in all the experiments are for *300 read operations*. The results shown are the average values taken from iterating the experiment for 10 times.

For the experiment shown in Graph 1 below, a single writer thread, submits write requests to the proxy. The write requests are <key, value> pairs. Total number of keys taken are 10, ranging from 0 to 9. A reader thread performs read operations on the keys by submitting a read request to the proxy. No time delay is induced between proxy and replicas. The variance for strong consistency is higher than other schemes because it gets affected by the distribution of write requests. If the lock key k on which read operation is requested is being hold by a write operation, then more time will be taken for that operation. It depends on which key is chosen for read and write operations.

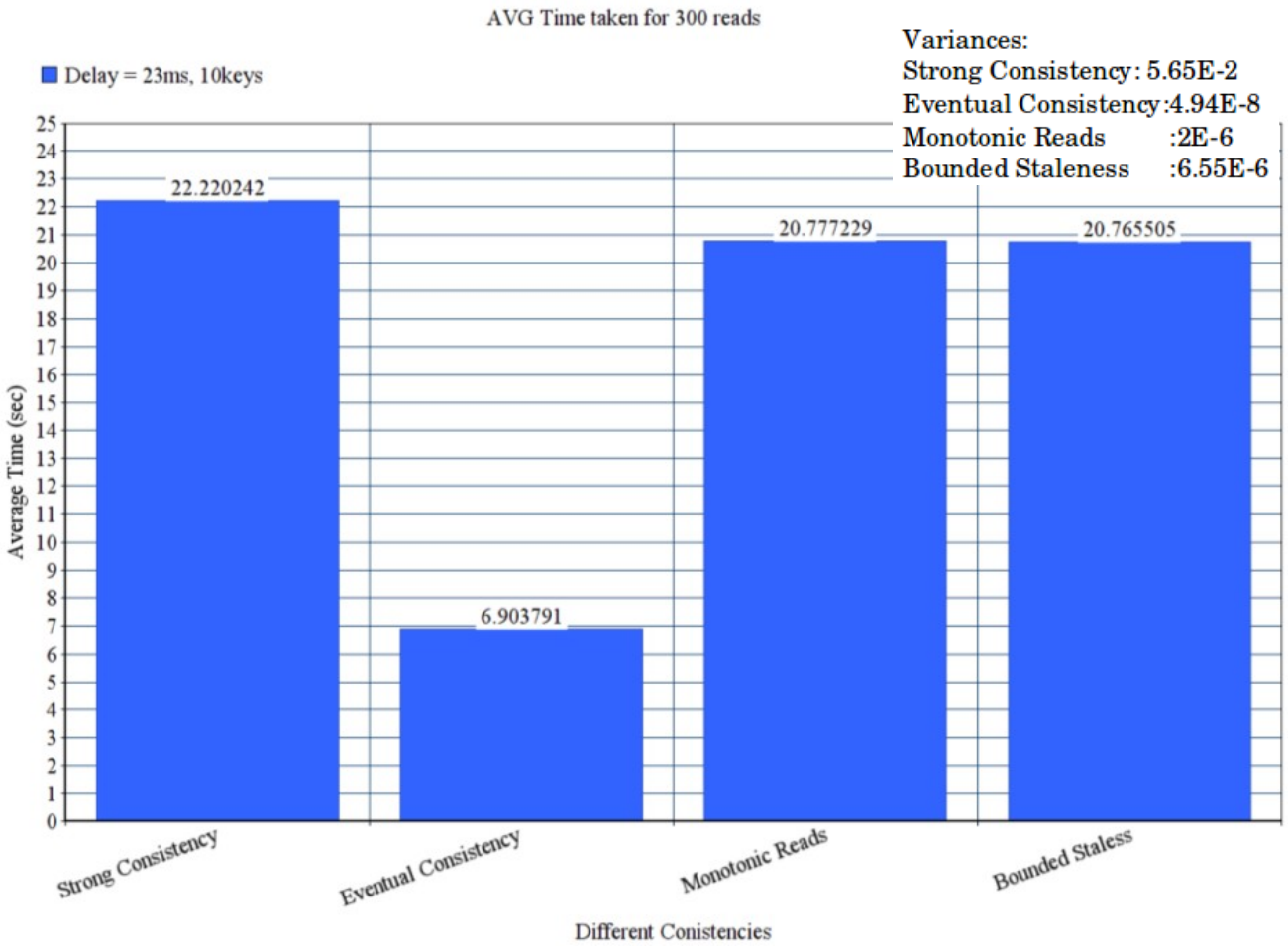


Graph 1

Strong consistency takes more time than all other approaches and Eventual consistency takes the lowest time as expected. Monotonic and Bounded stale reads are almost equal. Since no delay is induced, there is no network latency. The time taken for

strong consistency is more because it has to wait for the lock from both the replicas. Monotonic and Bounded stale reads are higher than Eventual Consistency reads, since as seen in algorithms, for eventual consistency, proxy need to access only one of the replicas whereas for other replicas, all replicas need to accessed and compared.

