

**A Project Report on UBmarketplace**

**Phase 4 Distributed Systems Master’s in Computer Science**



**May 6, 2022**

**University at Buffalo**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Contents** | **Page No.** |
| **01** | Introduction | 2 |
| **02** | Design Overview | 2 |
| **03** | Working Principle | 4 |
| **04** | Log Replication |  |
| **05** | Technical Implementation | 7 |
| **06** | Phase2 Log VS Phase 4 log replications | 9 |
| **07** | Deployment using Docker Container | 11 |
| **08** | Execution and Validation | 13 |
| **09** | References | 26 |
|  |  |  |

**Introduction**

The idea of our project is to resolve the issue that the incoming students or current students at the university at buffalo might face. Let’s consider Sara who will be joining the college and is interested in staying off-campus. She needs all the necessary goods for a proper stay like a study table, a chair, a desktop, or even a bicycle and so on. Buying all the things on her own might become a huge burden as they cost a lot. What if UB has a marketplace where students trade things at a cheaper price. This is exactly where our idea aroused as we have faced this issue and thought as many students will eventually leave the campus after their graduation and wanted to sell their products which both incoming and outgoing students of UB could benefit from. Our UB Market Place will be their one-stop shop for trading all the student-related pre-owned products at the best prices. This is all about the project that we are trying to build.

**Design Overview**

**Fig. 1.1**

**A single Node**

**Docker**

**Spring Boot**

**REST Controller**

**Mongo DB**

**UB Market Place React UI**

request

response

Above Fig 1.1 shows the design of a single node or container and its working design.

In this phase, we enhanced the project by replicating the containers to 5 nodes and making them work in different servers which are interconnected for Leader Election, Heartbeats, log replication, and Timeouts in RAFT to achieve consensus. Each node runs on 8080, 8081, 8082, 8083, 8084 ports respectively with node names as node1, node2, node3, node4, node5.

Below Fig 1.2 represents 5 nodes in the system connected to achieve consensus, and the leader election mechanism, log replication is explained below.

Req/Res

Req/Res

Req/Res

Req/Res

**Fig 1.2 Nodes Communication through leader**

Each node is connected and communicates over Sockets using UDP (User Datagram Protocol).

**Working Principle**

As shown in the above figure 1.2, nodes/state machines are connected to implement consensus in RAFT. At any point in time, at most one node acts as the leader which monitors and forwards requests to each node and makes sure the logs and commits of the majority of followers are in sync. Each node in the system consists of a currentTerm, votedFor, timeoutInterval, and heartBeatInterval, log parameters (PrevIndex, LogIndex, prevlogterm, Commitindex, term, success) to store the values at each instance of changes in the node role.

|  |  |
| --- | --- |
| Parameters | Description |
| currentTerm | Represents the term number and updates the counter for every election. |
| votedFor | Represents the vote for which the current node voted. |
| timeoutInterval | Time interval which makes a node timeout for a certain range of time. Here we generated time randomly in a range of 500ms and 1000ms to make the node timeout. |
| heartBeatInterval | Range of time in which heartbeat is sent in equal intervals. |
| Log | Entries of commands of every node to be performed are sent by the leader to the follower nodes. |
| Prevlogindex | Log index of the leader before committing the current request |
| Prevlogterm | Previous Term of the leader |
| CommitIndex | Index of committed requests in the current term. |
| Term | The current term of the consensus |
| Success | Committed or not? |

At the start of the servers, currentTerm, and votedFor are null since the leader election does not happen. All the nodes in the system are set as followers at first. The timeout counter starts, once the timeout is set to zero for a node, the leader election starts with updating the currentTerm counter by 1 and starts the election. The timed-out node is set as a CANDIDATE where this node requests a vote from the other nodes by incrementing their terms. Followers send vote acknowledgment to candidates. Follower node may reject vote based on the below condition:

If (lastTermV > lastTermC) OR (lastTermV == lastTermC && lastIndexV > lastIndexC)

Once the candidate receives the majority number of votes i.e., half the number of nodes +1, then the candidate will act as leader. This leader handles requests from the client and forwards them to its followers in the system. Parallelly, heartbeats are sent to followers by the leader for every heartbeat interval time. If heartbeats are not sent within the set time interval, then it is assumed as the leader is not active, or if a follower is timed out, then a new term leader election starts, and these repeats. There may be a condition of time-outs for multiple nodes, hence those acts as candidates.

For every term, elected leader checks for the logs of their followers by comparing followers’ log term and log index. If it is observed that the logs of the follower are not in sync with the leader. Then hence leader decrements its log index by 1 and compares the log. This decremental process continues till the respective follower node matches with the leader’s log. Once the log index matches with the leader, then incrementally nowhere updates the follower logs. The same process is applied to every follower for every new term. Once the logs of the majority of followers are updated and are in sync, then the current request or command from the client will be committed.

**Leader Election:**

Votes to Votes to Votes to

Votes to Votes to

Follower votes to the candidate only if the below condition is True:

If (lastTermV > lastTermC) OR (lastTermV == lastTermC && lastIndexV > lastIndexC)

For each election, the term field is incremented and the candidate node which gets the highest votes is set as leader and others as follower nodes. Timeout interval, heartbeat sender, the listener at followers run in separate threads

Listeners are the followers who acknowledge the heartbeats sent by the leader of the system. These timeout checkers, heartbeats, and listeners will run on separate threads.

**Log replication:**

index: 1 2 3 4

|  |  |  |  |
| --- | --- | --- | --- |
| Cmd1 | Cmd2 | Cmd3 |  |

Log of leader:

index: 1 2 3 4

|  |  |  |  |
| --- | --- | --- | --- |
| Cmd1 |  |  |  |

Log of follower2:

The leader has access to the log index of all followers:

From the above example the next indexes of followers are 3.

nextIndex -> {node1: 3, node2: 2, node3: 3, node4: 3}

From the above picture, the log of leader and follower2 are not in sync, hence leader decrements the index by 1 i.e., to index = 2, and now compares the log. This is repeated until log indices are the same. Since the log is in sync till index 2, hence incrementally nowhere updates the follower2 logs. The same process is applied to every follower every new term. Once the logs of most followers are updated and are in sync, then the current request or command from the client will be committed.

**Technical Implementation**

As shown in Fig. 1.1, implementation of the above problem statement mentioned in the introduction is a collaboration of several technological components namely User Interface, Database, API (Application Programming Interface), and Container to deploy the project on the local server.

**Technologies Used:**

1. ReactJS – To develop a basic User Interface that accepts the Name, ID, and age of the student.
2. Spring Boot Framework – Framework used to develop API to interact with the database.

**For Each Node:**

ReactJS is used to develop a User Interface that contains a form that accepts Name, UB\_ID, and age with some basic validations.

These parameters of students are implemented in student class in which the objects of the student class are used by the Spring Boot application to parse and store data in the database. Database fields are created using the student object itself in the application.

Database properties are incorporated in the application properties file of the application.

In this phase, log replication, leader election, heartbeats, timeouts, client requests are handled.

**Interconnection:**

Five child Nodes are created i.e., created separate containers for each node. These nodes are interconnected such that they communicate with each other.

Docker- compose.YAML file is updated to connect these nodes using networks parameter.

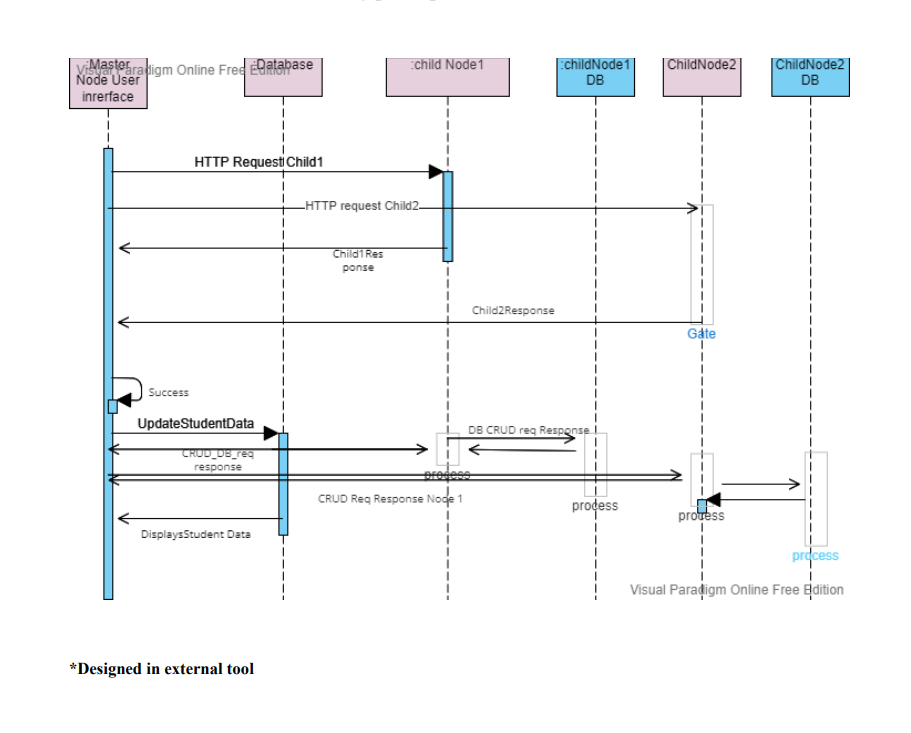
In this phase of implementation, no client requests are made or handled through RAFT. Hence focusing on the internal working of nodes as discussed above.

Finally, Maven is used to cleaning and build the dependencies required for the application.

**Phase2 Log VS Phase 4 log replications:**

In phase2, the log replication is based on the master node whereas, in the current phase, the log replication is purely based on the consensus i.e., achieved by leader election, heartbeats, timeouts, and log consistency.

For each term, the leader checks the log of their followers and makes sync with it by checking each log index detrimentally until a log index match is found.



