Milestone 3

Design Document

Qian Xuesen Group: Zifan Qu, Zhichao Lin, Zecheng Ding, Jie Hou

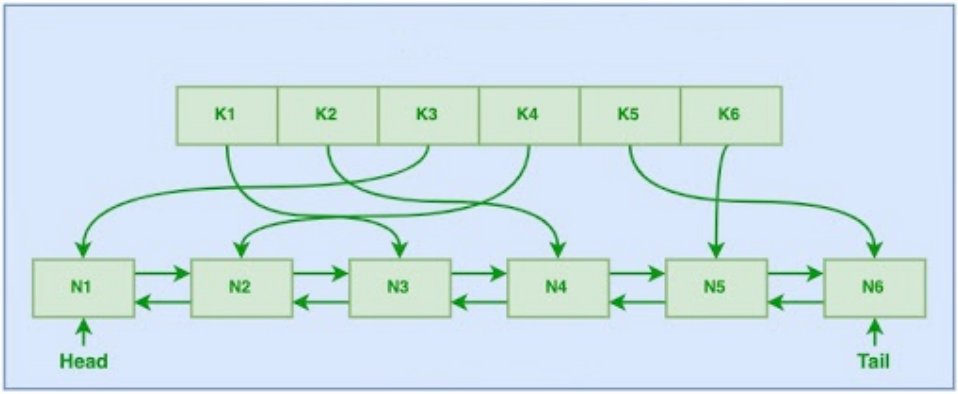
1 Abstract

The cache, which can improve the speed of reading instructions and data, has been widely used in the field of distributed computing with the expansion of local computer systems to distributed systems, and is called distributed cache. Distributed cache can handle a large amount of dynamic data, so it is more suitable for scenarios that require user-generated content such as social networking sites in the Web 2.0 era. After expanding from local cache to distributed cache, the focus of attention has been extended from the difference in data transmission speed between CPU, memory, and cache to the difference in data transmission speed between business systems, databases, and distributed caches.
The distributed system we will design is managed and controlled by a server, and multiple client nodes store data, which can further increase the data reading rate. So when we want to read a certain data, which node should we choose? If we find each node one by one, the efficiency is too low. Therefore, it is necessary to determine the storage and reading nodes of the data according to the consistent hash algorithm. Based on the data D and the total number of nodes N, the hash value corresponding to the data D is calculated through the consistent hash algorithm, and the corresponding node can be found based on the hash value. Therefore, the entire system includes the following 7 main components:

2 Project Design

2.1 LRU Replacement Strategy

The LRU replacement strategy is shown as following:



There are two data structures in the LRU.

1. The rectangles string on the above is a “Map” which stores the mapping relation between the keys and the values. With the storage advantages of map, we can search a specific value as keys and add a record into the map with little time cost.

2. The list below is a “Double Linked List” storing all the values in it. The crucial function of LRU replacement strategy can be achieved with the double linked list. If a specific value has been searched, then we move this block into the end of the list. Therefore, if a new record is entered, then the least recently used block can be removed from the cache, which guarantees the crucial functions of LRU. Moreover, the storage style of double linked list can also save a great amount of time cost if we want to delete or add a new record in this data structure.

The Go programming language has provided the convenient implementation of map and double linked list, so the main object in this part is to achieve the functions in the picture above with Go programming language.

2.2 Single-machine Concurrent Cache

About the concurrent cache in the single machine, we should address several problems as following:

1. Mutex

   In Go programming language, if several goroutines write or read a variable at the same time, the conflict will happen. We should guarantee that there is always one goroutine can access this variable. We use the Mutex to solve this problem. The Go programming has a convenient model about Mutex, and we should achieve this function with it.

2. Concurrent Write and Read

   In order to achieve the concurrency of writing and reading in the single machine, we are supposed to use Mutex to encapsulate some functions of LRU.

3. Intersection

   The distributed cache aims at constructing the reliable intersection between the devices and the clients. After receiving the requests of clients, we should check whether the object has been stored in the cache or not; if there is no avaliable values in the cache, we are suppoed to determine where to search the objective values and return it. In this project, we will construct a structure to implement these functions with Go programming.

