Distributed Systems
Project 1 - Report

Team name - Yew
Alexandra Fritzen (afritzen 866403)
afritzen@student.unimelb.edu.au
Annie Zhou (azhou 356076)
a.zhou@student.unimelb.edu.au
Borce Koteski (bkoteski 696567)
bkoteski@student.unimelb.edu.au
Hugh Edwards (hughe 584183)
hughe@student.unimelb.edu.au

1 Introduction

This report discusses EZShare, a distributed file sharing application written in java. Following a brief explanation of the challenges and outcomes of the system is a discussion of scalability, concurrency, and security. For each of these issues, shortcomings and improvements will be explored.

The system is based on the client-server model, where servers make resources available to clients. 6 commands are available to allow for the sharing and management of resources in channels.

Writing Classes which strictly adhered to the required protocols was a significant challenge, and required the creation of several generic JSON methods. Appropriately acting on every possible command also provided a challenge both in terms of programming and testing.

In testing our final system on a variety of cases as per the specification, all actions were correctly completed within a reasonable time frame. Our java classes are also modular and extensible.

2 Scalability

Scalability affects EZShare in three key categories; users, servers, and resources.

The current server implementation can handle a significant number of client requests. However, if too many clients attempt to access a server simultaneously, the requests will not be added to the server's queue and will be dropped. This is because the server can only allocate a certain number of threads at a given time to handle requests, and its incoming connection queue can only hold a certain number of requests. Hence for the system to be able to accept more users, the server itself must be scaled (in terms of available threads or queue size), or more servers must be added. If more servers are added, an algorithm must be used to divide load between servers. This could be implemented by clients transparently selecting random servers for queries, replicating resources over multiple servers, and/or having centralised routers for distributing requests evenly.

Server scalability is another issue. If a very large number of servers exists, performance issues may arise. Performance of the EXCHANGE command is not likely to deteriorate with the addition of servers; however, it is likely that changing server availabilities will take longer to propagate throughout the system. This may reduce the correctness of the QUERY command (and hence resource availability) when relay is activated, since servers are more likely to have incomplete or incorrect server lists. Furthermore, having more servers increases the likelihood of delays or errors resulting in timeouts with the relay command. Hence the response to the query may be incomplete. These issues may be reduced with the addition of more complex algorithms to maintain and update server lists[1], and the replication of resources over multiple servers. Whilst this may result in temporary inconsistency of resources between servers, it would increase availability, which is likely more of a goal for file sharing applications.

Resource scalability is yet another consideration. Our program's resource lookup could be improved, since a linear scan is performed to

[1] For example, when a server's list changes, it might immediately send one or more EXCHANGE commands.

find the correct file in the FETCH command. For scalability, hashes should instead used to lookup resources in constant time. Additionally, increasing the number of resources, or their file size, may cause servers to simply run out of storage. Further, large resource sizes may be more likely to result in failure if a connection is lost during a FETCH command. Large numbers of files may also cause issues with namespacing - that is, conflicting URIs of files. To reduce these issues, file sizes and the number of files shared by individual users may be limited. Secondly, total server storage must be scaled accordingly with the volume of resources hosted.

3 Concurrency

EZShare is designed to allow multiple clients and servers to issue commands to each other, potentially simultaneously. As a result, concurrency issues arise firstly in allowing concurrent client connections, and secondly in handling the resulting consistency issues.

To address multiple client connections simultaneously, the Server class instantiates multiple threads. However, any server has limited memory, storage, and processors, and hence the number of threads is inherently limited. When this limit is exceeded, commands are dropped, and the server cannot begin any new connections with clients. Hence, as discussed in section 2, a large number of concurrent users may cause the server to be overloaded and unable to serve all clients. This may be remedied by increasing the number of threads available on the server (which may require increasing server resources such as memory), or by increasing the number of servers and distributing the load between them (as discussed in section 2).