The setup for the experiment in graph 2 is same as previous experiment, except that a latency of 23ms is induced. Variance is more for strong consistency as explained for the previous graph. Important thing to observe is monotonic and bounded stale reads are close to strong consistency reads. This is because the majority of latency is due to network latency. The probability of a read and write operation occurring at same time for 10 different keys is relatively low. This probability is same for previous graph also but as there is no network latency, the difference appeared high for the previous graph. This point is proved by our next experiments with 2 keys.

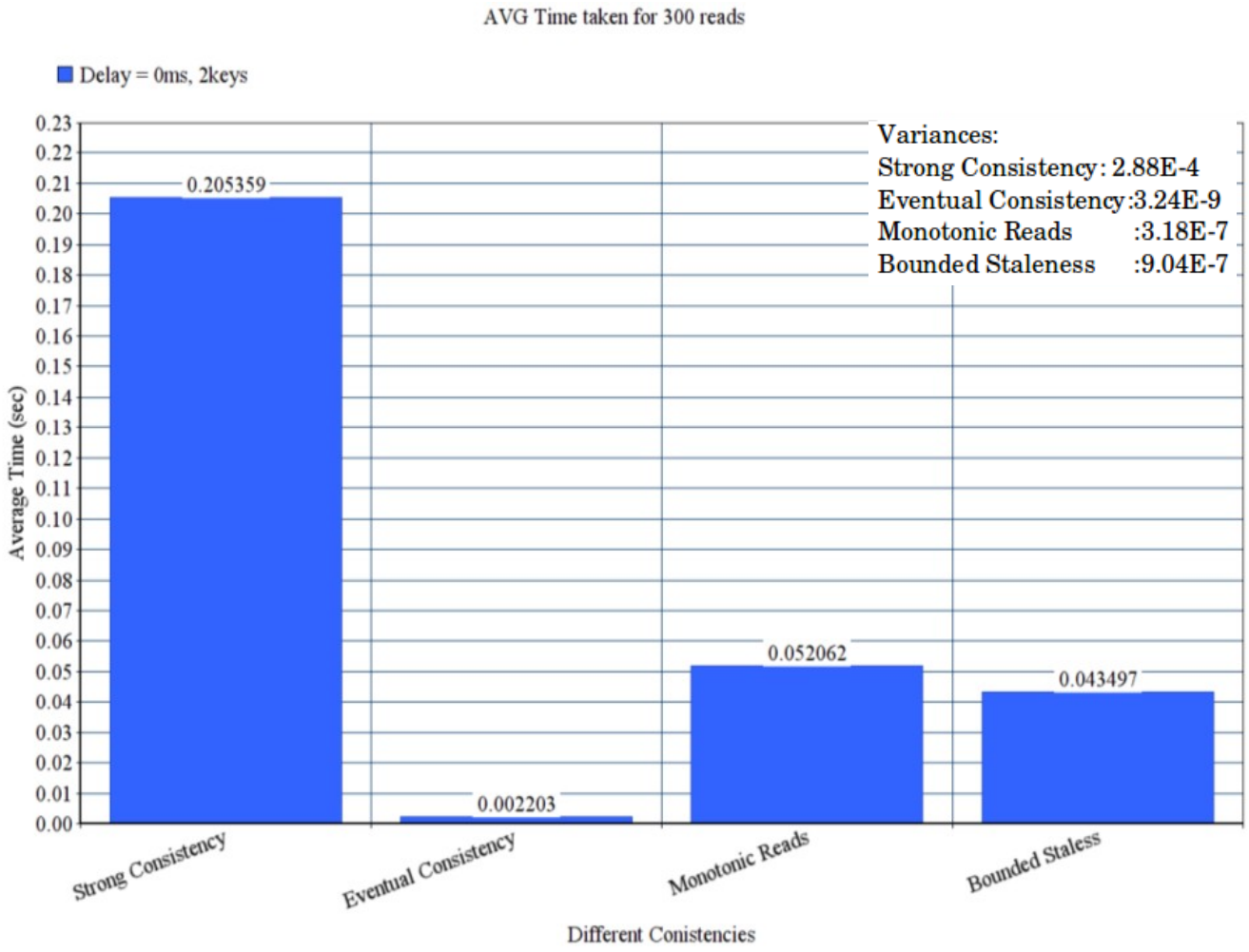


Graph 2

The other results are obvious with eventual consistency taking the least time. It suffers less from the induced delay as the proxy approaches a replica only once. In other schemes, proxy communicates with all the replicas, for getting locks in case of strong consistency and for getting sessionInfo, in case of other schemes. After this, proxy has to

