Algoritmos e Sistemas Distribuídos- Project (Phase 2)

João Carlos Antunes Leitão
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
and
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa
V 1.0

12 November 2017

1 Introduction and Goals

The goal of this phase of the Project is extend the previous phase (that provided a fault-tolerant and scalable global membership service) to build a scalable distributed data storage system similar (albeit more simple) to those offered by Dynamo, Cassandra, among others. We keep the same assumptions as in the previous phase of the project, in particular you should strive to make sure that your solution would operate adequately in a system with around 10.000 processes. Since replication is an essential aspect of this project, to ensure that data stored in the system is durable, you can assume that multiple faults across the members of a replica group will happen but not simultaneously. In particular, you can assume that such faults will happen in such a way that your system does have a large-enough time window to *regenerate* from the previous fault. Since you want to ensure that faults across replica groups are independent, it is essential that your solution can operate across multiple physical machines.
To build your system you will have to do the following tasks:

1. Enrich the global membership service with a logical (probabilistic unique) identifier. This can be done by adding to the process identifiers managed by this protocol an additional identifier, potentially obtained by performing an hash operation over some information that is unique to the process, such as its IP address and PORT. Make sure these identifiers are comparable and totally ordered.

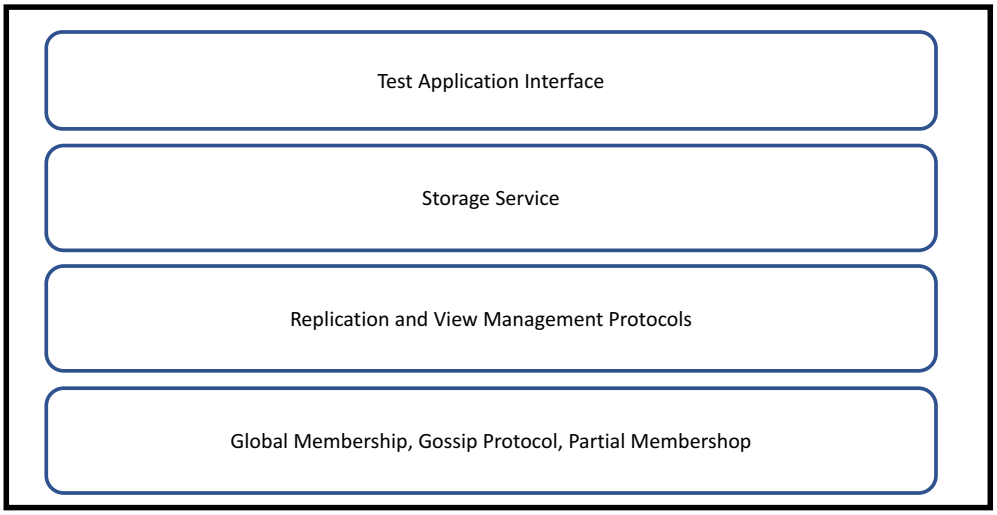
2. You will have to build a storage service that manages data objects. This service supports two operations, `Write(Key, DATA)` and `Read(Key)` (you might modify this interface to add relevant metadata if you feel the need to do so). The semantics of these operations are similar to those of replicated registers that we have previously studied in the lectures.

3. Since we want to ensure durability for the data stored within the system despite faults of individual processes, you will need to replicate data across a set of replicas. The minimum replication factor should be 3 replicas. To replicate data across the processes you can either use Paxos (or one of its variant) or a solution based on quorums. This replication solution should manage the replica set for each data object (more generally for each segment of keys i.e, a bucket). Meaning that when faults happen, a new process should be added to the replica set, and the failed process eventually removed. Additionally, when a new process joins the system, you might want to modify the replica sets of multiple buckets to improve load balancing across processes (the paxos-based solutions might be easier, since you will have to address the issue of managing the replica set for each data object (i.e, key) or more precisely, for each bucked of data objects (i.e, continuous set of keys)).

4. You will need to build a (simple) client (that executes in a different process or processes) that executes read and write operations over the system in order to test your implementation and measure its performance. The client test application should be able to emulate multiple clients and execute random operations over the datastore

system following a configuration defining the fraction of read and write operations. You can look into the design of YCSB[1] to get an initial inspiration.

The following Figure provides an abstract depiction of the algorithms (and application) that each process should have, the top layer is a simple protocol that receives and processes requests from clients by issuing commands to the lower layers.



