INF3200: Mandatory Assignment 1

Ivan Kashilov, Michael Schöffmann

October 8, 2019

Abstract

Modern applications are often used by large numbers of users spread over the entire globe. These applications require various functionalities such as permanent availability, worldwide reachability and fast access. The basis of those kind of applications is always a powerful and effective distributed system. Distributed systems act in general as middleware and take care of underlying software and hardware. Even though the distributed system could run on a cluster of many individual computers, it aims to appear for the user as an application running on one single computer.

Introduction

This project includes design, conception and realization of a distributed data storage system as well as the subsequent evaluation of the implementation. The expected features of the distributed data storage are an annular arrangement of the storage nodes such that each node has two neighbouring nodes, one successor and one predecessor. All nodes can be considered to be permanently available during the system operation. Therefore the support of connecting and exiting nodes during the system operation is not required. The distributed data storage should be able to work with at least a minimum of 16 nodes.

Within the distributed data storage system, all nodes have to be able to serve incoming PUT and GET requests. Nodes which are not responsible for an incoming request have to forward the request to the responsible node by the use of routing information. This approach requires intrasystem communication between nodes to handle incoming requests.

The API of the distributed data storage system requires a minimum of three specific request calls to ensure compatibility with the client test script. In order to run tests on the system, a predefined client test script has to be able to send requests to any of the system nodes. The given request calls are PUT /storage/ <key>, GET /storage/<key> and GET /neighbours.

- **PUT /storage/ <key>**: Store data at given key
- **GET /storage/<key>**: Retrieve data from given key
- **GET /neighbours**: Return predecessor and successor of the node

Design

This section gives an overview of our distributed data storage design concept including the description of the basic concept, the architecture, the handling of requests and the implemented algorithms.

The approach for our implementation of the distributed data storage is mainly based on the Chord [1] protocol. This protocol provides a distributed data storage concept with a high level of scalability which is essential for large internet applications. Chord basically uses consistent hashing to map a key to its responsible storage node. The consistent hash function assigns each node and key an m-bit identifier using SHA-1 as a base hash function. [1] Each node is responsible for a certain range of keys and data is only stored at a node which is responsible for the key of the data. This enables to balance the load almost equally between all nodes of a system. To guarantee lookup costs of $O(\log N)$, each node has its own routing table with its successor nodes such that the distance between node and successor node is a power of two. Besides the lookup cost of $O(\log N)$, in case of topology changes only the lookup table of $\log N$ nodes has to be adapted. Both of these properties are necessary for availability and scalability. These aspects are very important characteristics for large distributed systems in order to ensure reliability.

We used a number of selected aspects from the Chord protocol to realize our implementation of a distributed data storage system, though there are deviations from the original Chord protocol. In the following sections we explain the details of our implementation and reasons why we decided on certain deviant concepts.

Architecture

As mentioned before, our system is based on the Chord protocol. Nevertheless, there are more or less distinctive differences between the Chord protocol and our distributed data storage implementation. We use the identical system topology as the Chord protocol with the difference that our system topology remains consistent during the system operation. For this reason, we decided to disregard changes of the system topology during the system operation, for instance joining and leaving of nodes. This decision had serious impact on following further design decisions and implementations.

Under this condition, once the distributed data storage system is operating, there is no necessity to manage the system topology. Therefore all lookup tables have to be initialized only once because there are no topology changes occuring during the system operation. These lookup tables are initially created during the creation of the nodes and remain unchanged until the system operation stops.

Another difference in comparison to Chord is that our system is not using hashing for assigning an identifier to a node. All nodes within the system get an id during their generation and the keys are statically assigned to them, that means one node can be responsible for one ore more key identifiers.

This brings us to the creation of the key values. In contrast to the Chord protocol, we use the MD5 message-digest algorithm to create key values out of given keys. The relevant part for load balancing of the storage is the last nibble or the last hexadecimal character of the resulting MD5 hex string.

Communication

Our distributed data storage system requires two types of communication, intersystem and intrasystem communication. While intersystem communication only occurs when the system receives a request from a client, intrasystem communication deals with the communication between storage nodes in order to handle the request and to direct it to the node responsible for the requested key. Thus, the intrasystem communication is a more challenging issue than the intersystem communication.

In our system we use intersystem communication only for the case when a client sends a request to any of the nodes in our system. This is for instance part of our test script where we send PUT and GET requests to any of the nodes. At some point the corresponding node will send an answer to the request.

The intrasystem communication in our system works similar as the intersystem communication. Any node that received a request from a client and is not able to serve the request by its own, forwards the request to a successor node that is closer to the requested key. At that node the same process as before begins again. When the node is able to serve the request, it will send the response back to its predecessor. This procedure has to be sometimes done iteratively to serve any kind of request from any node.

Lookup Algorithm

A very important aspect of distributed data storage systems is the time we need for lookups. Very intuitive implementations provide lookup costs of $O(N)$. The approach in this case is usually to walk from one node to its successor and so on until we find the node with the required key. That results in increasing lookup costs when scaling up a system and therefore it is not suitable for our implementation.

In order to achieve lookup costs considerably lower than $O(N)$, we decided to use finger tables with routing information. Here we had various options for providing routing information. The most intuitive approach is a full peer-to-peer system which gives us lookup costs of $O(1)$. However, the use of peer-to-peer routing information implies significant disadvantages when it comes to scalability. As soon as it would be possible that nodes are able to join or leave the system during operation, the expenses for maintaining the routing information on every node in the system would be costly.