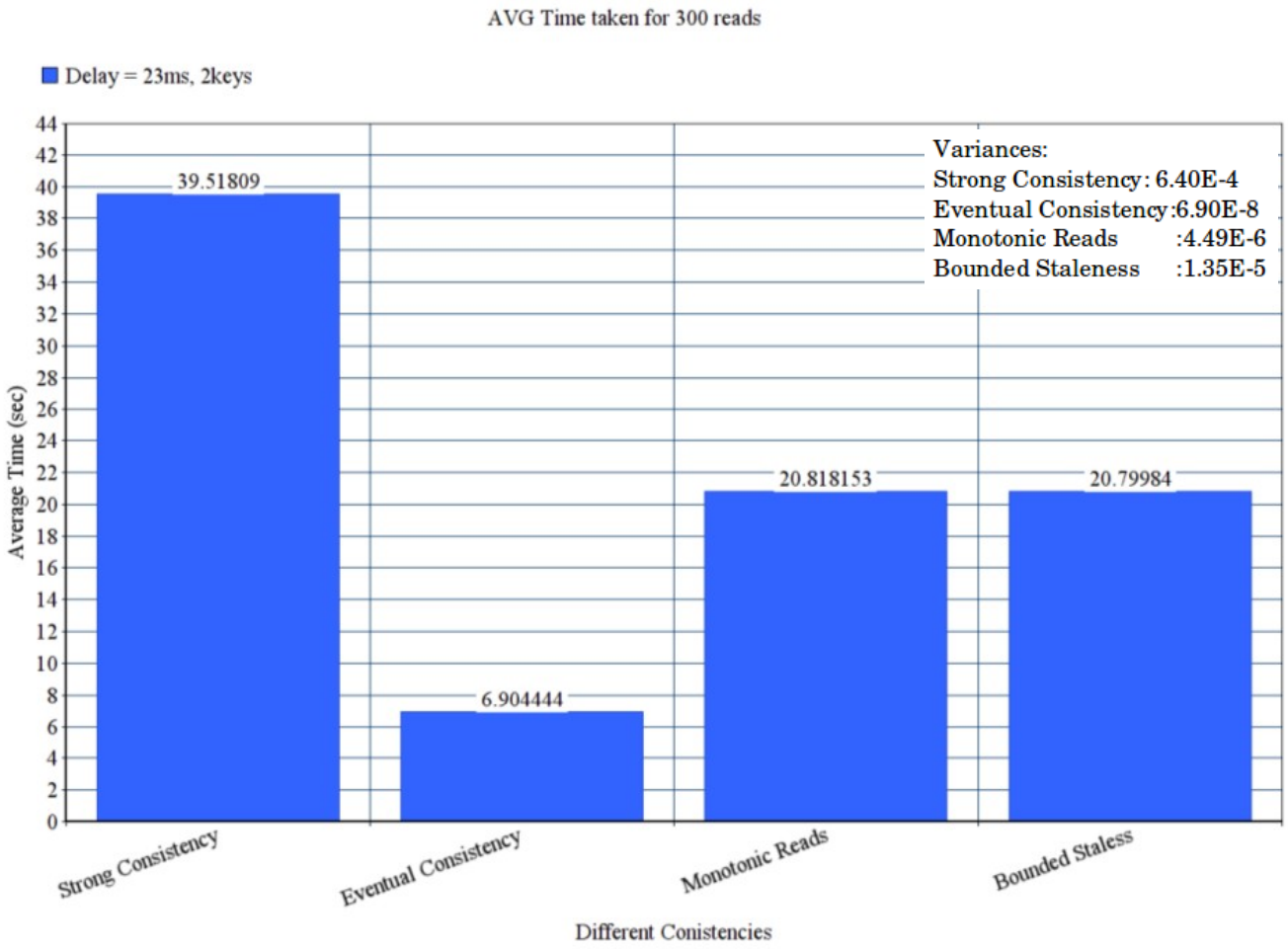
access a replica again by calculating which replicas have acceptable values.

The setup for the experiment in graph 3 is same as first experiment, except that the writes are performed continuously on only two keys. As seen before, variance is high for strong consistency reads. Time taken for strong consistency reads is increased relatively more because the probability of performing a read and write operation to same key is more. (Probability is close to 0.25 (½ * ½) whereas for previous experiment it is (1/10 * 1/10)).



Graph 3

The setup for the experiment in graph 4 is same as previous experiment, except that a network delay of 23ms is induced. Comparing this to graph 2, we can observe that monotonic and bounded staleness are not as close to strong consistency proving our point that the less popularity of keys is the reason for that in graph 2. Moreover, the readings of eventual, monotonic and bounded staleness remained almost equal to that of graph 2 readings because other schemes have nothing to do with popularity and they don't need locks. For strong consistency, the readings are almost doubled.



Graph 4

6.Conclusion:

Strong and eventual consistencies are two extreme consistency schemes. There is a lot of gap that can be exploited based on the application needs. So distributed systems should offer different consistency schemes to the applications. This paper discusses different consistency schemes and implements them and shows how they perform compared to eventual and strong consistencies.

7. References:

1. Terry, Doug. "Replicated Data Consistency Explained through Baseball."*Communications of the ACM Commun. ACM* 56.12 (2013): 82-89. Web.

2. Tanenbaum, Andrew S., and Maarten Van Steen. "CONSISTENCY AND REPLICATION."*Distributed Systems: Principles and Paradigms*. 2nd ed. 274-317. Print.