EZShare uses the ConcurrentHashmap class to maintain consistency during concurrent client interactions. This Class is used to maintain both resource and server lists. It is a reasonably robust solution to the problem of consistency; sequentialising access and thus preventing the resource or server lists from being left in an inconsistent state due to multiple threads operating on them simultaneously.

The use of ConcurrentHashmaps may increase the time taken for operations, since bottlenecks may form around the Hashmaps (as threads must form a queue to modify or access the hashmap). This may be seen as an acceptable tradeoff in order to ensure consistency of resources. It may, however, reduce availability due to timeouts threads being taken up by threads waiting for resource access.

One may consider availability as more desirable than consistency in this application, since the consequences of inconsistency are likely to be the incorrect listing of resources and servers - the impact of which is not likely to be disastrous to users. In this case, one may replicate resources over multiple servers. Such a system might be only eventually consistent (since changes to one server take time to propagate to another), however, have increased availability. Such a system would also exhibit a greater failure tolerance. Hence with concurrency, a tradeoff is faced between availability and consistency. This tradeoff is fundamental to distributed systems (Cavage 2013: 87, Bailis & Ghodsi 2013).

4 Security

EZShare presents four primary security concerns; the insecurity of the channel/owner system, lack of encryption, malicious file sharing, and the vulnerability to denial of service attacks.

Firstly, the channel/owner system is insecure. Many channels or owners might be named with easily ‘guessable’ names such as ‘Channel1’ or ‘John’. Secondly, there is no limit on the number of queries with incorrect channel/owner combinations other than a maximum of one request per second per client. Hence it would be easy to ‘brute force’ these secrets by repeatedly sending queries with random combinations of

channels and users every second. Furthermore, there is no enforced time limit on channel and owner names; hence attackers have an effectively unlimited amount of time to crack these secrets.

In response to this vulnerability, it is recommended a 'password' field be added, which must match the channel and user, and is required to be at least 8 characters and contain sufficient diversity of symbols, to make cracking more difficult (Ma et al. 2010: 586). Incorrect queries from specific IP addresses should also be limited in a period of time. A time limit may also be set after which passwords must be changed. These measures would reduce the risk of unauthorised resource access.

Secondly, data is unencrypted. This poses a security risk in that channels, owners, and data are transmitted as raw JSON strings. Since these are human readable, any unauthorised party who gains access to a message could easily view and abuse this data. It is recommended that encryption is applied to messages using the RSA public/private key system to reduce the risk of middleman attacks or unauthorised access. Passwords should also be stored in 'hashed' form in case of data leaks.

Thirdly, files may be shared which contain malicious mobile code (ie. a virus). Files shared in the public channel are especially vulnerable, since they may be shared by strangers, and hence may result in users unknowingly downloading malicious code. Files should thus be scanned by the server for viruses.

Fourthly, the system has a limited number of available threads, and hence is vulnerable to denial of service attacks. This occurs when an attacker floods a channel with requests to prevent other users from accessing the server (Lau et al. 2000). To reduce this vulnerability, the number of requests from individual IP addresses should be limited. Preventing *distributed* denial of service attacks is more complex and beyond the scope of this report (Lau et al. 2000).

5 Conclusion

This report has presented a number of issues relating to scalability, concurrency, and security. A system which takes into account these issues and implements the recommended improvements would have increased performance in each of these areas.

6 References

Bailis, P & Ghodsi, A 2013, 'Eventual consistency today: Limits, extensions, and beyond', *ACM Queue*, vol. 11, no. 3.

Cavage, M 2013, 'There's no getting around it: You are building a distributed system', *Communications of the ACM,* vol. 56, no. 6, pp. 63-70.

Lau, F, Rubin, SH, Smith, MH, Trajkovic, L 2000, 'Distributed denial of service attacks', *Systems, Man, and Cybernetics*, vol. 3.

Ma, W, Campbell, J, Tran, D & Kleeman, D 2010, 'Password entropy and password quality', *Network and System Security,* pp.583-7.