Tiny EPC

Deadline: Sunday, 8th December 2019 at 11:50 PM

In this assignment you will be implementing a miniature, trimmed down version of a 4G/LTE packet core. As part of this assignment, you will implement consistent hashing and a protocol for state synchronization.

Abbreviations:

- **UE:** User Equipment, also referred to as mobile phone.
- **EPC**: Evolved Packet Core, the core network of the 4G/LTE cellular system. You will implement a tiny version of EPC in this assignment.
- **MME**: Mobility Management Entity, is the storage/processing node in the EPC. Tiny EPC consist of one or multiple MME nodes.
- **LB**: Load balancer, maps user requests to a MME. Tiny EPC consists of a single LB.

Note: MME and Node are used interchangeably in this handout.

Note: course policy about **plagiarism** is as follows:

- Students must not share actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection.
- Students cannot copy the code from the Internet.
- Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Note: You can work on this assignment with a partner who is also in the class and submit the assignment in pairs or work individually. In case you submit the assignment as a pair, please make sure when you submit code files, names and roll numbers of both students are written. Please note that both the students should be able to explain any program code or documentation they submit.

1. Introduction

The purpose of this assignment is to implement a scalable and consistent key-value storage system for a cellular network. To design such a system, we will use two key ideas covered in class; consistent hashing (for load distribution) and state replication (for fault tolerance). As we have already seen in the course, consistent hashing is widely used for data partitioning and

data replication in key-value storage systems. To review consistent hashing, please revisit the relevant lecture slides on LMS. The second part of the assignment deals with state replication, which is widely used for fault tolerance in many large distributed systems. The key idea is to back up state on another node so that in case of failures, the state can be recovered quickly.

In this assignment, you will implement a miniature trimmed down version of a 4G/LTE packet core, which we will refer to as Tiny EPC. Cellular packet core is an important component of cellular networks that connects the base stations (cell towers) with the rest of the cellular network and the Internet. It consists of multiple servers with different responsibilities. The Tiny EPC you will implement consists of a load balancer and multiple processing/storage nodes, called MME. The load balancer maps user requests to a particular MME based on consistent hashing. MME nodes hold the state for the specific users they are serving at any given time. Whenever MME receives a user request from the LB, it updates the user's local state. For fault tolerance, it also replicates the updated user state on its replica nodes. At any given time, a user has a single primary MME and one or more backup MMEs.

Figure 1 shows the high level design of Tiny EPC. We then explain the role of each entity; User Equipment, LB, MME, and the entire workflow.