**Phase4: Log replication:**

index: 1 2 3 4

|  |  |  |  |
| --- | --- | --- | --- |
| Cmd1 | Cmd2 | Cmd3 |  |

Log of leader:

index: 1 2 3 4

|  |  |  |  |
| --- | --- | --- | --- |
| Cmd1 | Cmd2 |  |  |

Log of Follower2:

From the above picture, we now see that there is an increase in the log index in leader so leader updates the logs of followers in next RPC by using the nextIndex as mentioned below. Once the logs of most followers are updated and are in sync, then the current request or command from the client will be committed. This leads to the incrementation of commitIndex.

The leader has access to log index of all followers:

nextIdx -> {node1: 3, node2: 3, node3: 3, node4: 3}

**Deployment of Application using docker container**

Docker is a platform that enables developing, deploying, and delivering applications in the form of containers that can run irrespective of the infrastructure of the local system, that’s the service provided by the docker called OS-level virtualization.

These containers are linked using a docker-compose file which contains the specifications and properties of the Spring Boot application and MongoDB.

Below are the few commands to build and run the container.

1. docker-compose build or maven build.

To build and link containers mentioned in the docker-compose file.

1. docker-compose up

To pull and run the containers.

The server can be accessed via <http://localhost:8080/>

We have tested the conditions using API calls,

We made names for nodes as node1, node2, node3, node4, node5

**Controller**

Please check your terminal for all the log information. We will discuss more in our video recording.

* To know which node is the leader use

[**http://localhost:8080/getLeaderInfo**](http://localhost:8080/getLeaderInfo)

* To make a leader as a follower use the following

**<http://localhost:8080/convertToFollower?node=node1>**

* To make a node timeout use the following API

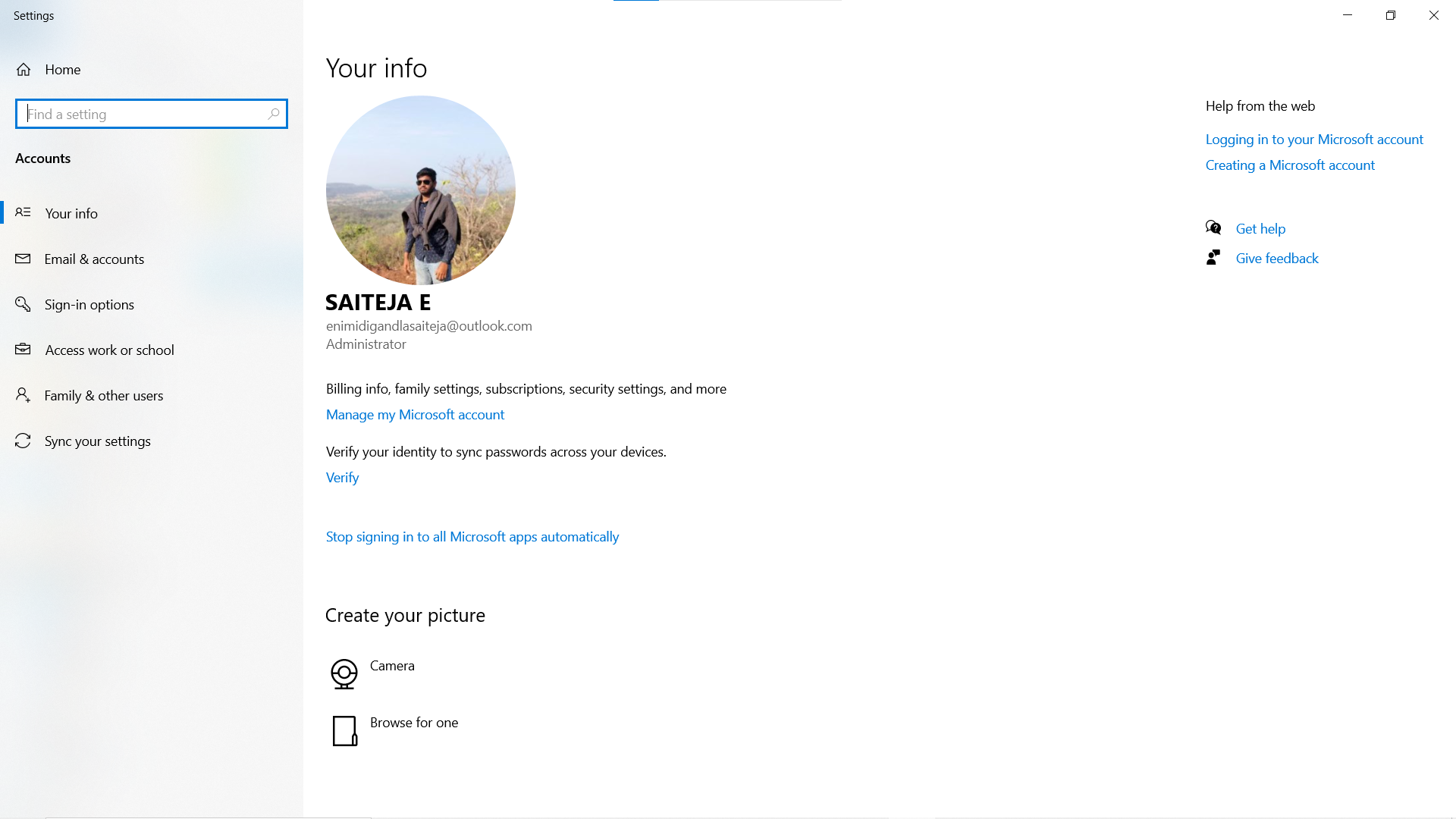
[**http://localhost:8080/timeout?node=node2**](http://localhost:8080/timeout?node=node2)

* To shutdown/ stop a node please use docker application and check the logs in terminal.
* STORE-<http://localhost:8080/controller/store?key=4&value=4&node=node2>
* RETRIEVE-<http://localhost:8080/controller/retrieve?node=node4>

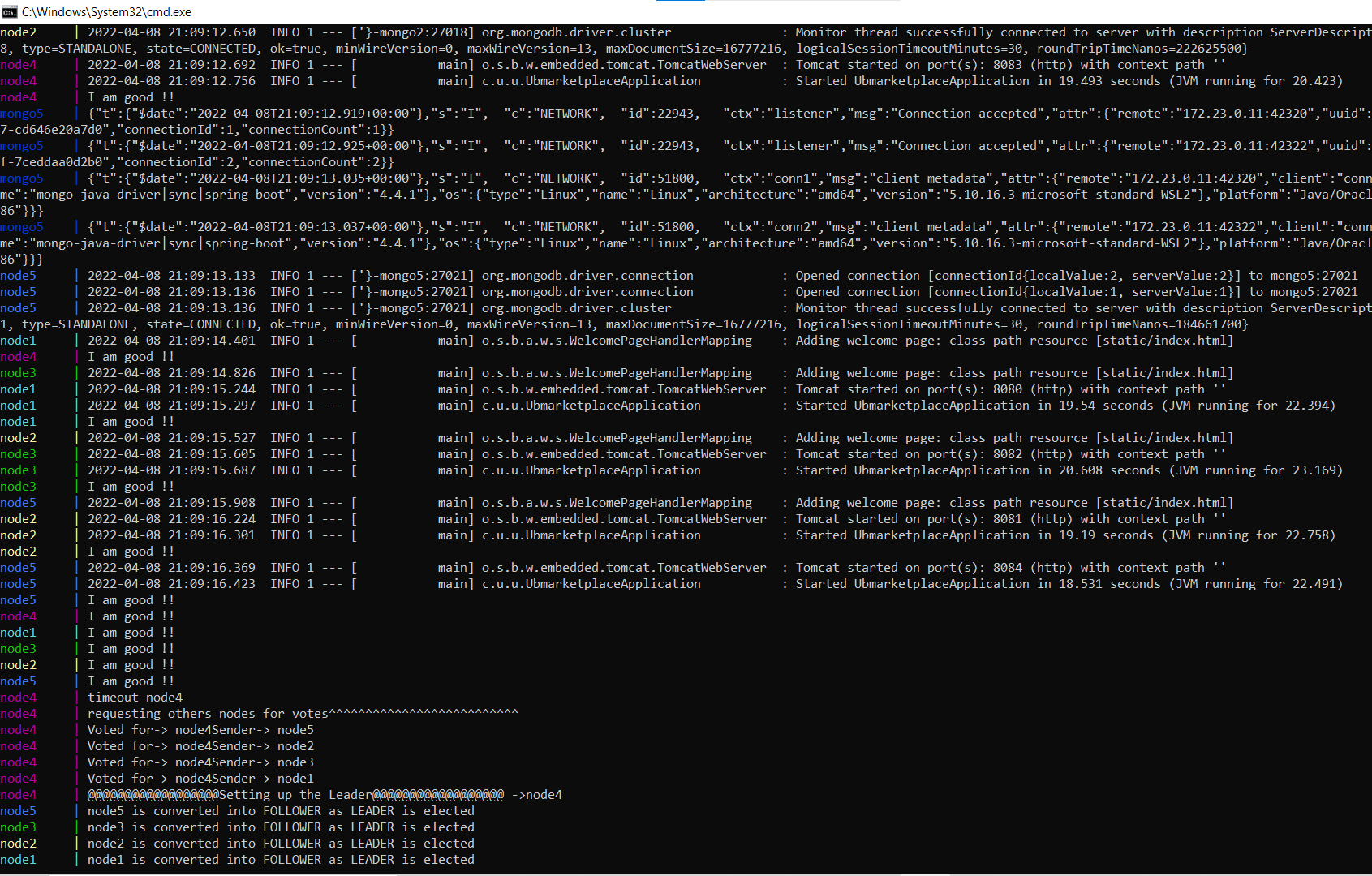
\*Change node parameter to node1, node2, node3, node4 and key, value params if necessary.

