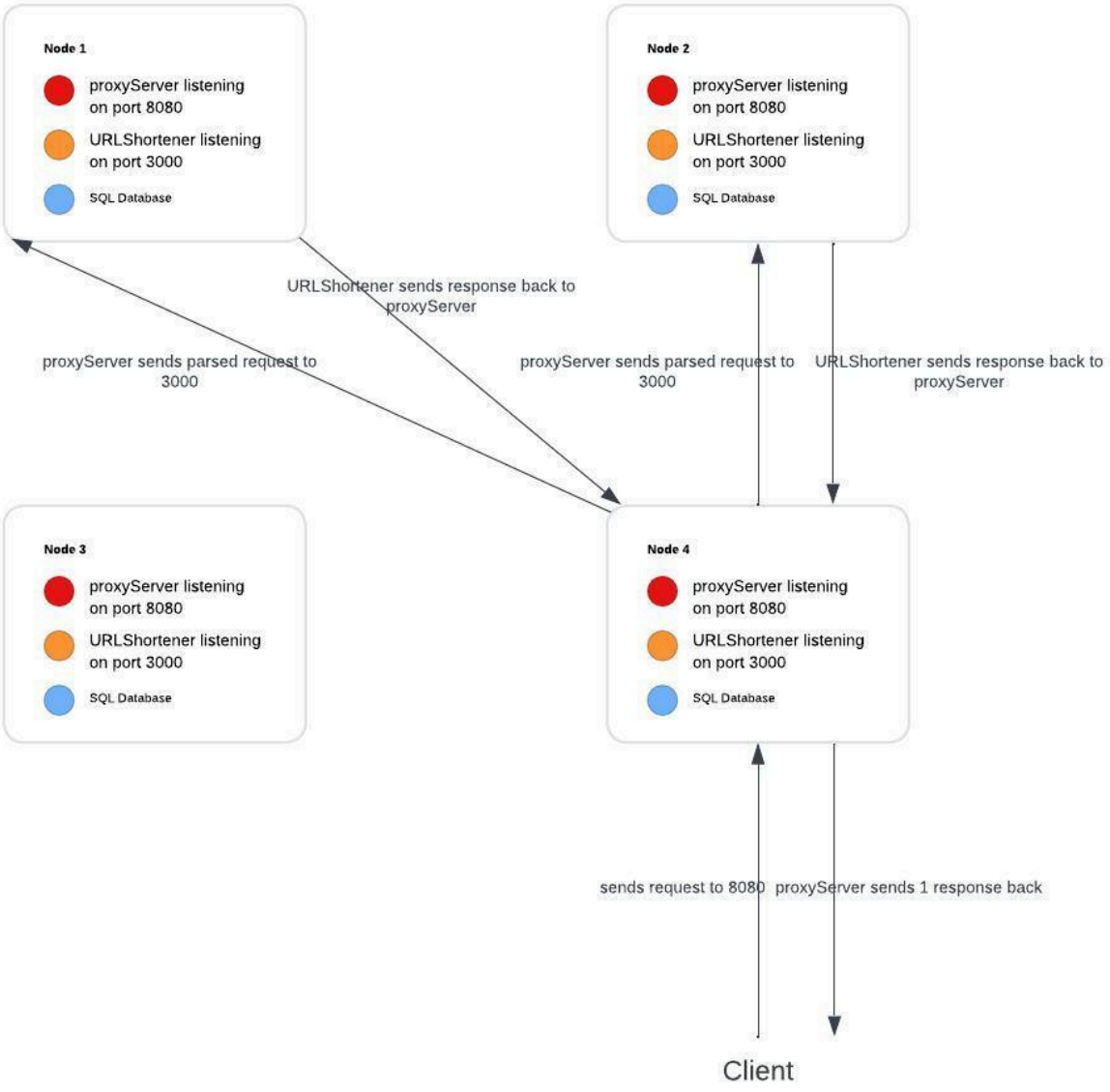