Representation of a Node in the System
(Phase 2)

2 Implementation Suggestions

Similar to the first phase of the project, you can implement the project in Java, Scala, or Akka (with Scala). If you want to implement this in a different language you should discuss that option with the Course Professor. Using Akka, you probably want to have each process having a set of Actors, where each actor represents one of the layers depicted above. The client application should be implemented as an independent process, eventually in Akka materialized by a set of similar Actors, each representing an emulated client.
In relation to each of the layers, the materialization is the responsibility of students. However, you can consider the following suggestions:

Storage Service: You are not required to store data in a durable fashion to the hard disk. Objects can be maintained in memory only (an hash map might be a good implementation decision). You can also assume that if a read is issued over a data object that was never written, that read should return a default initial value. You should model data objects as blobs of bytes (i.e, byte[]) being the responsibility of the client to interpret the contents of the data blob. You should consider using some additional metadata associated with each object stored in the data store. The definition of the metadata required to provide the consistency guarantees expected of the data store is of the responsibility of the students. Consider the use of consistent hashing to select the process in the system that should be the primary for each continuous segment of keys (data buckets).

Replication Service: The suggestion here is for you to apply your knowledge of state machine replication and paxos (or multi-paxos) to design this layer. Alternatively, you can consider using a quorum system to achieve the same goals. Considering the selection of elements to form each replica group based on the relative order of nodes according to their logical identifier. For instance, to select the two additional replicas for a the contents managed by a given process, consider using the following two processes (considering the order of their logical identifiers) to form the replica set. Evidently these replica sets will have to be modified to react to process failures, or new processes joining the system. You will need to take care of state-transfer whenever a new process is added to a replica group.

[1] `https://github.com/brianfrankcooper/YCSB`

Client application: Each client process should be able to emulate multiple clients. This can be achieved by having multiple threads concurrently executing operations over the data store. The client process will be used in practice to measure the performance of the solution. To this end, the client should be able to measure both the total number of operations performed by time unit, and the average latency of all operations (and optionally the average latency of read operations and write operations).

2.1 Performance Evaluation

An important aspect of designing and implementing distributed systems is their (experimental) evaluation to access their performance. To this end, in this phase of the project you should conduct a simple experimental evaluation and report the results that you have obtained. To do this you will leverage the client application discussed previously.
While you can perform additional experimental work and report the results (written reports have no page limits), you should at the minimum do the following:

Run a system composed of (at least) 5 datastore processes scattered across at least 2 physical machines (in the same local network).

Run (in a different physical machine) a client emulating at least 20 clients, issuing a total aggregate of 1000 operations with the following read write distributions:

1. 0% writes 100% reads.
2. 10% writes 90% reads.
3. 50% writes 50% reads.

the operations are executed selecting a random key to manipulate, and when required to write, selecting a random value to write (with a fixed lenght, such as 200bytes).

Repeat the previous experiments but forcing one of the processes to fail during the execution of clients.

For each experiment run you should report the throughput of the system (number of operations per second across all clients accessing the system) and the average latency for all operations executed. You should compare these results and discuss them in the written report.
Very important: In your experiments you should make sure that all processes are actively performing operations and avoid a scenario where all objects being manipulated by a client are managed by a single process in the system.

2.2 Evaluation Rules & Criteria

Projects should be conducted by groups of at most 3 students, ideally with the same configuration as in the first phase of the project. Single student groups are highly not advised.
The project includes both the delivery of the code and a written report that must contain clear and readable pseudo-code for each of the implemented layers alongside a description of the intuition of the protocol. A correctness argument for each layer will be positively considered during grading. The written report **must** provide information about the experimental work conducted by the students to evaluate their solution in practice (including results).
The project will be evaluated by the correctness of the implemented solutions, its efficiency, and the quality of the implementation. With a distribution of 50%, 25%, and 25% respectively for each of the layers.
The quality and clearness of the report of the project will impact the final grade in 10%, however the students should notice that a poor report might have a significant impact on the evaluation of the correctness of the used solutions (which weight 50% of the evaluation for each component of the solution).
Each component of the project will have a maximum grade of (out of 20):

Storage Layer: 5/20.

Replication Layer: 8/20.

Experimental Evaluation: 5/20.

Client Application: 2/20.

Written Report: 2/20

(Yes, the sum of this is not 20)

2.3 Delivery Rules

Delivery of all components of the project is due on 4 December 2017.
The methods for delivery will be the same as in the first phase of the project (e-mail).