SDLE - Planned design

Emanuel Gestosa José Gaspar Lucas Sousa

November 2023

Our proposed design has three main components: a **Web Application**, that provides an interface that the client can interact with, and also locally store data that can be accessed offline; a **Proxy**, that is the gateway between the local application and the cloud infrastructure, with the limited responsibility of having full knowledge of the available cloud nodes, and instructing the local application where it should redirect its requests to; and several **Node**s, responsible for actually dealing with client requests of reading or updating shopping lists, as well as data replication. A high level overview of the architecture can be found below.

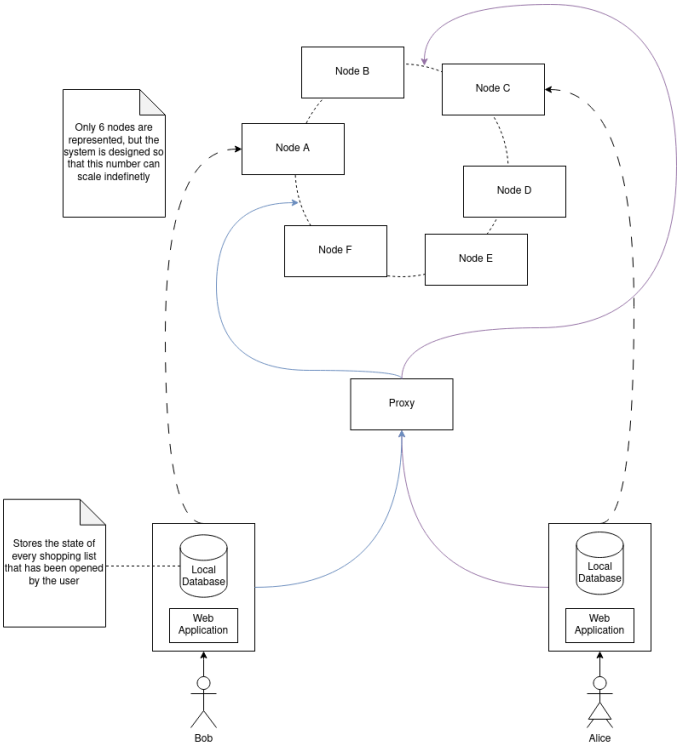

Figure 1: High level overview of the design

When the client opens a shopping list for the first time, it first asks the Proxy who he should be communicating it from now on for requests related to that list. So, from the clients perspective, the Proxy listens to GET requests on the form of *GET <shopping list id>*, and responds with an IP and the port of another machine. The Proxy determines this IP by using a technique similar to one used by Dynamo, applying a hash function to every Node, forming a sort of node circle, and applying this same hash function to the id of the shopping list, choosing the Node whose hash is immediately next to the hash of the list. So why does this Proxy exist at all? Why can't the client just communicate directly with the final machine to start with? Our design is made out so that machines (Nodes) in the cloud can be added or removed at any given time, so if we decided that the client should have knowledge, before interacting with the system, of the Nodes (by, for example, hard codding all the Nodes IPs and listening ports directly in the client code), we would need to force an update on the client every time the cloud infrastructure changes. With the Proxy, we just hard code this single IP into the client's code, and the client uses this to find out where to go from there.

Once the client has received the address of the Node responsible for handling requests of his shopping list (that Node is called the Coordinator), he is exposed to two API endpoints: a GET method to get the state of a given shopping list, and a PUT method to update the state of a list with a new one. The Coordinator Node then might consult/store information from a few other Nodes for replication purposes, to ensure availability of data in the event

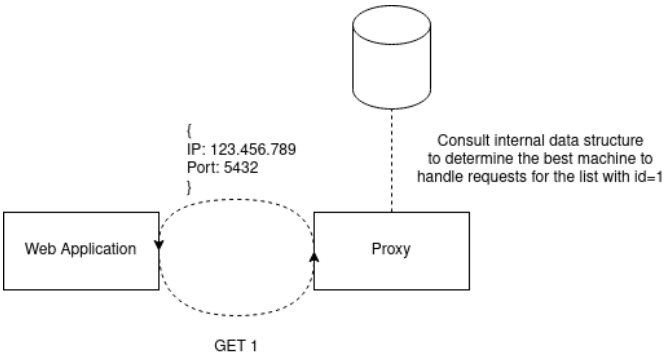

Figure 2: Web application-Proxy interaction

of an outage. Since we want to focus on high availability, the system will be prepared to allow writes even though the replicas are not in a consistent state. This can be achieved using a strategy based on quorums. The idea is that each node computes a preference list of nodes (next nodes found clockwise on the circle) for each object it stores (write). All nodes in the preference list keep a replica of that object. A write only succeds if at least N nodes, being N an arbitrary number, store the object properly. Under this conditions, on read operations, you can find different version of replicas that are then going to be "merged" based on the stored information and then retrieved to client. Even under the failure of a node, this stil works because the write does not need all the nodes to write successfully, only N.

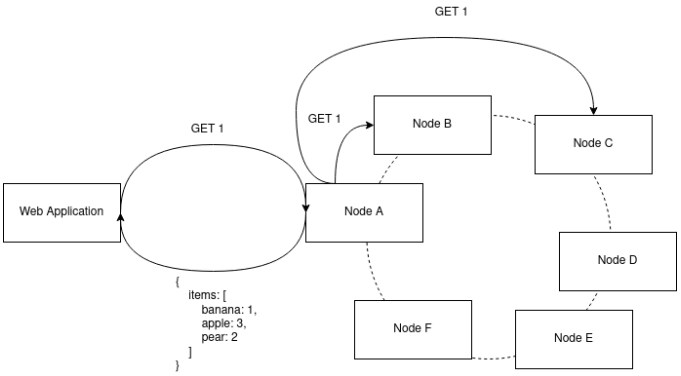

Figure 3: Web application-Coordinator interaction

Our final concern is how to add and remove nodes from the cloud. To add a new node, a machine that desires to become a Node, will tell the Proxy, with a special POST request, that it desires to join as a Node. The Proxy will then add it, and inform all the other nodes of this new addition. If a machine leaves, either by choice or by some server outage, there is also a POST request that can be made to the Proxy to inform of a Node leaving the cloud. Note that this request doesn't necessarily have to be made by the node that is leaving, but can be made by any Node that fails to establish a connection with the leaving Node. We would also like to note a security concern here: if this were to be done in the real world, this requests would be have a firewall in place, to prevent unintended actors from becoming a Node or forcing other Nodes to leave.

On a final note, we are aware that our Proxy is a single point of failure, and although we don't consider it to be a bottleneck in our design due to its limited responsibilities, there is the possibility that this machine could become unavailable, rendering our entire system basically useless. Therefore, we are considering the addition of *Backup Proxies*, that could be added dynamically by a system administrator by connecting to the Proxy and maintaining a list of proxies, that would act as backups in case of failure of the main Proxy. Then the client would, every once in a while, download this proxy list, and in the event of the Proxy becoming unresponsive, it could try a proxy in this list. This would fix the issue of a temporary outage of the Proxy, for everyone except fresh new users (which should be neglegible for a short outage, even for an application with millions of users). In the event of permanent failure of the Proxy, a update would need to pushed to the client application, in order to update the static Proxy address to some other new Proxy. Even better, if the area of the Proxy has bad weather possibly coming in future, we could push this update in advance as a preventive measure.