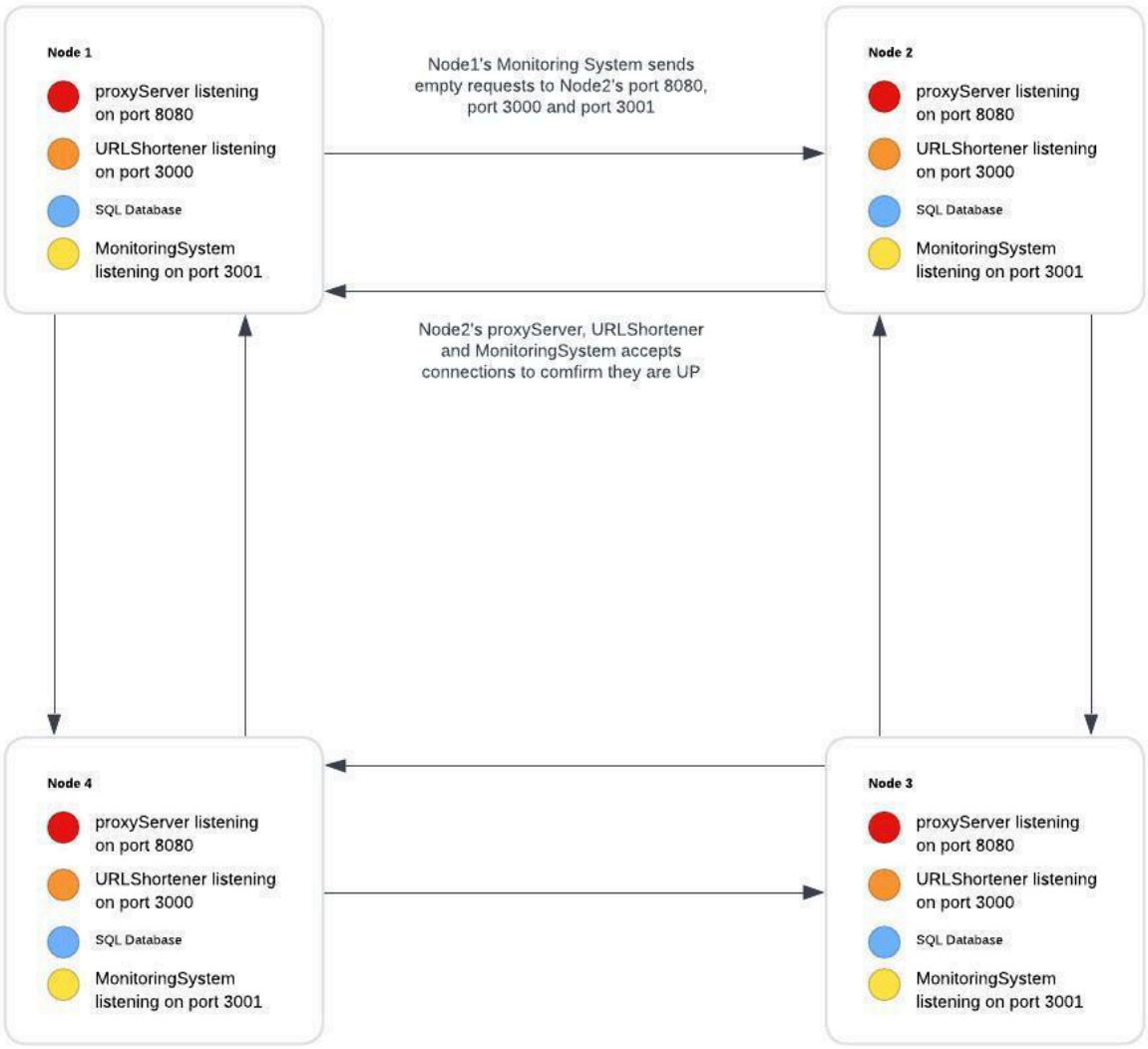
System Architecture

