1. Introduction (125 words)

We implemented an asynchronous request-reply model to build a file sharing system EZShare.

1. Scalability (375 words)

For a file sharing system like EZShare, scalability was reflected by ensuring the effectiveness of the system while amount of resources and amount of real-time traffic were increasing.

* 1. Resource Storage

We identified resource storage as an issue that constrained the scalability of the system. Currently, we used a HashMap to store all the resources in memory, which limited the amount of resources that could be maintained by a server. And once the server was closed, the resources saved by this server were lost.

Thus, we chose NoSQL database as a solution to this problem. For one thing, all the resources were in form of JSON, which was a perfect fit for NoSQL database like MongoDB and CouchDB, where one JSON could be saved as one record. It would be much harder for us to query or remove a resource if we applied relational database because according to the normalization principles, we would have to separate the data for one resource into several tables. For another, NoSQL database would be easier to apply for partitions and distributed designs, while this would be harder for relational databases.

* 1. Real-time traffic

We identified two important issues caused by large amount of real-time traffic that may affect the performance of the system. Firstly, our system adopted the ‘thread-per-connection’ principle, which meant the more clients connected to the server, the more thread would be used, thus the more memory amounts would be consumed. Secondly, if the real-time traffic was of high load, the locks used to ensure the concurrency would result in unacceptable waiting time, which was also a factor that limited the scalability of our system.

One possible solution for the first issue was to implement an ‘event-driven’ model.

【江一明要查下资料】

The solution for the second issue would be discussed below in concurrency part.

1. Concurrency (375 words) 【有时间的话之后画几张图】
   1. Concurrency Issues

We identified two major concurrency issues for our EZShare System: write-write conflicts and read-write conflicts.

Write-write conflicts represented for the situation where two clients tried to publish, share, remove, or modify the same resource at the same time. For example, if two clients wanted to publish a resource with the same uri and same channel but a different owner, they would possibly both get a successful response, which was not supposed to be allowed by the server. This was because when both threads called the publish function and checked the existing resources, they did not find any duplication with the resource to be published, so they would both publish the same resource as a legitimate resource.

Read-write conflicts represented for the situation where one client queried the server with read authority, while the other client modified the queried resource right after the query, which let the previous query no longer effective. For example, if one client used query command and obtained a resource uri as ‘file://dog.png’ and tried to fetch the file later, while the other client finished updating the uri of this file to ‘file://puppy.png’ right after the query, the fetch command with uri ‘file://dog.png’ from the former client would fail.

* 1. Potential Solutions

A straight-forward solution to concurrency issues was to lock the shared resources each time it was used, no matter for reading or writing. Although this would thoroughly solve the above issues, there would be an unacceptable waiting time under high traffic load and the availability of the server was in some way sacrificed, and as we mentioned before, the scalability could not be ensured either.

An alternative solution was to lock the shared resources only for writing operations such as publish, share and remove. This method would avoid the ‘write-write’ conflicts but still have some problems. On one hand, the ‘read-write’ conflicts still existed and might cause the inaccuracy of query. On the other, the waiting time could not be shortened if lots of clients tried to use the writing commands.

Therefore, a better way to solve the concurrency problem as well as to ensure the accuracy of query and acceptability of waiting time would be locking on a smaller basis. We may partition the shared resources to several parts, and block each part only when the operation required using this part. This way, we may only need to block one part at a time for a client, while the other parts were free to be used. The smaller one part was, the less waiting time client may need to spend. However, there was another tradeoff between the size of resource chunks and storage overhead or query overhead, and we could not put each resource in a new table. Furthermore, we may add a load balancing function for utilizing different parts of the resources. For instance, if function A wanted to query resources part I, which was locked by function B, then instead of waiting in queue for that part, the load balancing function would recommend function A to use resources part II first.

1. Other Challenges (375 words)
   1. Consistency

Consistency was one the issues that our system did not handle very well. In other words, we did not ensure the consistency of application logic and the consistency of resources across different servers. For instance, the server should not allow the same resource to be published twice, but if a client published a resource on one server and another client published the same resource on another server with the same primary key, both operations would succeed because the servers did not check the resources saved on other servers while publishing. However, when the client tried to query across different servers, they would get the duplicated results.

One solution to this problem was to keep a centralized database that all servers could get access to. But a centralized database design was not easy to extend, and we may sacrifice scalability and partition tolerance.

According to the CAP theorem, we ensured the availability and partition tolerance for our system by storing the resources in a distributed manner, it would be impossible to ensure consistency at the same time. Nevertheless, we may ask servers to exchange their resources when there was less traffic from clients to check whether there were any resources breaking the logic rules and to make revision. By doing so, the resources saved on all servers would eventually become consistent with each other.

* 1. Reliability and Failure Model

Although we had considered the situation of server failure and had implemented the timeout mechanism on client side and the periodically exchange function on server side, there were still some challenges regarding system failure modeling.

One of challenges was that we could not distinguish a server failure from the network crash. For instance, when server A periodically exchanged server list with server B and found server B could not be connected, server A would consider server B as disabled and removed server B from server list, while in fact the connection failure was due to the instability of network.

A possible solution for this problem was to test more times later after one-time failure, which meant we would not consider a server to be broken only after one time of failure. We may keep the unreachable server on server list until we tested three times at different time with the same failure result.

* Increased Response Time
* Repeated Application Errors
* System Unavailable

1. **Conclusion**
2. **Reference**

[1] Textbook

Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, Edition 5, ©Addison-Wesley 2012

Distributed systems : concepts and design第一章challenge节Failurehandling部分最后二段。。电脑里只有中文版的textbook\_(:зゝ∠)\_

**Appendix**