2.3 Consistent Hashing

The classic hashing approach used a hash function to generate a pseudo-random number, which is then divided by the size of the memory space to transform the random identifier into a position within the available space. However, the problem in a distributed system with simple rehashing—where the placement of every key moves—is that there is state stored on each node; a small change in the cluster size for example, could result in a huge amount of work to reshuffle all the data around the cluster. As the cluster size grows, this becomes unsustainable as the amount of work required for each hash change grows linearly with cluster size. This is where the concept of consistent hashing comes in.
Consistent Hashing can be described as follows:

1. It represents the resource requestors (which we shall refer to as “requests” from now on, for the purpose of this blog post) and the server nodes in some kind of a virtual ring structure, known as a “hashring.”

2. The number of locations is no longer fixed, but the ring is considered to have an infinite number of points and the server nodes can be placed at random locations on this ring. Of course, choosing this random number again can be done using a hash function but the second step of dividing it with the number of available locations is skipped as it is no longer a finite number.

3. The requests, ie the users, computers or serverless programs, which are analogous to keys in classic hashing approach, are also placed on the same ring using the same hash function.

We need to implement the following to make it work:

1. A mapping from our hash space to nodes in the cluster allowing us to find the nodes responsible for a given request.

2. A collection those requests to the cluster that resolve to a given node. Moving forward, this will allow us to find out which hashes are affected by the addition or removal of a particular node.

2.4 HTTP Server and Distributed Node

Our system implements management and control of client nodes that store data through a server. The overall system flow is as follows. After the distributed cache system receives the key, it will check whether the key has been cached in the system. If it has been replaced by a walk-through, the cached value will be returned. If not, it should be obtained according to the remote node to determine whether the remote node is There is the cache value. If the remote node cannot obtain the value, the callback function is called and the current value is cached in the system.
Therefore, based on the above system operation process, it can be known that each node in the system needs to realize communication between nodes, obtain node information, and it is a relatively common and simple way to establish a communication mechanism based on HTTP. If a node starts the HTTP service, then this node can be

accessed by other nodes. Go language provides the http standard library, which makes it very convenient to build HTTP server and client.
We refine the process mentioned above, and we select nodes through consistent hashing. The key obtained by the hash function is used to judge whether the currently selected node is the target remote node. If it is, we will access the remote node through the HTTP client, and the client will return the return value after successful operation. If the operation fails, it will be processed at the current node. According to this function, to design a distributed node, the node needs to access the server through HTTP request, and obtain data, and compare and judge the data obtained through HTTP request.

2.5 Prevent Cache Breakdown

For an existing key, when the cache expires, there are a large number of requests at the same time. These requests will penetrate the database, causing a sudden increase in the amount of database requests.This phenomenon is called cache breakdown, and it may occur in hot queries.
The above phenomenon is that multiple threads simultaneously query this data in the database, then we can use a mutex lock on the first request to query the data to lock it.At this point, other threads will wait if they can't get the lock, wait for the first thread to query the data, and then do the cache. Later threads come in and find that there is a cache, so they go directly to the cache.
In our project, we will use the singleflight library officially provided by the go language to solve this problem. singleFlight enables the cache to obtain source data for an invalid key when multiple concurrent requests are made, only one of them is executed, and the remaining blocks wait until the executed request is completed, and then the result is passed to other blocked requests to prevent breakdown effect.

2.6 Communication Using Protobuf

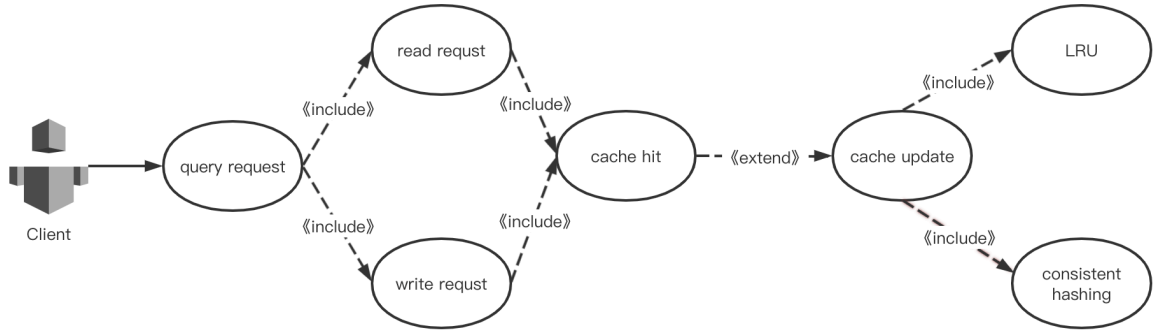
protobuf stands for Protocol Buffers, a data description language developed by Google. It is a lightweight and efficient structured data storage format, independent of language and platform, scalable and serializable. protobuf is stored in binary mode and takes up a small space.The purpose of using protobuf is very simple, in order to obtain higher performance. Using protobuf encoding before transmission and decoding by the receiver can significantly reduce the size of binary transmission. On the other hand, protobuf is very suitable for transmitting structured data, which facilitates the expansion of communication fields.
In our project, using protobuf is divided into the following 2 steps:

1. According to the syntax of protobuf, define the data structure in the .proto file and use protoc to generate Go code.

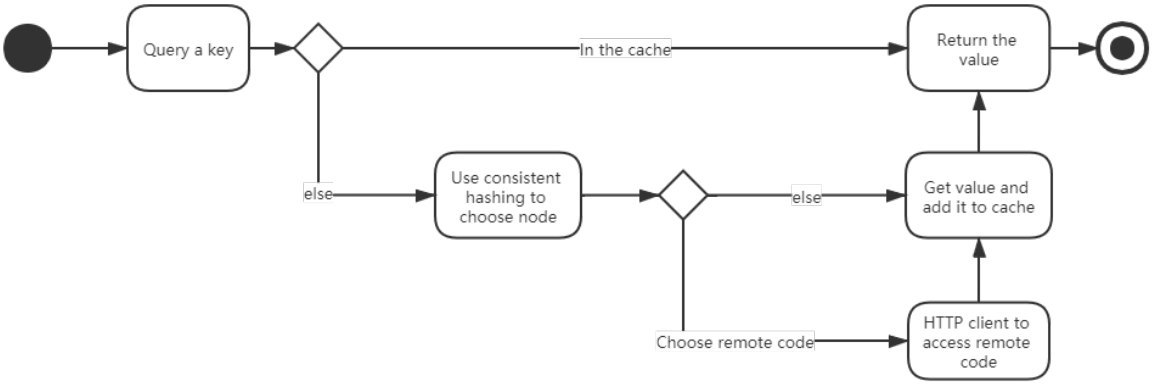
2. Quote the generated Go code in the project code.

3 Design Diagram

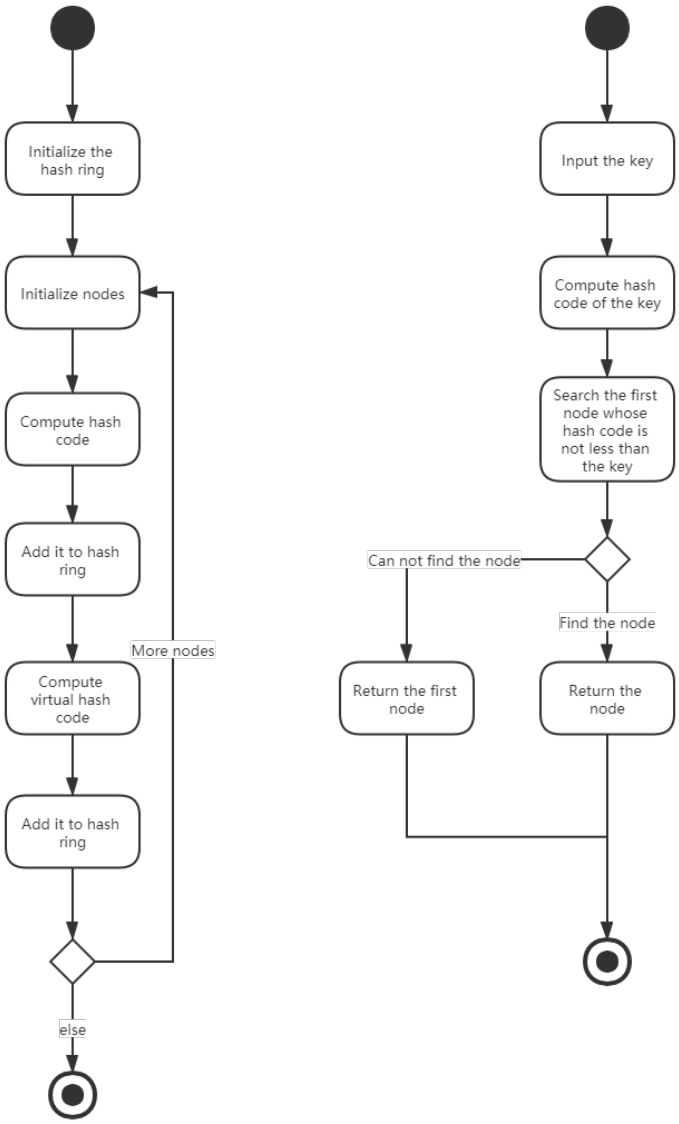
The use case diagram is shown as following:



The use case diagram has shown the intersection of clients and distributed cache system, including the possible requests of clients, different modules and functions in the cache. For the crucial wrok flow in the distributed cache, the activity diagram of our project will illustrate it and the diagram is shown as following:



In order to illustrate more mechanism of the consistent hashing, which is one of crucial technologies in out project, we draw two activity diagrams as following:



The left figure illustrates the process of initializing the hash ring; and the right figure is the flow of accessing the nodes.

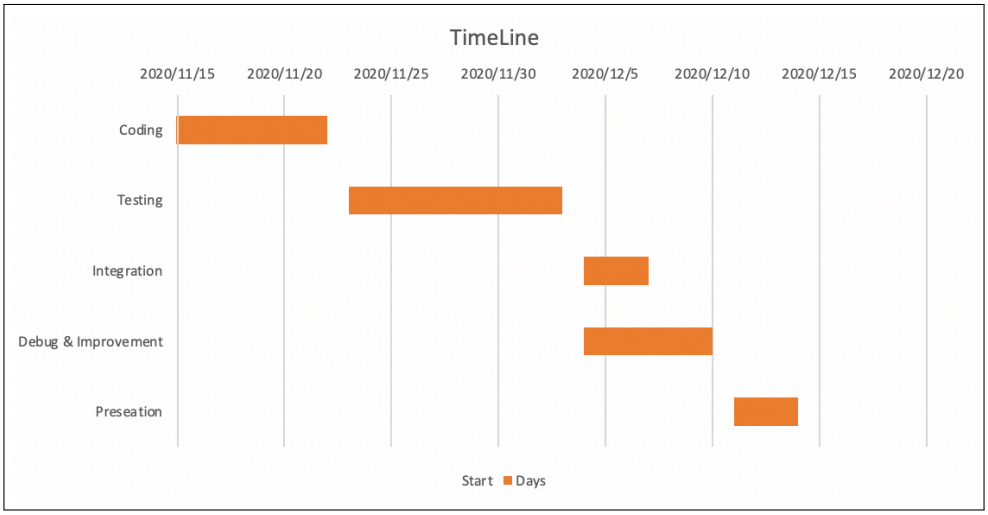
4 Project Schedule

4.1 Break Down and Test The Project

As mentioned in Part Two, we have divided the project into six parts. The goal of our project is to implement a distributed cache system, so the core goal is to verify the operability and correctness of the system. According to the project design, we need to test several major modules of core functions. For testing HTTP server and distributed nodes, we could test if each nodes could communicate with server properly and get the needed infomation. When testing cache breakdown, we can initiate multiple concurrent requests to the server, and then output the number of times the server executes the request. If only once, it can effectively prevent cache breakdown.

4.2 Timeline

There are still 6 weeks left for the presentation at the end of the semester. We will divide our task into the following parts. We spend most of our time writing the core module code. It is expected to take 2 to 2.5 weeks. After that, we will use 1.5 weeks for module testing, and then integrate the code of each student. Debug and improve. The remaining time is to prepare the final presentation of the project. The Gantt chart is shown as following:



In our project, Jie Hou is responsible for LRU and single-machine concurrent caching. Zhichao Lin is responsible for the communication between http server and node. Zecheng Ding is responsible for cache breakdown. Zifan Qu is responsible for consistent hashing.