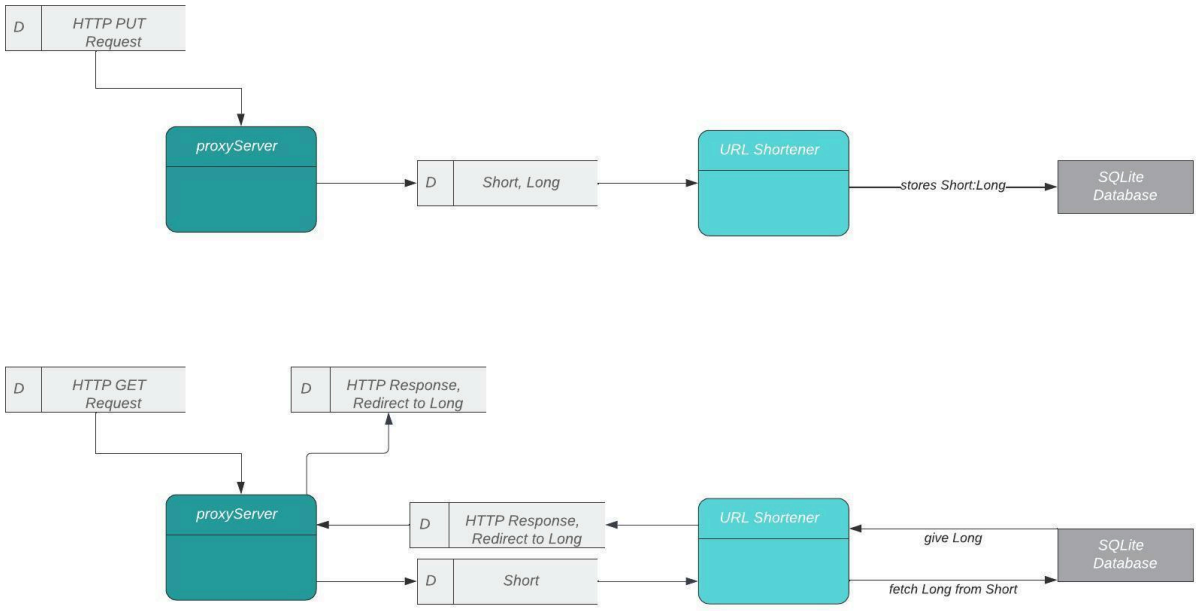
Application System Diagram:



Monitoring System Diagram:



Data Flow Diagram:



Discussions:

Consistency:

- The system does not guarantee consistency, because when the proxyServer sends a PUT request to the hashed database and subsequent backup databases, it does not confirm the response from all databases but only the first 200 OK response from any database.
- Therefore, if the client requests a read after the write, the proxyServer could potentially proxy the request to a database which has not yet written the data.

Availability:

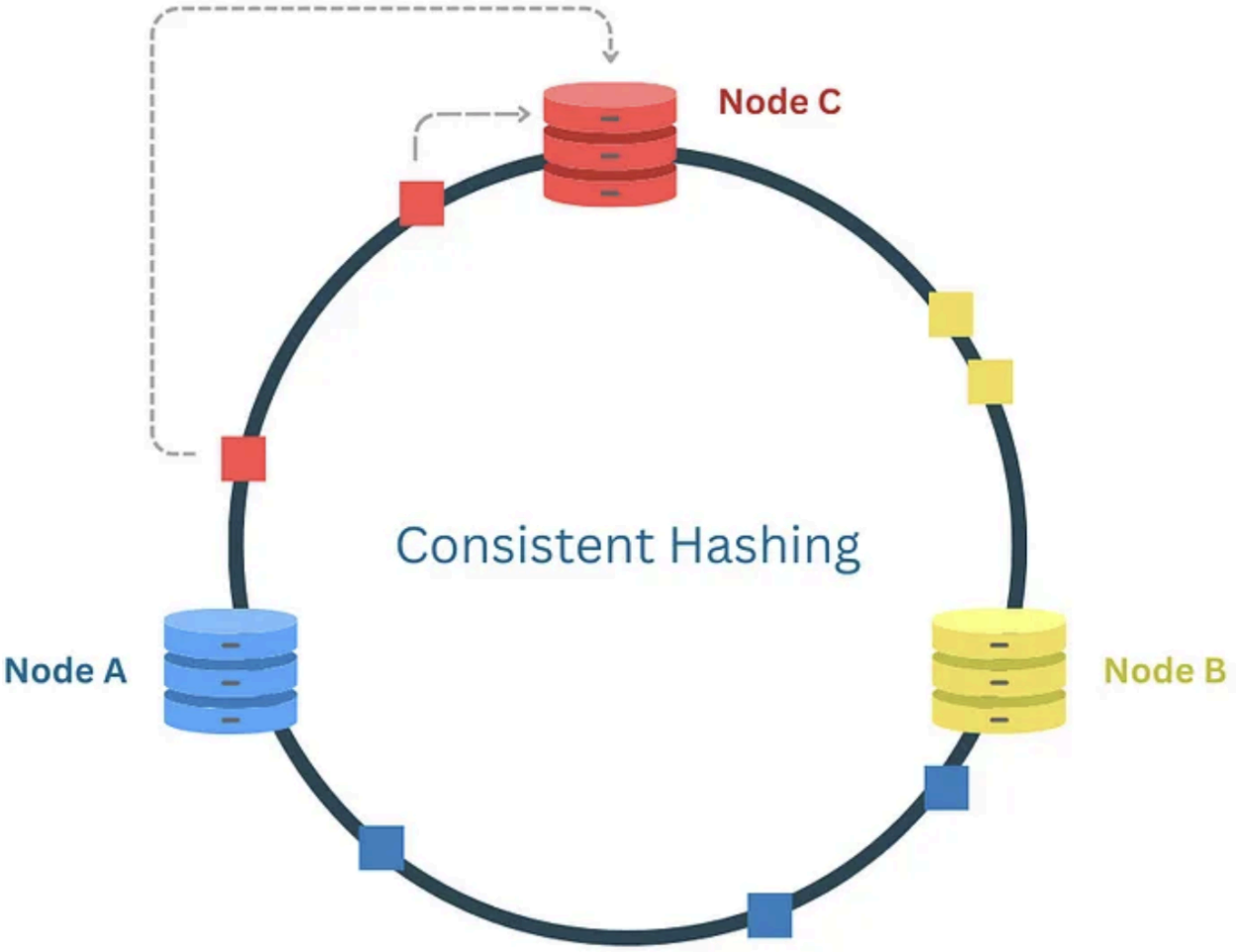
- The system guarantees availability as long as there is at least 1 node running in the cluster of nodes assigned to the request through hashing
- Each node in the system is identical to each other, and they all have a proxyServer and a URLShortener running, as well as a database. For each request, the proxyServer determines the cluster of nodes responsible for it through hashing the Short URL. Therefore, as long as 1 of the nodes is running inside the cluster, then the request will guarantee a response
- In general, if each request is assigned to a cluster of “n” nodes, then the system will guarantee availability until “x” nodes, where “x >= n” are down.

Partition Tolerance:

- The system is partition tolerant as long as there is at least 1 connection intact between the proxyServer and 1 of the URLShorteners from the cluster of nodes assigned to the request.
- The only communication between nodes will be between the proxyServer and the URLShorteners in this system, therefore as long as there is 1 connection intact between the proxyServer and 1 of the assigned URLShorteners, the system will function as expected.
- In general, if each request is assigned to a cluster of “n” nodes, then the system is tolerant up until “x” connections lost, where “x >= n”.

Data Partitioning:

- The proxyServer chooses a node to process the request through Consistent Hashing, using the name of the nodes and the short URLs, with MD5 hashing algorithm.
- From the diagram below, if the hashed short URL of the request falls between the hashed nodeA and hashed nodeC, then this request will be assigned to nodeC, and the Short to Long mapping will be found in node C’s database.

