Denobo

End-of-Project Review/Discussion

Chapter I – Development Methodology and Areas for Improvement

During this project, we used the collaborative development and version control system GitHub to allow each team member to contribute to the project codebase in small commits. The main advantage this offers is the ability to roll back any changes in response to problems or bugs that may emerge as the software is developed.

Code was written both collaboratively and independently as the responsibility for creating certain packages/class libraries was delegated to individual team members. While one team member is working on the encryption library, for example, another might be developing the testing UI or fine-tuning synchronisation.

Area for Improvement – JSON Packet Serialisation

A JSON packet serialiser would be a more suitable choice than the current format that uses nested query strings. While the query string-based serialiser is fit-for-purpose, the more open and standardised format of JSON would lend itself nicely to a potentially reflection-based packet serialiser.

Area for Improvement – Feature-Completeness of Testing UI

The Testing UI (Denobo Central Command) is missing a few features (saving/loading the state of the network designer, for example) that would be very convenient to have in a finished product. A more advanced, scroll-enabled network designer with a more accessible suite of debugging tools would also speed network building and testing.

Area for Improvement – More Efficient Route Calculation

The current route calculation algorithm in use by RoutingWorker is rather basic and there is potential for a large number of improvements to be made to it. For example, at the moment, route calculation is one-way. If a message is transmitted back from the recipient to originator the route is recalculated. For large networks, route calculation is an expensive process and more efficiency in this area would greatly increase Denobo’sr scalability.

Chapter II – Use of Design Patterns

At the heart of the Denobo software lie several recognisable design patterns, the use of which make Denobo highly maintainable and extensible.

Use of the Observer Pattern

Perhaps the most fundamental use of a design pattern can be seen in the way Denobo uses the observer pattern. By implementing the MessageListener interface, any class can register instances of itself as a message listener to any number of agents in order to be informed when those agents receive messages.

In this way, the internal workings of the agentware can remain well-hidden. Using the observer pattern saves the end-user from having to derive their own class from Agent to be able to receive and process messages it receives.

Use of the State Pattern

When Agents communicate over sockets, they use a specially-designed protocol called DMTP (Denobo Message Transfer Protocol). The session can at any time be in one of a number of different session states, affecting the types of packet that can be legally transmitter peer-to-peer.

Using the state pattern saves the peers at either end of the connection having to use excessively large and deeply-nested case statements by handling packet transmission on behalf of a socket-enabled agent and offering a more maintainable and less error-prone way to dictate which packets can and cannot be legally transmitted at any point in the session.

Use of the Factory Pattern

When a remote peer sets the compression mode on a local peer, it transmits the name of the compression mode over the socket. This name is passed into a simple factory method that returns an instance of a Compressor associated with that name.

Use of the Adapter Pattern

When an object wishes to implement the ability to observe messages from an Agent, it implements the MessageListener interface. The problem with this is the MessageListener has one method for getting notified about intercepting a message that could not be destined for the current subject Agent and a method that is invoked whenever a message is received whose destination was our subject Agent. If our observer only cares about messages that were destined for the subject Agent, it still needs to implement both which can clutter the code. We solve this by creating a MessageHandler object that implements the MessageListener interface for us. We can use this MessageHandler instance as an anonymous class and only override the methods that we require.

Chapter III - Threading

This chapter describes the use of threads within Denobo.

Agent

Denobo makes use of threads throughout its core. Every time an agent is instantiated, a thread is created. This thread is the message queue thread whose sole task is to wait for messages to be put into a BlockingQueue at one end by another thread and it will take it off and process it. The processing of a message consists of notifying any listeners about the message then it checks whether the current agent is the recipient of the message. If it is the recipient, the listeners are notified then the thread pulls or waits for the next message from the queue. If the message is not intended for the current agent, the thread looks at the next agent to pass it on to in the message's route. The thread then checks through all agents connected to it for the name of the one next in the route. When it finds it, it then adds it into the queue of that agent whose responsibility it now is. If in the event it does not have the next agent in the message's route connected to it, the message is simply lost and the thread moves onto the next one in the queue.