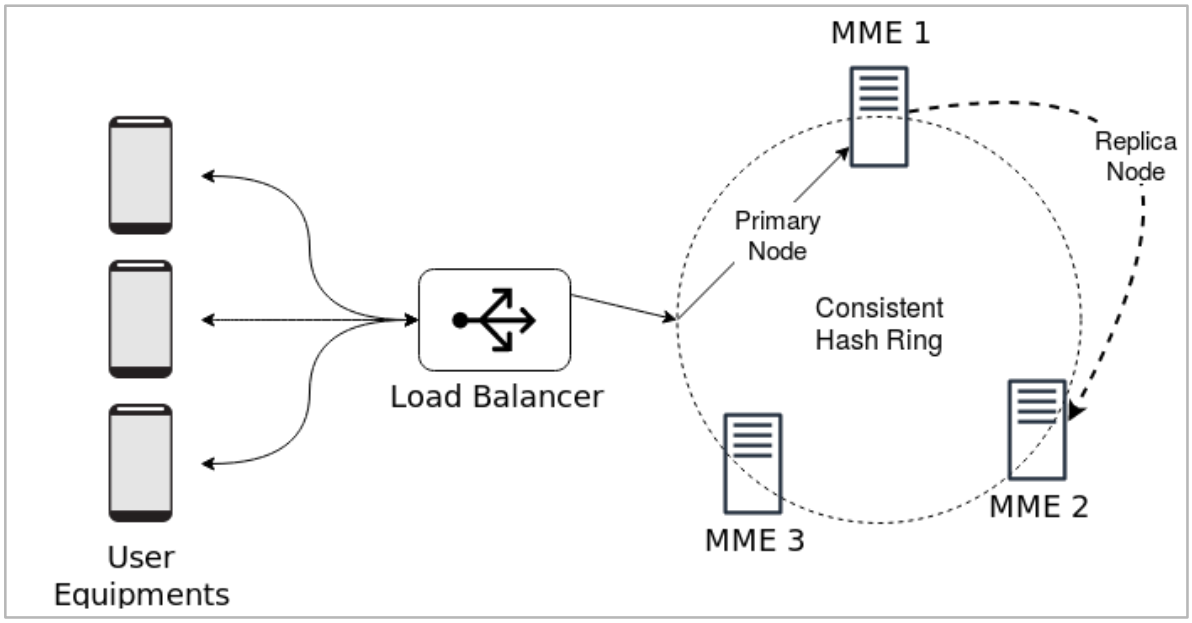

Figure 1: Tiny EPC - overall design

- **User Equipment (UE)**

A user equipment simulates a mobile user which can perform multiple operations at EPC (i.e. SMS and Voice Calls). In this assignment, user operations are limited to SMS, CALL and LOAD (explained later in this document).

- **Load Balancer (LB)**

The load balancer works as a proxy between the users and the MME. It receives user requests (with ID of the user as a key) and send it to an appropriate MME based on consistent hashing. Users interacts with the load balancer through RPCs.

The load balancer also communicates with the MME through RPCs. Any MME can join or leave the load balancer at any time. When a new MME comes up, it sends a join request to the load balancer which adds it to the hash ring. To simulate the MME failure scenarios, **`RecvLeave`** RPC is called which removes MME from the hash ring.

- **Mobility Management Entity (MME)**

MME serves as a processing node in the Tiny EPC. The first node on the ring, in the clockwise direction, from the hashed position of the user ID is called its primary node. A typical MME holds up to more than 130 parameters per user. But since we are implementing a tiny version of the MME, we have confined it to only one parameter which is the net balance of a user. Whenever a user arrives at the MME for the first time, a default balance (of 100 units) is added to the user state. User can send three different types of request to an MME:

1. SMS (charges 1 unit)
2. CALL (charges 5 units)
3. LOAD (award of 10 units)

State replication can follow synchronous and asynchronous model in general. But, for this assignment, you will be implementing synchronous model only, by which MME will only respond back after the user state has been updated on primary as well as on backup replica.

- **Tiny EPC Workflow**

To understand the working of all the components, following is the list of steps performed in sequence:

1. Load Balancer listens to the user requests and MME **`Join`** messages.
2. Each MMEs send a **`Join`** request to the load balancer at startup.
3. Load Balancer assigns replicas to the new node.
4. Load Balancer updates the replicas of the affected nodes due to addition of new MME.

5. Load Balancer rebalances the hash ring due to the addition of a new MME (key relocation).
6. MMEs perform the state migration as per the load balancer updated keys relocation.
7. User generates a user request and sends it to the Load Balancer.
8. Load balancer extracts user ID from the user request, hash it, embed hash into the user request, and forward it to an appropriate MME based on consistent hashing scheme.
9. MME serves a user request based on the type of request as follows:
    a. **SMS:** deducts 1 from the user's balance
    b. **CALL:** deducts 5 from the user's balance
    c. **LOAD:** adds 10 to the user's balance
10. MME replicates the user state on the backup replica.
11. MME respond back to the load balancer with its MME ID.
12. Load balancer forwards the received reply back to the user.
13. Keys relocation is performed similarly to that in Step 5 when MME leave request is received.

2. Requirements

As you write your code for this project, also keep in mind the following requirements:

1. You must work on this project individually or **ONLY** with your partner. You are free to discuss high-level design issues with other people in the class, but every aspect of your implementation must be entirely you and/or your partner's own work.
2. You may use any of the synchronization primitives in Go's sync package for this assignment.
3. You **MUST** format your code using gofmt and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.

3. Important Points/Hints

1. Use the hash functions provided in *loadbalancer/consistent.go* for calculating physical and virtual node hashes.
2. State of each MME is replicated on the first neighbor in the clockwise direction on the ring (replication factor 1). Note that:
    a. It becomes tricky when virtual nodes are present on the ring. In the presence of virtual nodes, keys belonging to one physical node may be replicated on more than one physical nodes.
    b. Replica node may be a virtual node but should not be from the list of virtual node belonging to primary node.

3. Think about what will happen if a new MME joins the load balancer? How keys will be relocated? How MME will get to know about its neighbors changed? How MME will know about what range of keys will need to be relocated? Such cases can also occur when MME leaves the load balancer.

4. Evaluation

You can earn up to 25 points from this assignment. There is no extra credit or bonus.