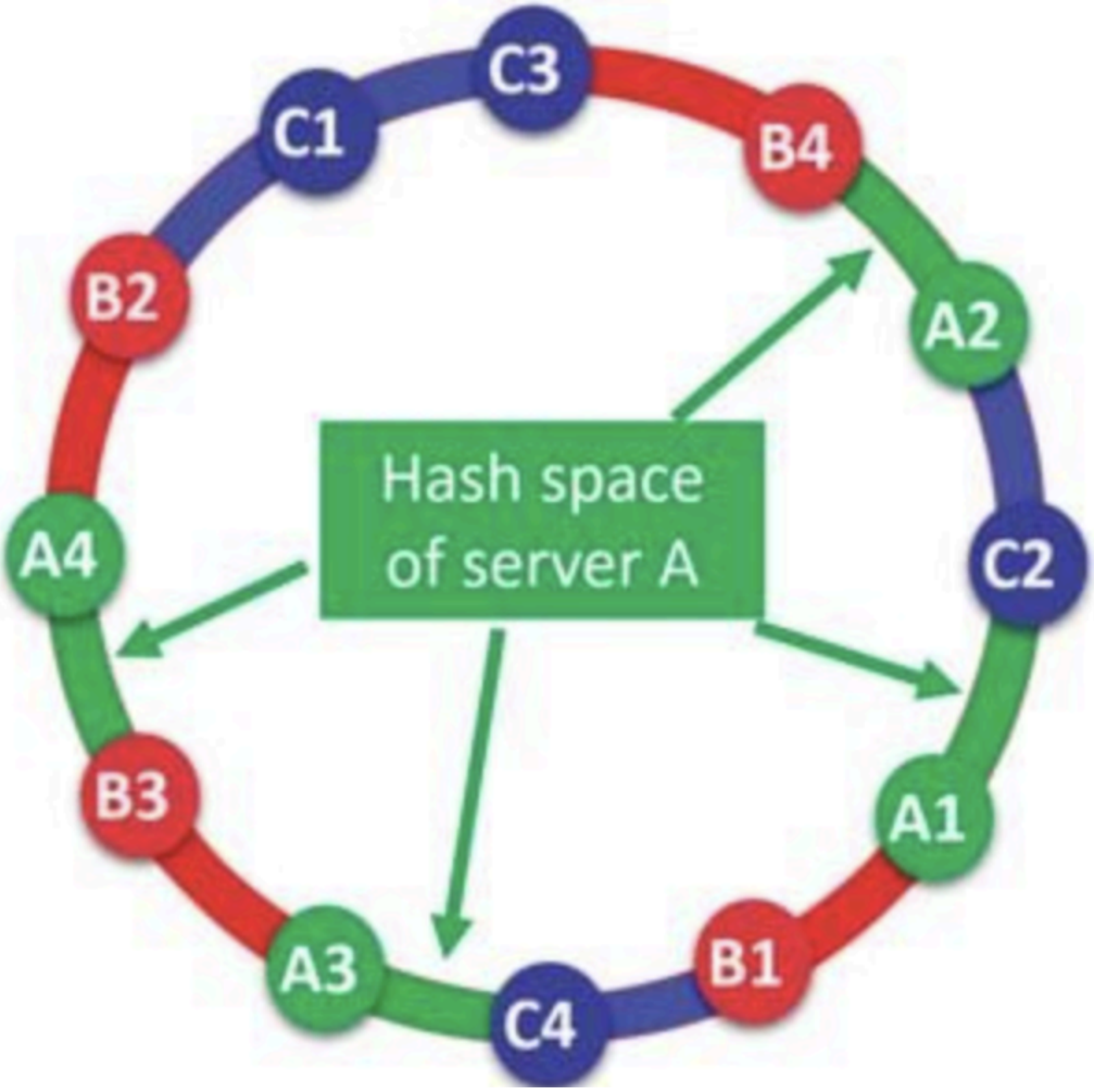


Data Replication:

- If we set the number of nodes assigned for each request to “n”, where “n <= the number of total nodes in the system”, then referring back to the diagram above: if the hashed short URL of the request falls between the hashed nodeA and hashed nodeC, and if “n==2”, then both nodeC and nodeB will be assigned to the request, and the Short to Long mapping will be found in both nodeC and nodeB’s databases.

Load Balancing:

- Load balancing is done through the Consistent Hashing system, in which through hashing the ShortURL we hope to direct the traffic to be processed on different nodes.
- However, if the range between the hashes of the nodes are different, for example if there is a significantly larger gap between nodeA and nodeB than between nodeB and node nodeC, then there is a higher probability for an incoming request to be assigned to nodeA than nodeB, resulting in imbalance. This could be improved through hashing multiple times per node so it can divide more evenly, like the diagram below.



