# Heterogeneity

The chat server project deals with heterogeneity well. The choice of protocol (text-based JSON) provides a platform-agnostic source of communication and does not tie the server or client to any particular platform or programming language – either component could just as easily be written in C#, JavaScript or Erlang. The client has no and needs no knowledge about how the server is running – provided it implements the specification correctly, it will be able to use the services.

# Scalability

The chat server does not deal with scalability very well. Although additional servers can be added to handle multiple chatrooms, in order to converse in the same chatroom, all clients must be on the same server meaning there is an upper limit on the number of clients that can be in the same room. Furthermore, due to protocol design and implementation there are numerous operations that require communication with every single other server instance. While this is not an issue with just two servers, scaling to a thousand servers could potentially cause operations like creating identities or switching rooms take a very long time.

To mitigate this, potentially a central ‘state manager’ could keep a master list of identities and rooms and perhaps even use publish/subscribe to allow messages to be routed between servers, rather than having all clients belong to the same server to access one chat room.

# Failure Handling

Chat server does not handle failures at all well. In the event a server crashes or is rendered disconnected from other servers, it will not restart, nor be able to clean up its state (for instance, remove chatrooms that no longer exist from other servers). Clients will not automatically move to a different server, nor will other servers know the failed server is no longer available. This was out of the scope of the project, but still would be an important consideration were this project to become more substantial. There is no redundancy, no error detection, no error masking, no error toleration, nor any possible recovery. Any of these would substantially increase the complexity associated with the project.

# Concurrency

There are some provisions within the code for managing synchronization across threads, however I am not altogether convinced that it is not possible for a deadlock to occur. The concept of locking and unlocking identities and rooms does provide a good level of protection against this – at worst, two concurrent attempts to register the same room or identity would result in both being denied unnecessarily. This is a clever provision of the protocol. However, within the code, there are places where different threads access the StateManager singleton that could potentially result in unexpected behaviour. I have attempted to mitigate this by flagging the StateManager as requiring synchronous access, however am not sure how effective this will be, particularly at scale.

# Transparency

The system does exhibit some aspects of transparency. It provides access transparency – a client need only establish a connection to one server in order to access chat rooms across any – it has no idea that it might have transferred to a different server. Furthermore, there is complete transparency with regards to location, scaling (within the constraints highlighted above), replication (similar to access) and mobility.

There is, however, a lack of failure transparency. An error in the protocol is immediately displayed as a stack trace in the client and, depending on the nature of the failure, may render further communication impossible. Additionally, due to the requirement of uniqueness for usernames and chatrooms, there is arguable limited concurrency transparency.