The following table describes each test case along with its maximum marks:

Test Description	Marks
Test 1: Consistent Hash Ring	**Total: 1**
Test 2: Load Distribution (a) With no virtual nodes (b) With medium number of virtual Nodes (c) With high number of virtual Nodes	**Total: 3** 1 1 1
Test 3: Keys Relocation (a) With no virtual nodes (b) With new MMEs joining in between (c) With MMEs leaving in between	**Total: 8** 2 3 3
Test 4: Replica Assignment (a) Single MME (b) Multiple MMEs (c) Multiple MMEs with virtual nodes (d) One MMEs leave in between (e) Multiple MMEs leave	**Total: 7** 1 1 2 1 2
Test 5: State Synchronization	**Total: 1**

Code formatting and documentation (total 5 points)

We will manual grade on the style of your code in this project. This means you should have **well-structured**, **well-documented** code that can be easily read and understood. For example, you should not have dead code and you should not write a lengthy function. You have received feedback from us for previous assignments on what is considered good style. The Effective Go guide is a great resource to look at should you have any questions.

Also note the following:

- Race conditions will be checked.

- You have a total pool of 5 days for late submissions, without deductions. Late days from this pool can be used for any of the four assignments. Once you have used up all the 5 days, no late submissions for the assignments will be accepted.

5. Logistics

You will need to implement the following files:

1. *mme.go*
2. *loadbalancer.go*
3. *consistent.go*
4. *uerunner.go* [ungraded - it is up to you to implement this]
5. *protocol.go*
6. *rpc.go*

These files contain definition of multiple APIs which you need to implement. Please see comments on each API for further detail.

We have provided a test pack along with the starter code. You are encouraged to write some of your own tests as well to check the functionality of your implementation. It will be up to you to thoroughly test your code before submitting on LMS.

Starter code

The starter code for this assignment is contained in the folder **tinyepc**/. The code is organized as follows:

```
├── loadbalancer              // Folder contains implementation of the load balancer
│   ├── consistent.go         // Consistent hashing implementation
│   ├── loadbalancer_api.go   // Load balancer interface
│   └── loadbalancer.go       // Load balancer implementation
├── mme                       // Folder contains implementation of the MME node.
│   ├── mme_api.go // Interface definition for MME APIs implementation
│   └── mme.go                // MME implementation
├── rpcs                      // Folder contains RPCs interface and messages
│   ├── protocol.go           // Messages definition
│   └── rpc.go                // Helper APIs
├── runners                   // Folder contains runner apps for different modules.
│   ├── lbrunner
│   │   └── lbrunner.go       // Runs load balancer
│   ├── mmerunner
│   │   └── mmerunner.go      // Runs MME
│   └── uerunner
│       └── uerunner.go       // Runs UE (user equipment)
```

Folder placement:

Place **tinyepc/** in your **$GOPATH/src** directory.

Test application uses the three runner applications to test your implementation.

- ***The lbrunner program***

  The *lbrunner* program creates and runs an instance of your *loadbalancer* implementation.

- ***The mmerunner program***

  The *mmerunner* program creates and runs an instance of your *mme* implementation.

- ***The uerunner program***

  The *uerunner* program creates and runs an instance of the UE.

Test application retrieves results from the MME and LB through RPCs. You will have to implement the following RPCs (details about the input and output parameters is provided in the code comments):

- At MME, you will have to implement the **`RecvMMEStats`**
- At Load Balancer, you will have to implement the **`RecvLBStats`**

Executing the test cases

A binary executable *test* is provided with the assignment package. The *test* application can be executed from anywhere within the system.

To run all the tests, run the following command:

```
$ ./test
```

To run all the tests with **race** condition run the following command:

```
$ ./test race
```

To run a single test with all its sub-tests, run the following command:

```
$ ./test <testcase number>
$ # Example
$ ./test 2
$ # Example with race condition
$ ./test race 2
```

To run a sub-test, run the following command:

```
$ ./test <testcase number><sub-testcase>
$ # Example
$ ./test 2b
$ # Example with race condition
$ ./test race 2b
```

If you have other questions about the testing policy, please don't hesitate to ask us a question on Piazza!

6. Submission

You are only required to submit all contents of the **tinyepc/** folder. Please make a zip file, name it using the format **<R>.zip**, where **<R>** should be replaced by your roll number (e.g. **17030010.zip**) and submit it on LMS.