Caching:

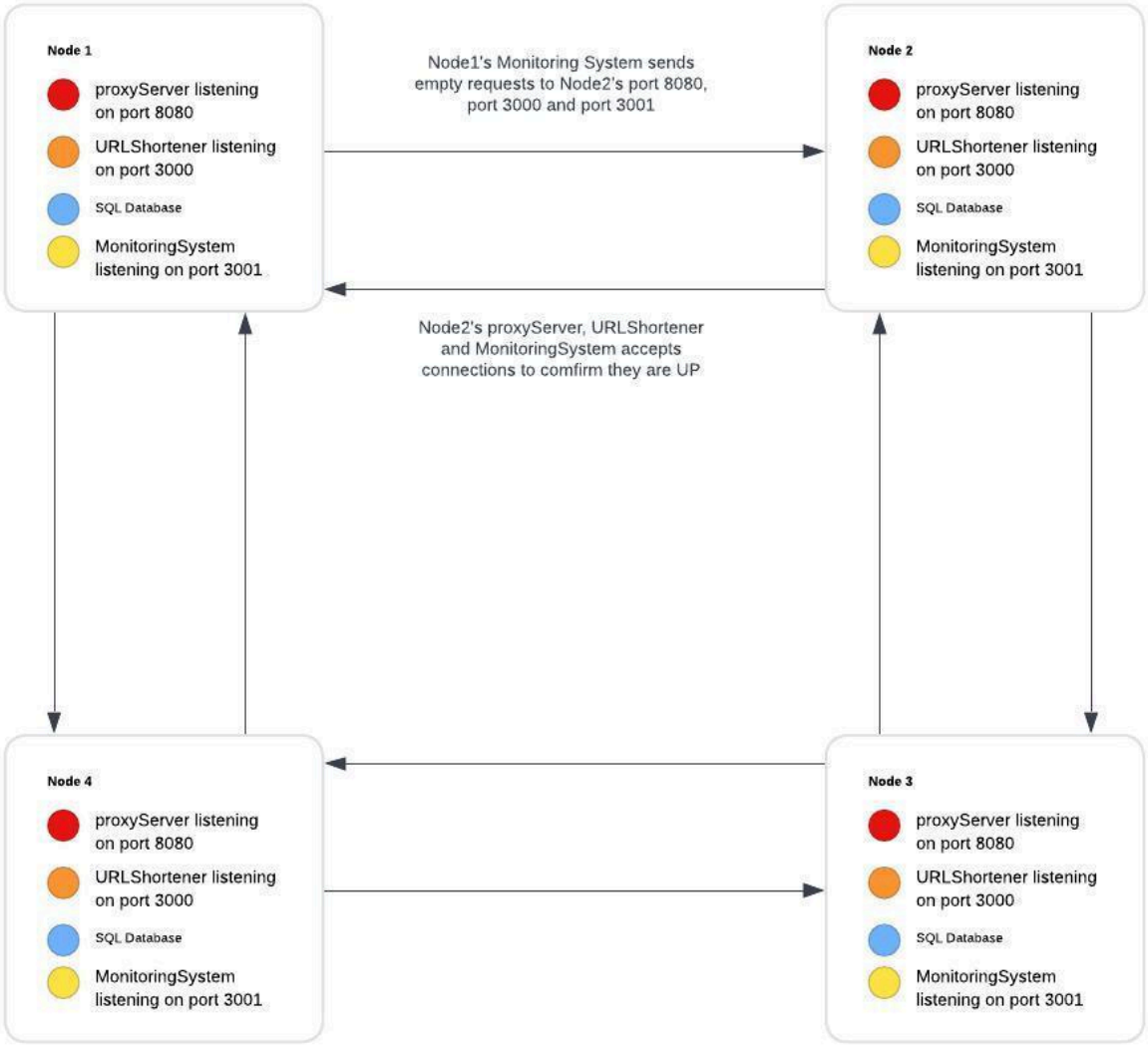
Process Disaster Recovery:

There are in total 3 services running per node:

1. proxyServer running on port 8080: if this process dies, then the client will have to direct their requests to another node's proxyServer manually. This process is stateless, so when it is revived nothing needs to be done
2. URLShortener running on port 3000: if this process dies, when it is revived it can recover by accessing its log file in projectroot/logs/writes/<node name>.log. This is because the proxyServer logs all write requests it sends to a node to the NFS mounted folder projectroot/logs/writes/<node name>.log. The recovery process will consult the

MonitoringSystem for the time when the node is down, and send all requests it should've received during its down time to the node. However, if the log files are lost or corrupted, then the system will fail.