**Execution and Validation:**

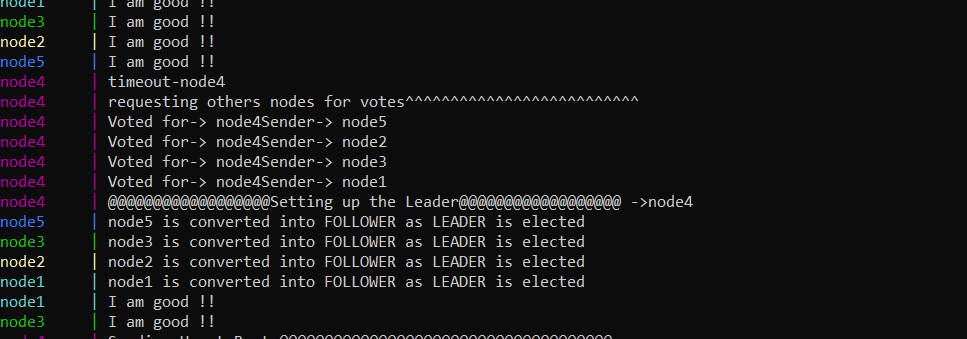
Identity of My Machine



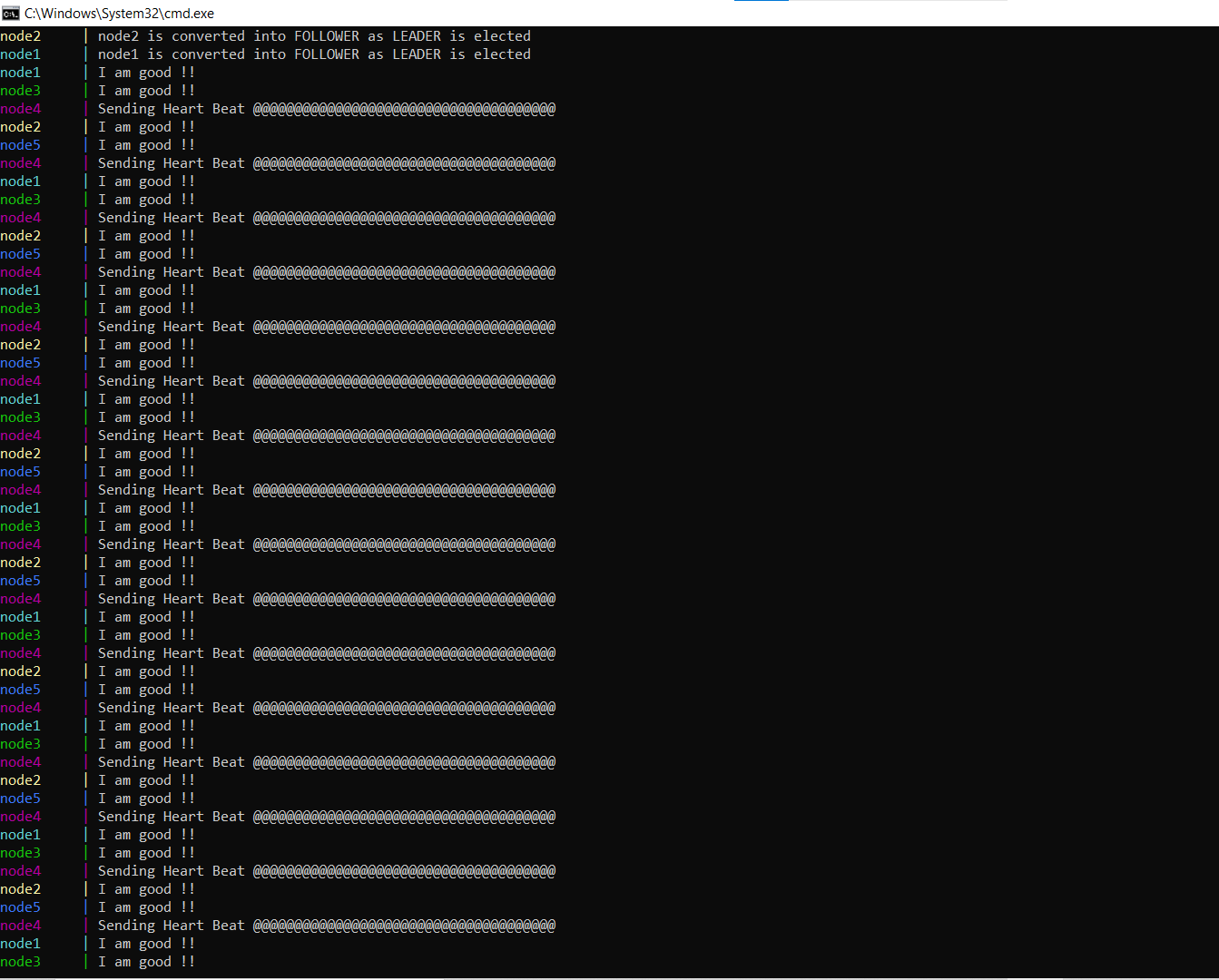
On start of the nodes:



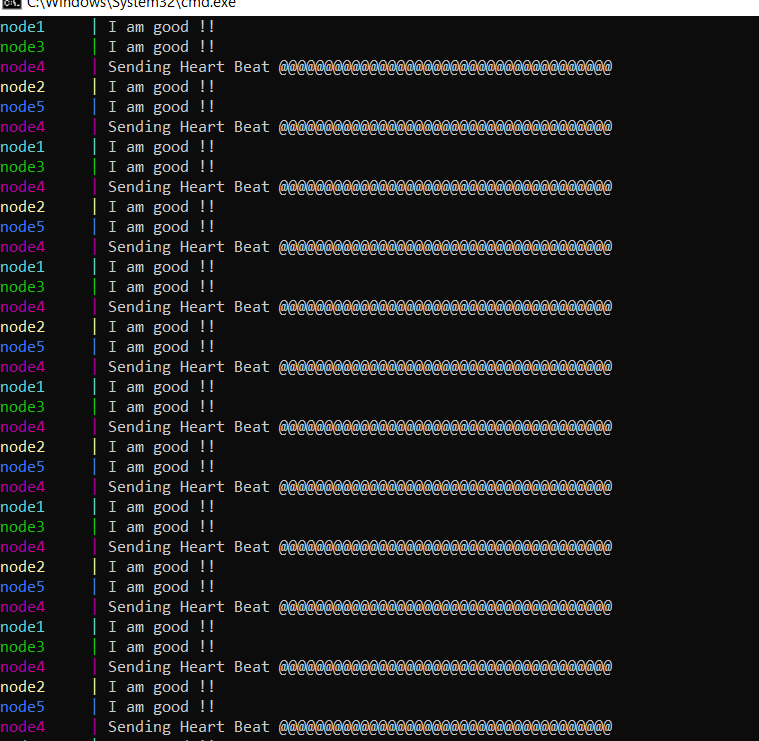
Node1 got timeout4 and requesting modes to other nodes and got the majority votes, hence selected as leader.



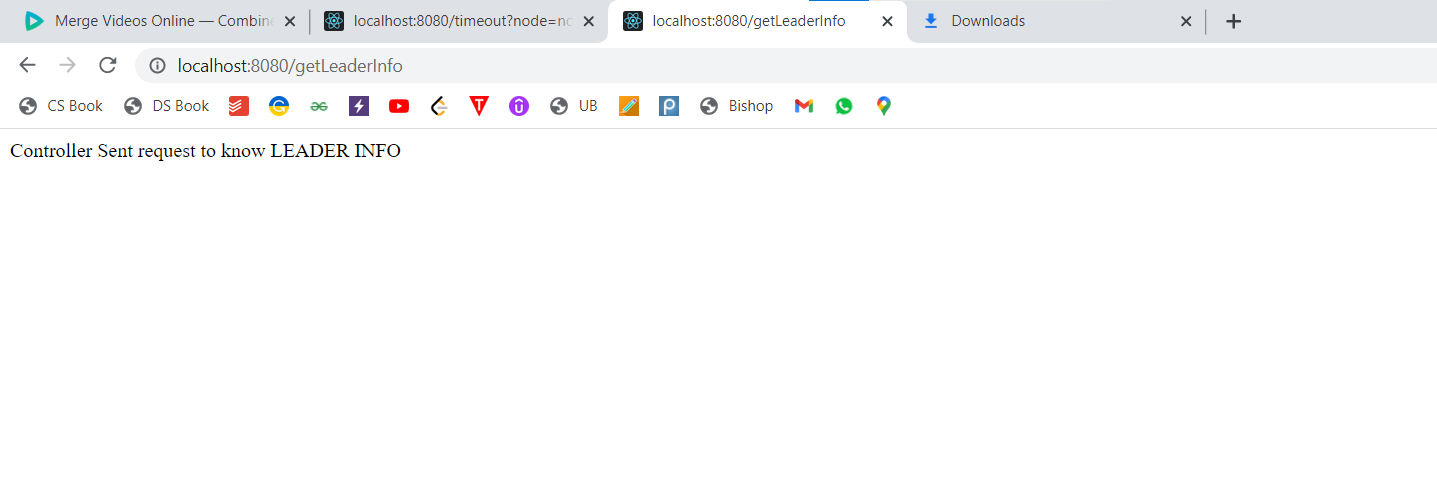
Heartbeats: “Sending Heart Beat”