Based on these considerations, we decided to implement the lookup based on the Chord protocol. This gives us two important advantages concerning lookup costs and scalability. On the one hand we achieve lookup costs of $O(\log N)$ and on the other hand we would still be able to extend the system during operation without changing all of the finger tables. This would help us to keep our system expandaple for further features at a later stage.

As already mentioned in the previous section, all types of communication are realized with standard HTTP requests. In combination with so called finger tables we can assume the following lookup procedure. If a node is not able to serve a request from the client script, another request is sent to the node in the finger table that is responsible for the requested key or is located most close to the node with the suitable key. When the suitable node was found, the node will respond to its predecessor from which it got the request. If this is not the client, it will be another node that processed and directed the request. This process causes higher effort for serving requests and therefore higher latencies for single requests. In addition to this point about lookup procedure, we have to mention that our system is incapable for multi-threading.

Experiments

In order to evaluate the performance of our distributed storage system implementation, we did several tests and experiments regarding the throughput of PUT and GET requests. The test setup consisted of our implemented distributed data storage system and a client script for sending requests to our system.

The tests consisted of different types of requests and various sizes of the test set. Small test sets with only 50 PUT and 50 GET requests were only used for general testing and evaluation of the functionality. The statistical significance is not good enough to make a valuable statement about the throughput because of the fluctuating variance. Larger test set with 10,000 samples have better statistical significance and can therefore be considered as more valuable.

The following graph shows the throughput of our system which is given as the amount of transactions the system handles per second and the related number of nodes working within the system. This particular experiment was performed with an amount 10,000 requests, 5,000 PUT request and 5,000 GET requests. After each PUT request to key k, a related GET request to the same key k was sent. To model the test case more realistic, both requests are sent to random nodes of the system. The large amount of requests in this example ensures that the throughput converges towards a faithful value. The calculation of the throughput is done by the calculation $throughput = 1/(time/number\ of\ requests)$

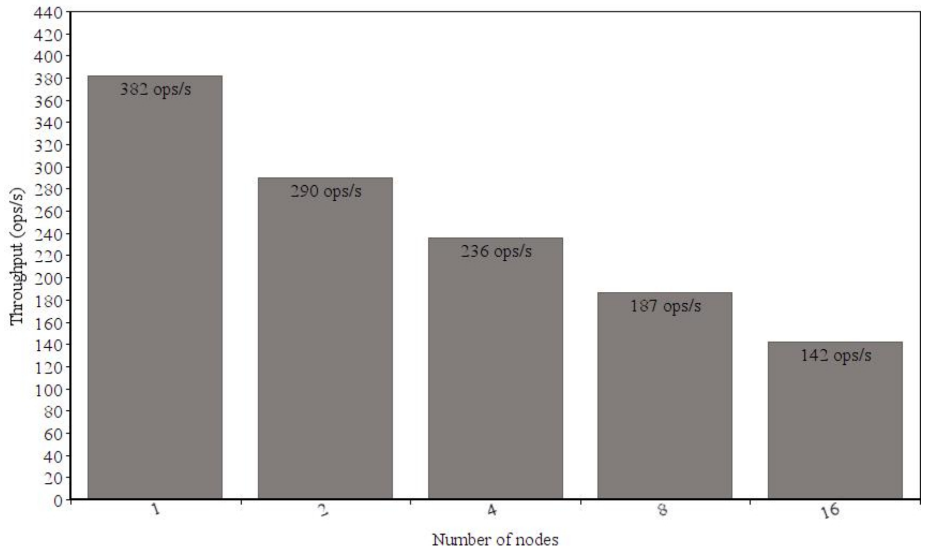

Figure 1: Throughput measurements for PUT and GET requests

As shown in the graph above, the more nodes we use in our distributed data storage, the lower is the throughput of it. This is in general an effect caused by the fact that in some cases several subsequent requests are necessary to serve the initial request.

Discussion

The tests we explained in the previous section about experiments are a representative evaluation of our distributed data storage system and give indications about positive and negative aspects. It is to mention that the latency of systems with a smaller number of nodes is better than of systems with a higher number of nodes. This is a result of the additional requests we need to send when we have to find the corresponding node for the key and also go back via each predecessor until we arrive at the initial node.

Most of the tests concerning our distributed data storage system have a focus on the performance measurements for instance transactions per second. This shows that the performance is an important characteristic of a distributed data storage system as well as many other functionalities which are not possible to measure with comparable values.

To summarize this section, we can say that there are several improvements that have to be done in order to make our system scalable and more reliable. The direct delivery of a response to a corresponding request from the serving node to the initial node or to the client instead of tracing back would result in lower latencies. In addition to that, the implementation of multi-threading would strengthen this enhancement supplementary. If we would consider our distributed data storage system as error-prone, we would have to implement appropriate mechanisms to solve issues concerning joining and leaving of node. In the end this would make our system more reliable with higher numbers of nodes. The general topology and the concept with finger tables is already a favorable basis for adding additional functionality.

Conclusion

We have seen that the design, conception and realization of a distributed data storage is a complex computer science project which requires various capabilities. At the same time it provides a lot of freedom when it comes to design, architecture and algorithms. Our system fulfills the given requirements and provides acceptable performance. Nevertheless, are some adaptions and suggestions to mention in order to to improve the performance of our distributed data storage system.

The ability to run multi-threading would increase the throughput of our system and make it more suitable for actual real-world requirements and also applicable to use cases. This is an actual bottleneck of our distributed data storage implementation and causes massive performance declines. Furthermore, direct responses from serving nodes to the client without tracing back to the initial node could enable significant latency declines already without multi-threading.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 17–32, 2003.