3. MonitoringSystem running on port 3001: if this process dies, its watcher will take over its responsibility to look after the next node. For the diagram below, if node 2's MonitoringSystem dies, node 1's MonitoringSystem will take over watching node 3. When node 2's MonitoringSystem is revived, node one will give up its responsibility watching node 3, and node will continue to watch node 3.



Data Disaster Recovery:

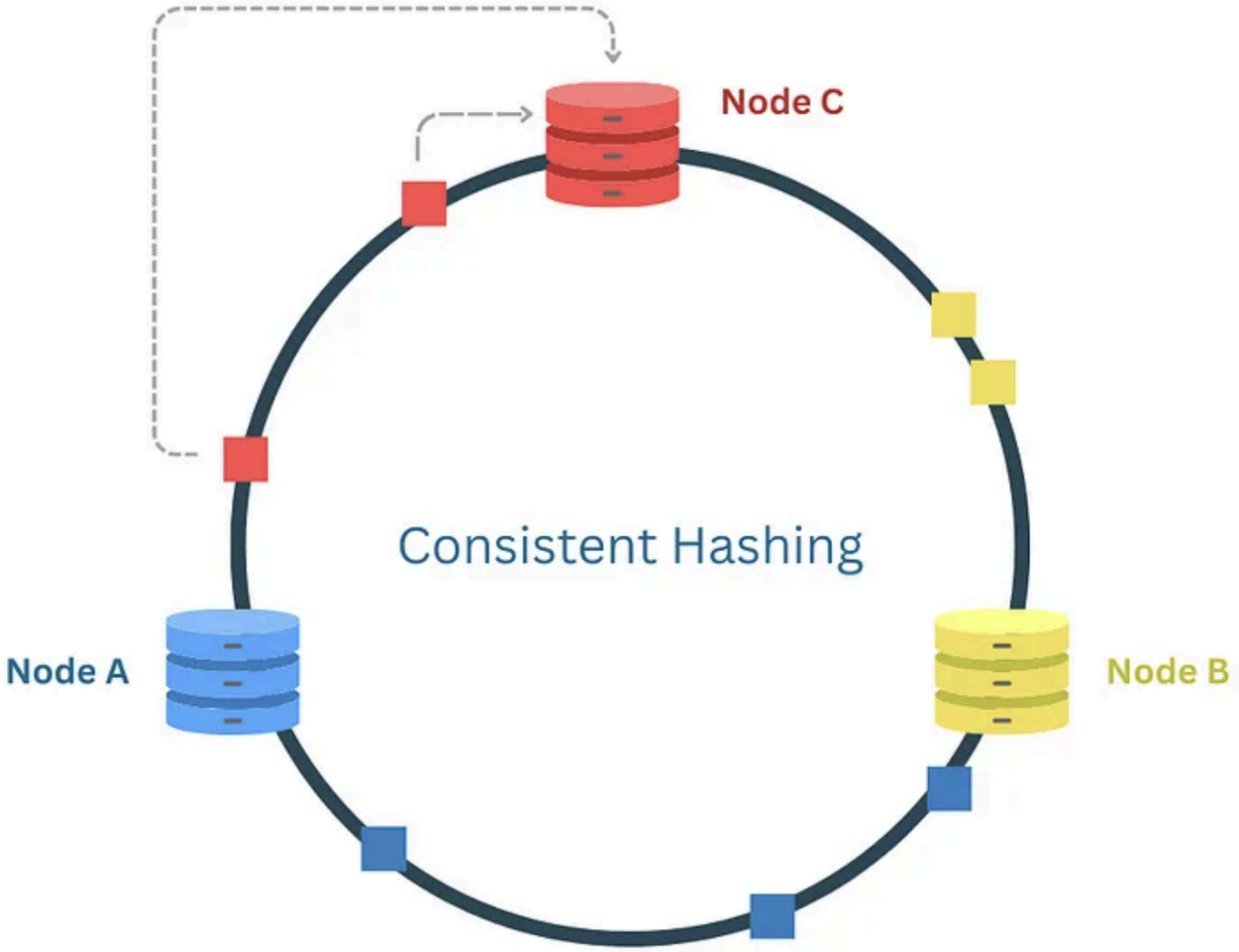
- If the database of a node is lost, it can recover by repeating the instructions inside the log file.

