# COMP90015 Project 1 Report

In this project, a client-server resource sharing application was designed and implemented. Following the protocol specified, servers are able to host resource records including links and files, and clients are able to create, remove and query for the records by communicating with the server. The servers are also able to communicate with each other, to share server list information, or to make queries on resources. Multiple running clients and servers forms a distributed system.

Different types of challenges appeared during the development, in aspects such as scalability, concurrency and security. It appears that the system needs to be carefully designed and implemented, especially for uncommon situations. The system works as expected in the specification, with no significant issue in its performance. However, it is still questionable whether the system will be practically effective due to its limitations.

## Scalability

Scalability issues can be divided into three major areas.

The first area is number of requests from clients. As the number of connections increases, performance in each thread will decrease, which will eventually result in longer turnaround time for clients. Since each client is handled by a thread in our implementation, the number of connections is limited by the maximum number of threads the OS allows, and performance of each thread will be limited by CPU power and bandwidth. Therefore, better hardware and network are required in order for the system to scale up, which can be challenging. The performance can be slightly improved by adopting the worker pool approach. Instead of creating a new thread for every new connection where overheads are introduced, the threads can be created at initialisation and reused for different connections.

The second area is number of servers in the whole system, which affects exchange and query operations. Automatic exchange of server is currently implemented in a way that exchange requests to servers are sent sequentially. As number of servers increases indefinitely, the server may not be able to complete the exchange actions within the specified time interval. Using asynchronous requests or multiple threads can help, but the performance is still limited by bandwidth and CPU. When a server knows every other servers by exchanging, each query request from the client will result in one request to every single server because of the relay setting, which makes scalability with number of requests more important.

The last area is about the number of resources. When more resources are put into the server, all requests about the resources will take longer. For example, the query operation in our implementation has a O(n) time complexity, where n is the number of resources. Adopting more efficient algorithms to access the resources would also help, especially when the algorithm can be parallelised, so that multiple CPU cores, or even a cluster of computers, can contribute to overall performance of a server.

Many of the issues above can be addressed by having a distributed data structure. If storage of resource on different servers is well organised, less connections will be required to perform tasks (for example, query relay can be a single connection if a single possible server holding desired resources is known), and load balancing will be possible for client request and number of resources.

## Concurrency

Concurrency is naturally challenging to the EzShare system, because a server needs to run multiple actions at the same time. More specifically, it needs to handle requests from multiple clients, while doing regular exchange operations simultaneously. In our implementation, concurrency and its potential issues are mainly in the resource storage and the known server list.

The resource storage is accessed by multiple service threads, each handles a request from client. Query and fetch requests requires the storage to be read, while publish, remove and share commands correspond to write operations to the storage, concurrent operations on the storage then becomes a reader-writer problem: writer needs exclusive access on the resources, while multiple reading operations can happen at the same time. If a resource is removed when another thread is iterating over the storage, a runtime error is likely to happen, since the reading thread did not expect data to be changed. Currently, resource storage is implemented as a Java monitor object, where every operation is atomic (marked with "synchronized" keyword), so that only one of them can happen at the same time. It is safe to for multi-thread access, but its efficiency can be improved by implementing reader-writer lock (using semaphore for example), so that query and fetch commands can be executed concurrently. The system can also support more concurrent operations by having multiple lock granularities, so that only a necessary section of resources is locked during an operation, where other sections can still be accessed.

Concurrency for server list is similar, where servers are added to list by accepting exchange request in service thread, and servers in the list are read (and removed if not reachable) in exchange thread. The proposed changes above can also be applied to server list, and for both of the components, these changes can be applied without affecting the protocol.

Even the server support concurrent operations efficiently, they can cause issues externally from the perspective of clients. By the time a client receives results of a query command, the resources listed may have been modified or deleted already. A client can never have exclusive access to a resource, as long as its channel and owner names are not secret. Although private channel can be useful in this case, it could be helpful to support locks on resources between multiple commands from a client.

## Security

In addition to scalability and concurrency, security is another major challenge to the system. For secrecy and integrity, since all messages (JSON and file bytes) are transmitted without encryption or signature, critical information such as secret, channel and owner can be easily obtained and modified by the man-in-the-middle. If so, any security mechanism in the protocol will lose its effect. One possible solution is to adapt SSL/TLS as a middle layer of network component, similar to HTTPS. Using encryption helps secrecy, and digital signature helps checking integrity of messages. The change can be done independent to the protocol itself.

The security policy of the system can also be improved in different ways. Currently, a resource is made private by hiding its owner and channel names, which somehow acts as secrets when accessing the resource. However, the names may not be a good choice being the secret, since they can be very predictable, and more vulnerable to brute force attack using dictionary, even with the connection interval limit. It may be a good idea having a dedicated password field for private resources, preferably with some limitation on minimum complexity. (The limitation should also be applied on the server secret, if provided by the user through command line.) It may also be helpful banning clients with too many incorrect secret attempts.

Security can be even better enforced by disallowing user input secrets and regular refresh of the secrets, but it may lead to worse usability.

Compared to secrecy and integrity, availability of the current system is slightly better protected. Client requests from the same source are limited by the minimum connection interval, so that denial-of-service attack cannot be performed from a single address. However, a distributed DoS attack is still possible, and handling this issue can be difficult. Other than scaling the system up (with issues addressed above), mechanisms such as CAPTCHA, or some more sophisticated algorithms can help determining whether requests are from an actual user, but leads to usability and implementation issues.