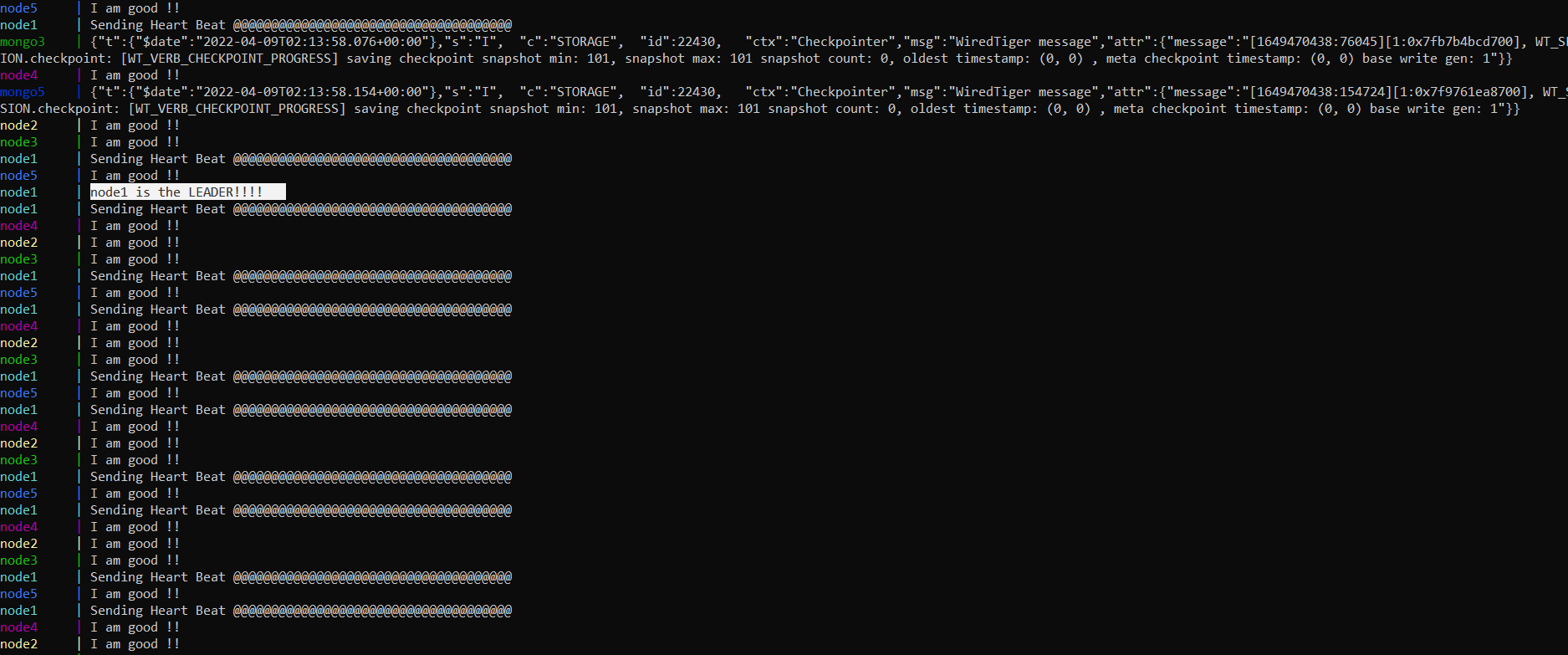
****

Heartbeats response: “I am good”

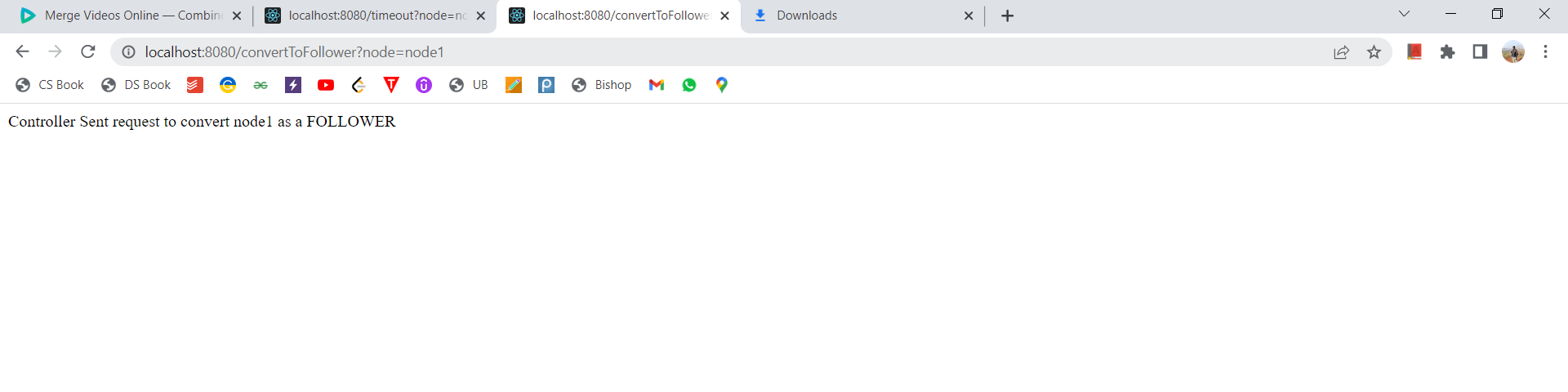


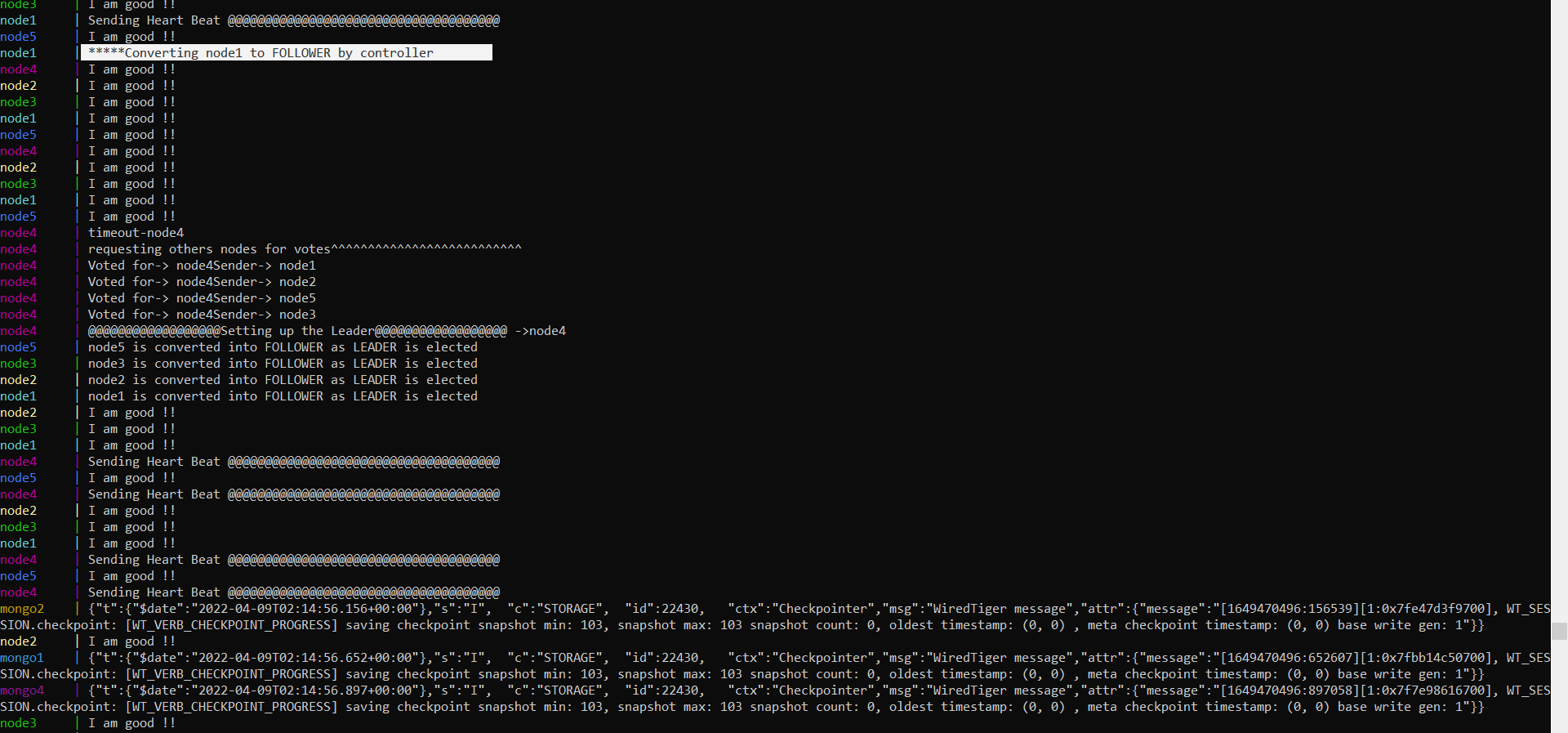
Getting leader info through API call:





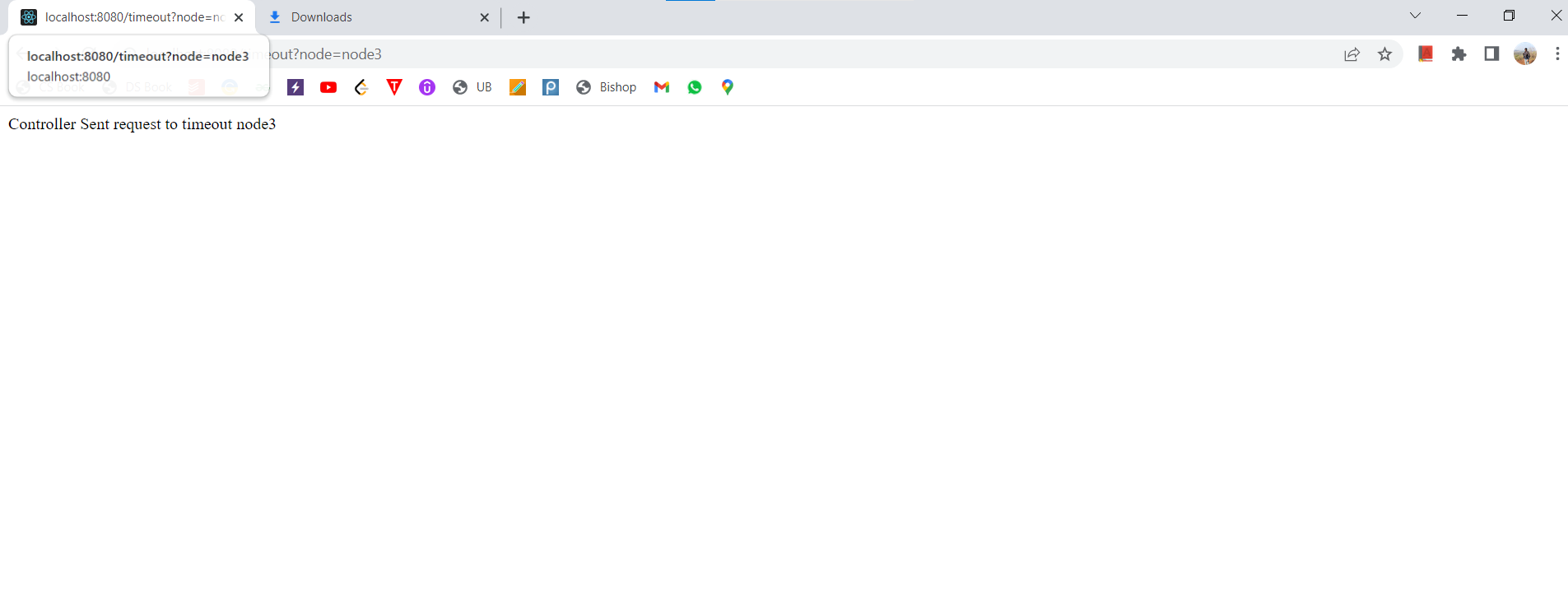
Making leader as follower:



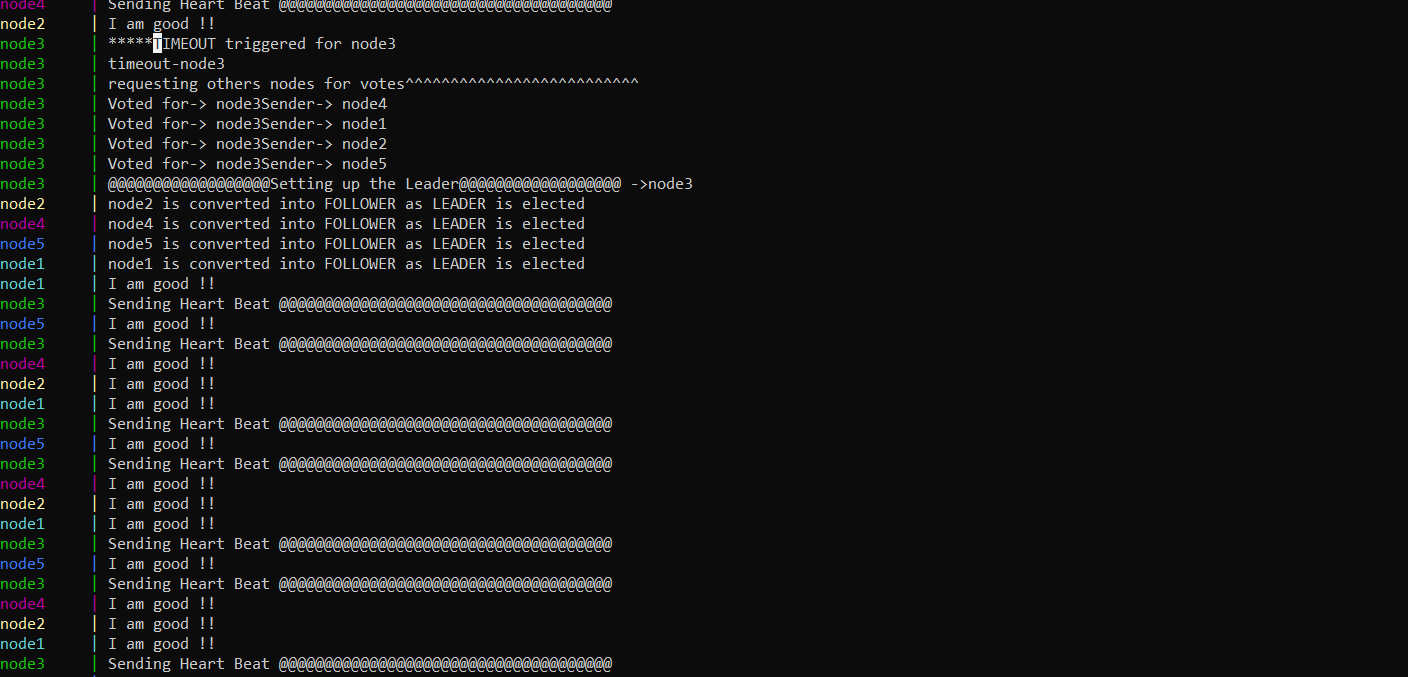
****

From the above, we can see that leader node 1 is converted to the follower. Hence a new election is being called and the process is repeated.

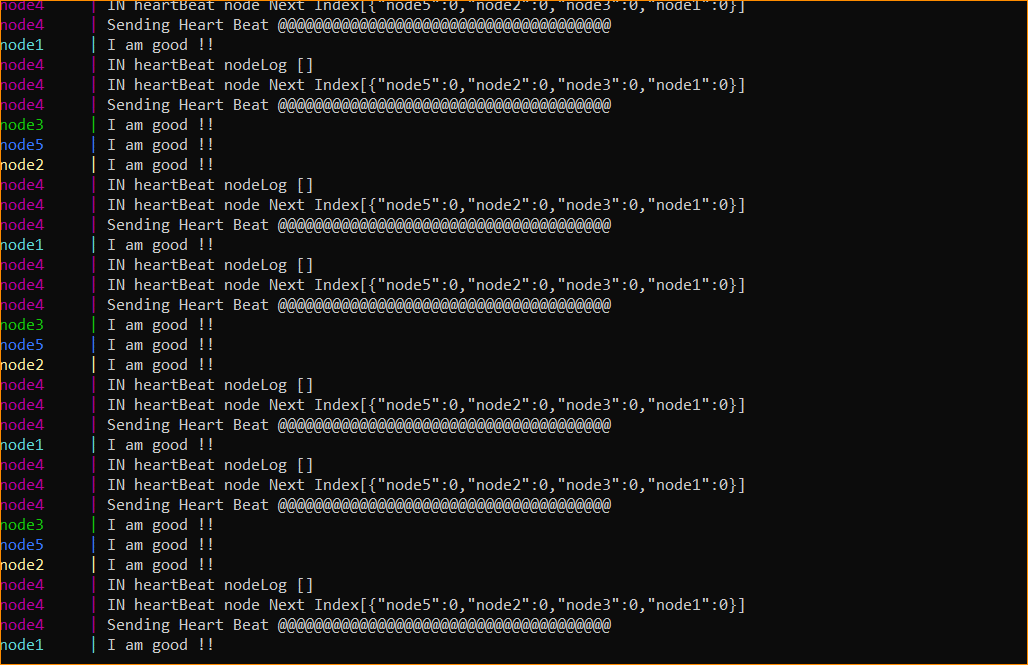
Making node3 timeout.



Hence a new election was called and the same process is being repeated.

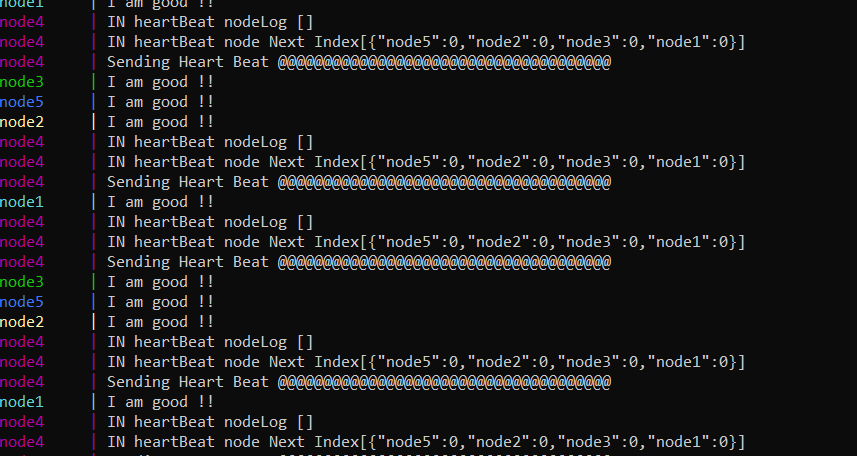


Lets store a key value via leader node, here cuurently leader node is node4:

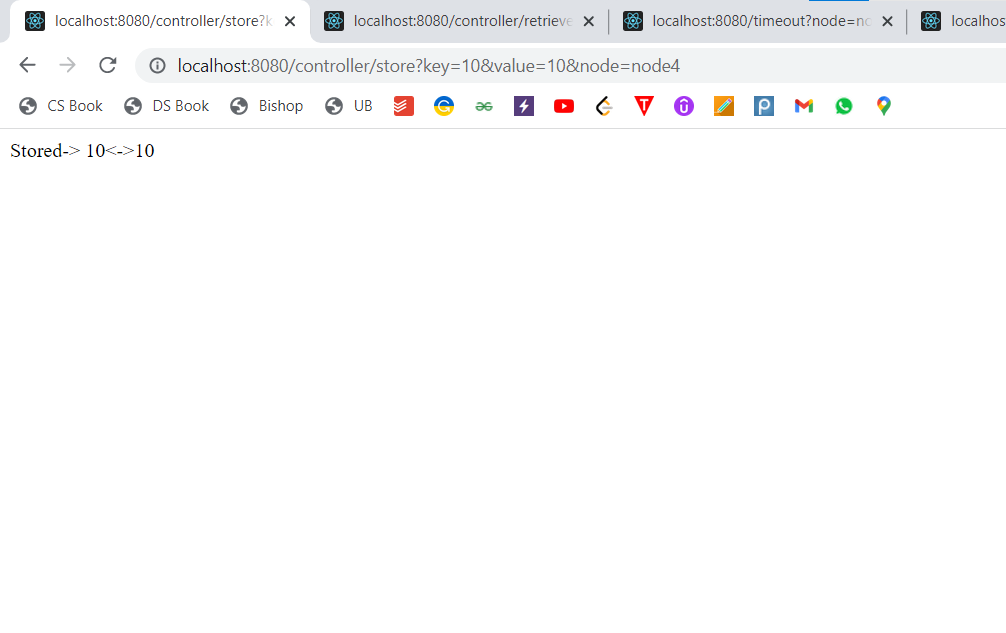


Lets send a store request via node4, since node4 is the leader:

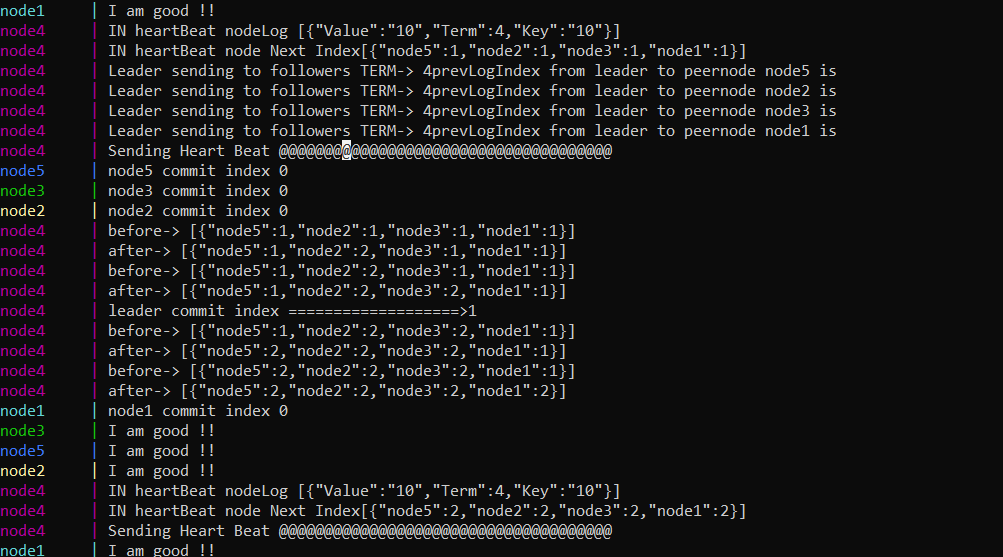
Initially :



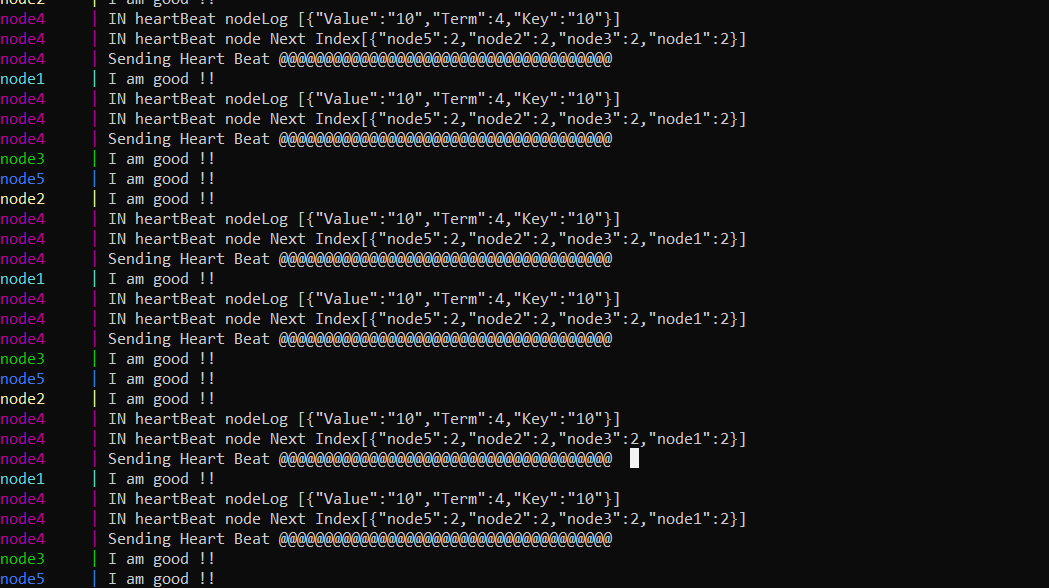
Request submission:



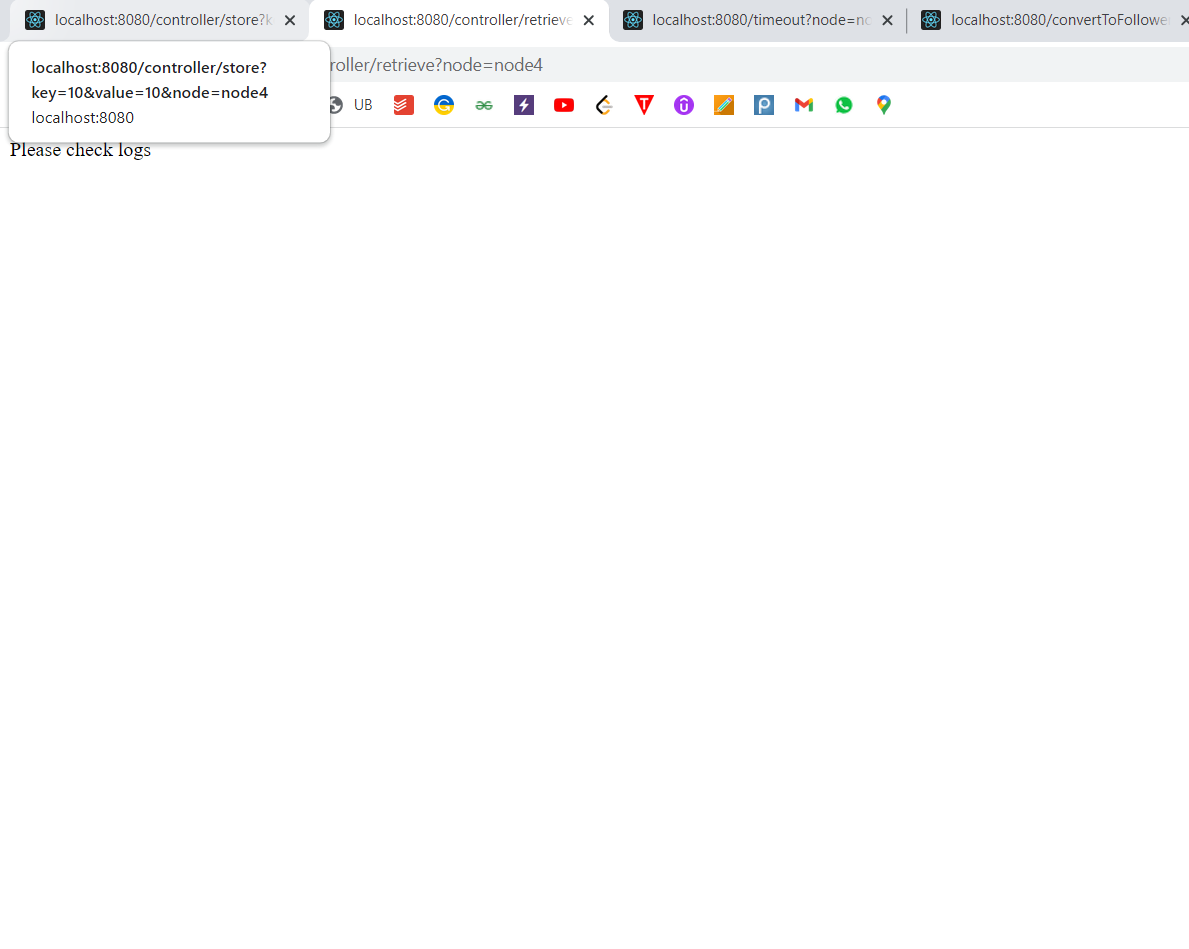
Commit Index intially:

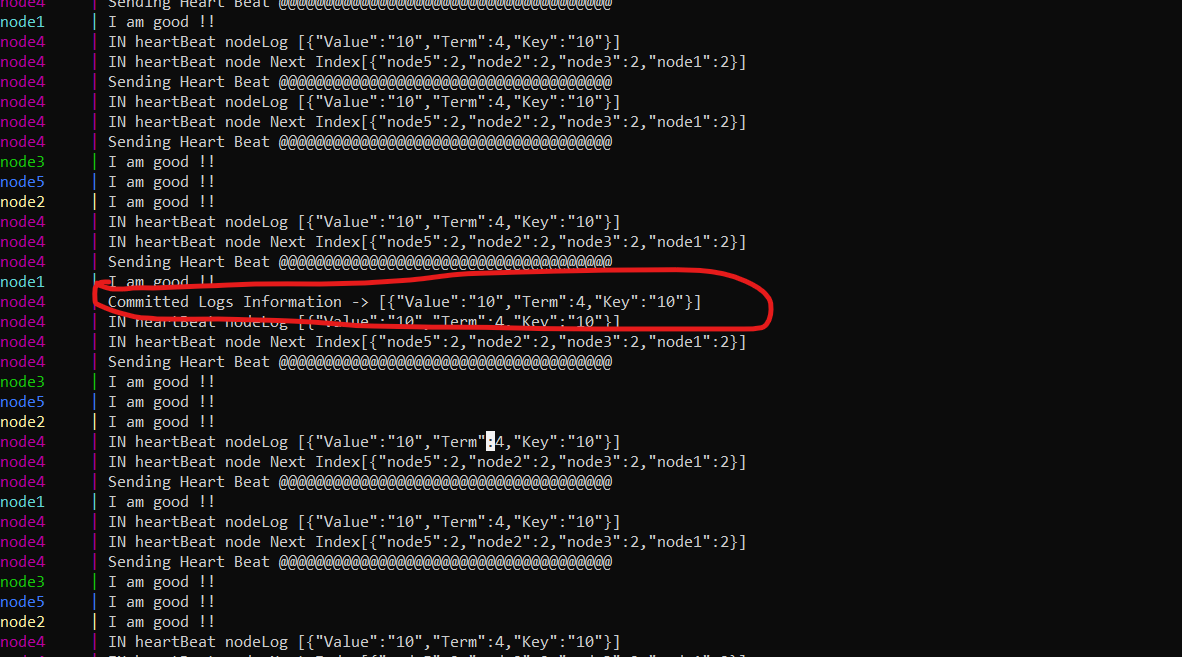


Log updated:

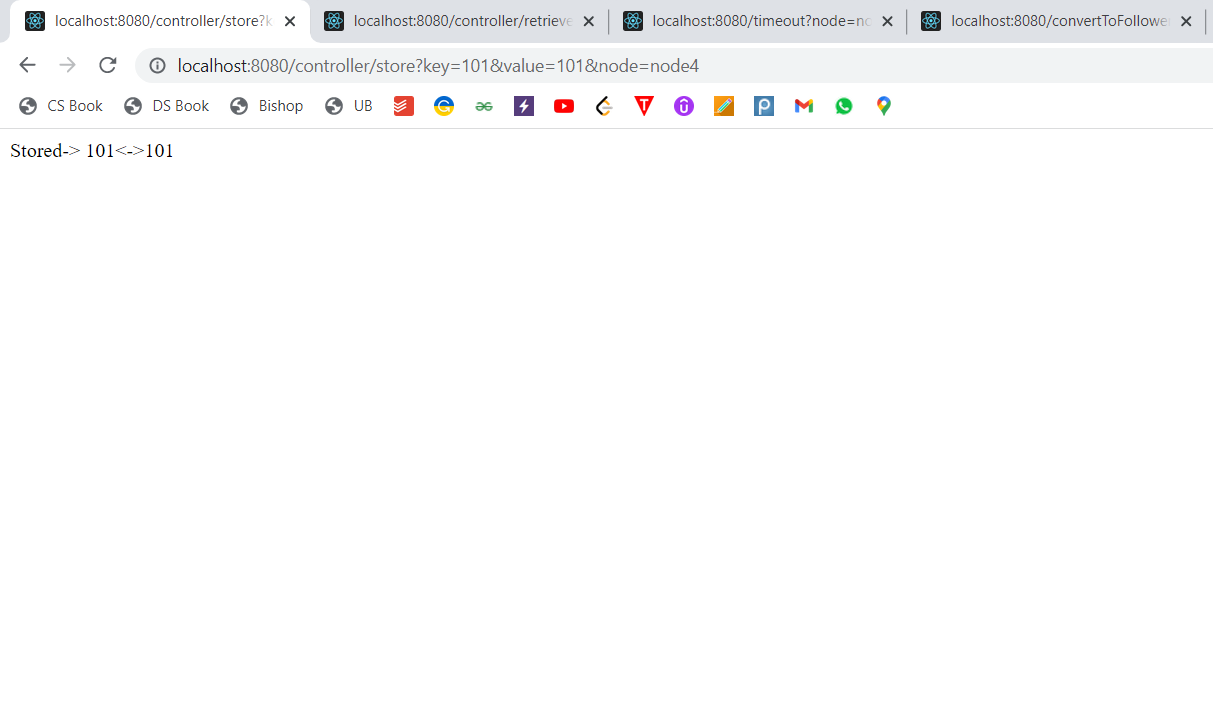


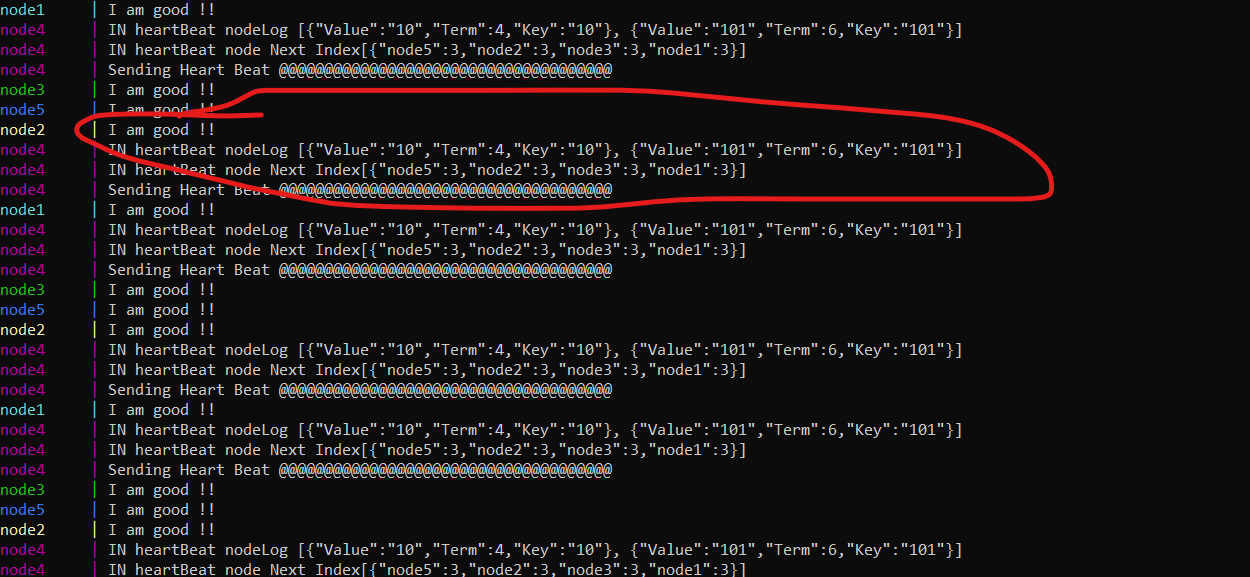
Log retrieval via node4, since node4 is the leader:



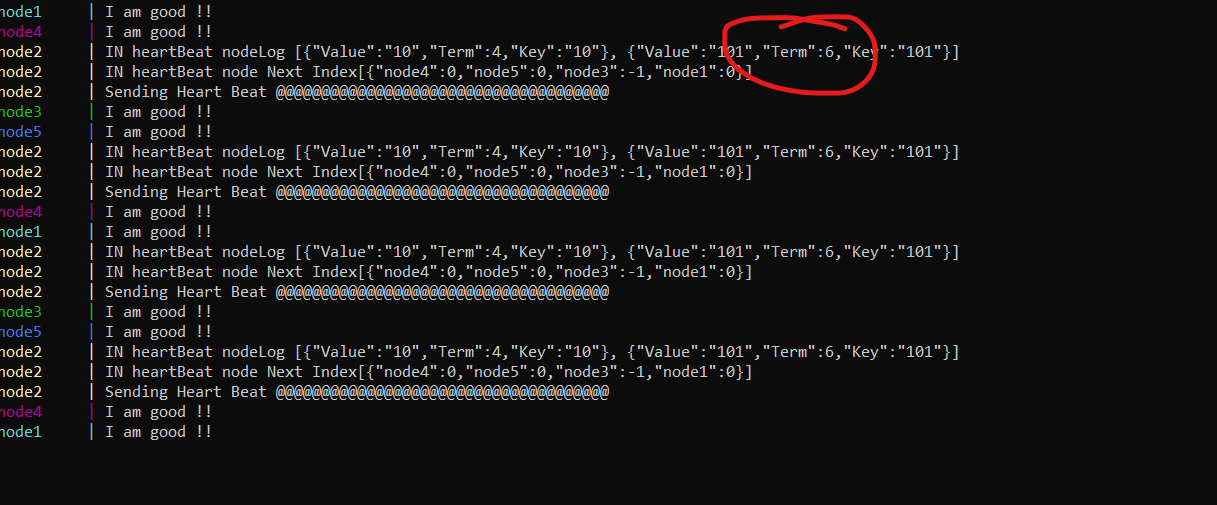


Store once more:

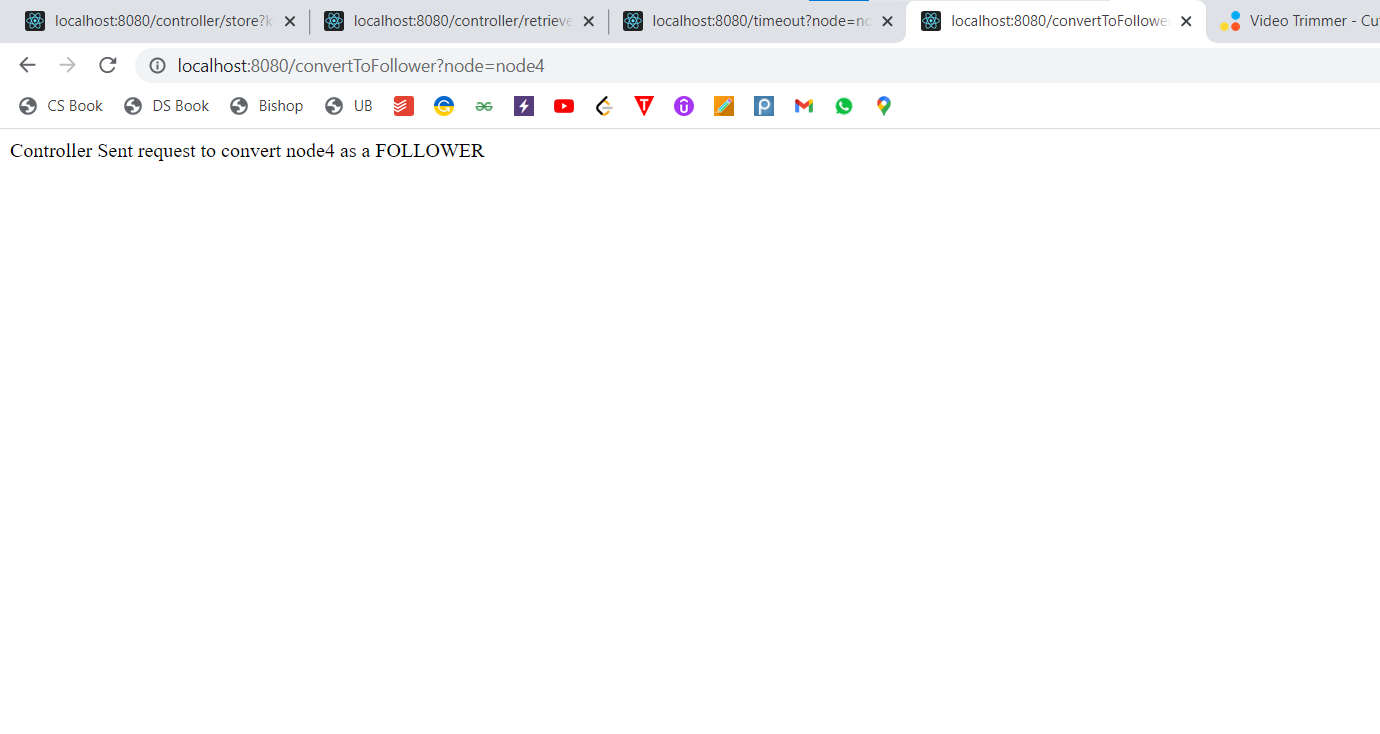




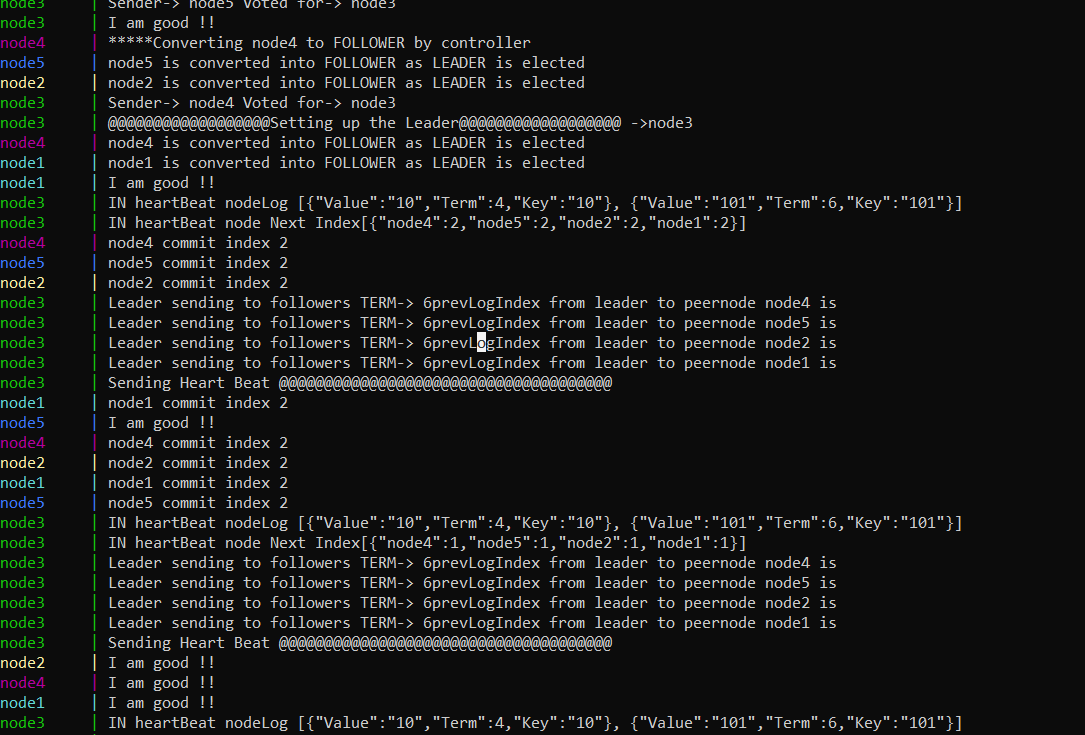
Current term:



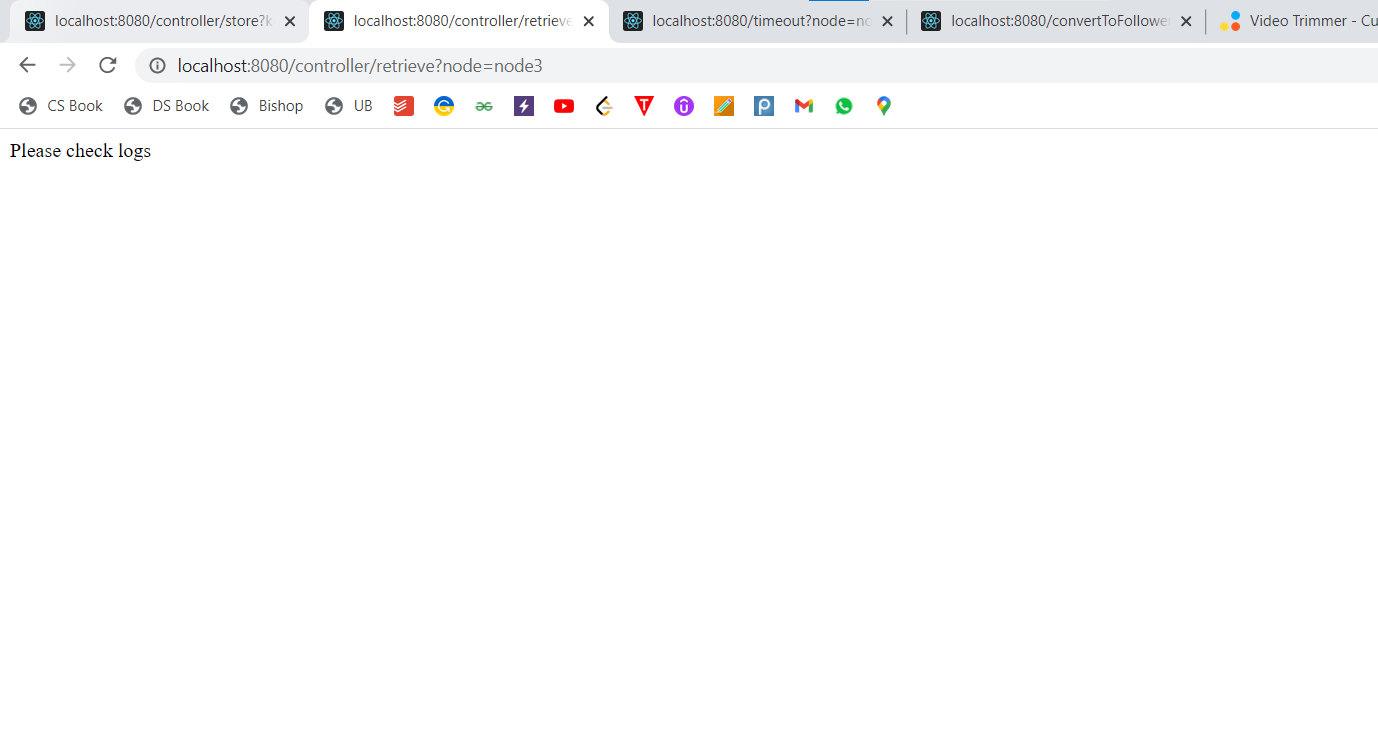
Converting node4 to follower:

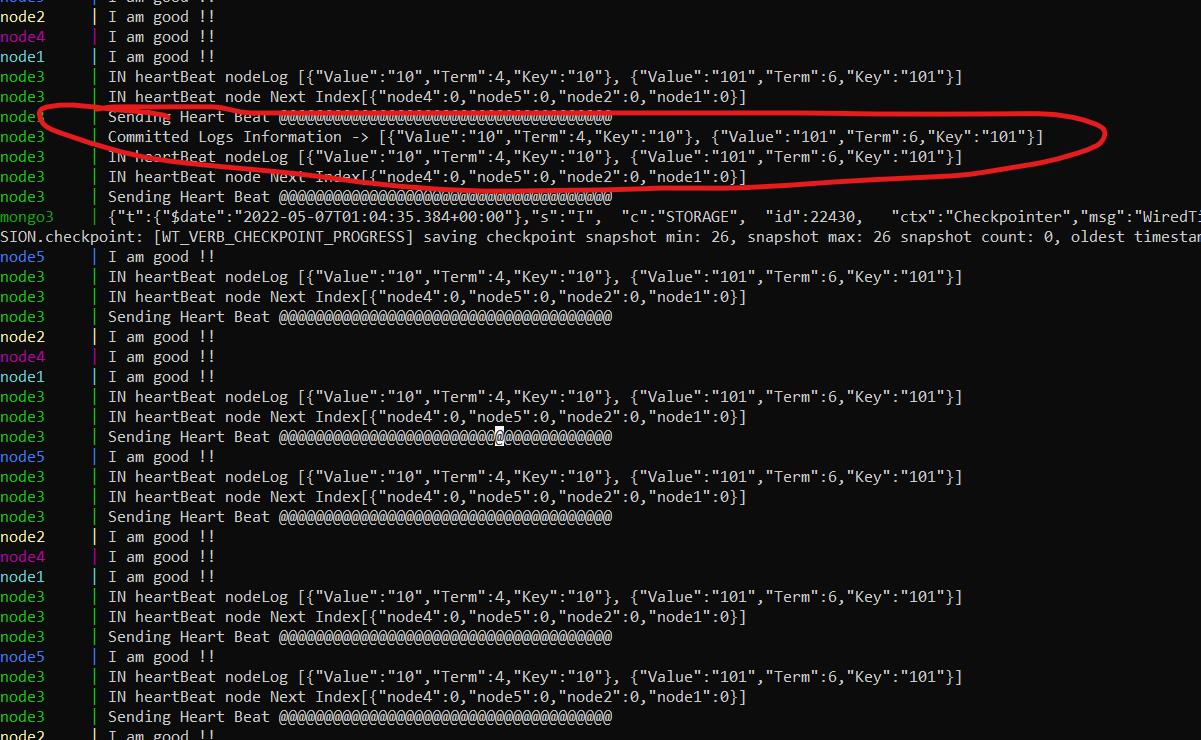


Node3 is elected as leader:



Retrieve request via node3:





**References:**

1. <https://docs.docker.com/>
2. <https://raft.github.io/>
3. https://raft.github.io/raft.pdf