Orchestration:

- To start the system, we can run “./start.sh” inside the project root folder on any of the NFS mounted machines. The script takes in nodes listed in the “nodes.txt” file, and for each node it ssh into it, run the script “startLocal.sh”, and outputs the node’s stdout and stderr inside the “project root/logs/starting/<node>.log” file.
    - The script “startLocal.sh” sets up the local node with 3 services: the proxyServer, the URLShortener, and the MonitoringSystem, all running in the background.
- To end the system, we can run “.end.sh” inside the project root folder on any of the NFS mounted machines. The script takes in nodes listed in the “nodes.txt” file, and for each node it terminates all services running on it, and outputs the nodes’s stdout and stderr inside the “project root/logs/ending/<node>.log” file.

Health Check:

- When the MonitoringSystem service is run on a node, it takes in nodes listed in the “nodes.txt” file and finds its primary subject. The primary subject is the node after this node in the Consistent Hashing circle. For example, in the diagram below C would be the primary subject for A, A would be the primary subject for B, and B would be the primary subject of C. If node2 is the primary subject of node1, then node1 would send connection requests to node2 on ports 8080, 3000, and 3001, and node2 would respond by simply accepting the request. The results for each node’s health will be printed in “project root/logs/status/<node>.log” by its health checker. The health checker checks its primary subject once every 5 seconds.
- The MonitoringSystem is fault tolerant, up until there are less than 2 working MonitoringSystem programs running in the system. This is because when a node’s monitor is down, its health checker detects it and will automatically take over the nodes it's watching over, and continues to output to the “project root/logs/status/<node>.log” file. However when there is only 1 working MonitoringSystem running, no node will be watching the health of the node with the running MonitoringSystem.
- Unfortunately, MonitoringSystems are stateless, so after they recover they lose information on its subjects, including the time when they were last down. This means after the subject is up, it will have to recover by reading its entire “project root/logs/writes/<subject>.log” file, not only entries after it’s been down.



Horizontal Scalability:

- The system can scale horizontally by adding one more node. This can be done by inserting the new nodes into the file “new.txt”, and running the script “scale.sh”.
    - In “scale.sh”, it does the following:
        1. For each node in new.txt, it finds its “parent” node, which is the node previous to it in the Consistent Hashing circle. For example, from the above diagram, if after getting hashed, the new node nodeD is placed between nodeC and nodeB, then its parent is nodeC. The parent is significant, because in the new System involving the new node nodeD, it will be responsible for the backups datas for the n(number of backups per node) nodes before it, and its parent nodeA will have those datas.
        2. Then, for each node, create a new log file: “root project/logs/writes/<new node>.log”, and copies the parent’s write logs into this log file.
        3. Append the nodes in new.txt into nodes.txt
        4. Restart the system

5. Setup the new node by populating it with its write logs previously prepared

- The new node will be fed with all the necessary backup data, but it also includes data from a node for which it is not responsible for. For example, if n = 2, then the new node nodeD which is placed in between nodeC and nodeB will be fed with data from nodeC and nodeA, but also with nodeB who is its parent's responsibility instead. However, I've decided this is more desirable than logging the owner of each writes, because distinguishing each owner will take up a lot of computational resources and be a lot slower.
- This approach strives for minimum down time, because the log files are copied before the system restart and the new nodes are setted up after the system has been restarted. The system will be down during the restart process, but it will only take a few seconds.
- Right after the restart, the new node will lack the necessary backup data, so if "n=1" and it was the sole backup for the previous node, it will no longer be able to serve as a reliable backup, potentially leading to inconsistencies.

Vertical Scalability:

Threading is used to address vertical scalability

- In the proxyServer:
    - When a client's connection is received, a thread is created to run the proxy logic and the main thread continues to listen for the next connections.
- In the URLShortener:
    - When a client's connection is received, a thread is created to run the shortener logic and the main thread continues to listen for the next connections.
- While scaling horizontally, a worker thread is created for each new node to send write requests from its log to its URLShortener server.

Threading achieves vertical scalability through utilizing the full potential of the node's physical resources, such as the CPU, as the load on the server increases.