The above process describes a non-cloneable agent. A non-cloneable agent only processes one message at a time. The opposite of a non-cloneable agent is a cloneable agent. This is where there is still one thread pulling from the queue but it delegates the task of processing the message to a pool of threads. This can increase the throughput of messages and would be recommended for systems where the throughput is more important than the order the messages are processed. If the order the messages are processes in is essential, the agent should not be cloneable as there is effectively no order once they're taken off the queue as the next message could be theoretically taken off the message queue before processing has started on the first message.

SocketAgent

A SocketAgent is basically an extension of Agent so at least one thread is created for processing the internal message queue. The extra functionality of a SocketAgent is that it acts as container for DenoboConnection objects. DenoboConnection's are either added by connecting it to another SocketAgent through the use of a socket or another SocketAgent connects to our SocketAgent. No dedicated thread is required to make our SocketAgent connect to another but we require one for accepting connection requests from other SocketAgents. This means each SocketAgent uses a maximum of two threads, one for the underlying message queue processing and one for potentially having to accept connection requests - Potentially because not every SocketAgent might need to "advertise" its presence. The reason we require a thread for accepting connections is that socket I/O in java uses blocking procedure calls which block execution of the current thread until the call is complete. In the case of accepting connections, a call to the 'accept' procedure will block until another SocketAgent initiates a connection.

The simplified process this thread follows is:

1. Accept the connection request or wait until there is one.
2. Wrap the socket into a new DenoboConnection instance.
3. Add the DenoboConnection instance into the list of connections.
4. Notify any observers of the new connection.
5. Spawn a thread for waiting and processing any data received from this connection.
6. Go back to step 1.

As you can see from step 5, each connection request is going to cause another thread to spawn as like the 'accept' call, the 'read' call will block until we have received some data or a special indicator of a closed connection. The process each data receive thread follows is:

```
Do while (disconnected flag is not set)
    Read the next packet from this connection
    Process and handle the packet
```

RouterWorker

A RouterWorker is a Runnable task whose responsibility is to find a route to a specified destination agent. It does this by recursing through the local network of agents. Once it finds a route, it saves it and continues until the whole web of connections has been explored because there is a possibility that multiple routes can be found. Once all routes have been found, the one with the least number of agents in the path is considered as the best and is chosen. If a route is not found, the agent is not notified. We chose not to notify the Agent because we need to be able to force the agent to handle cases where mapping across unstable network points in the network can occur such as when using socket connections. Agent handles this by having a timeout at which point it determines that there is no route to the specified destination if a route is not found within a certain time.

This task was chosen to run in a separate thread as it prevents message queue processing coming to a halt every time a route needs mapping as we assume that a large network could take a while to discover a route.

Undertaker

An Undertaker instance is responsible for updating the routing tables of every Agent in a network. This entity 'crawls' the local network in the same way as a RouterWorker instance does. An Undertaker instance is spawned whenever there is a 'negative change' in the network. A negative change is defined as.

- A link between two Agents being broken.
- An Agent that has shutdown.

Either one of these can potentially break a route to another agent so it is important that every agent in the network is notified which gives it a chance of discovering a new route. This is a good candidate for threading for the same reasons as discussed in the section above for RouterWorker.

Concerns

Whilst the use of threads can greatly enhance the performance of a piece of software, it is well known that many non- reproducible bugs can occur due to bad use of threads that we will not repeat is already well understood in software engineering.

Most issues caused by threads can be summed up with state modification - one thread modifying the state of one object whilst another is reading from it or even modifying it at the same time. This can result in a corrupt state for the subject object. To alleviate against issues like these, synchronization is required but this isn't a once-applied-and-fixed solution as the logic of the code has to change. For example:

```
If (hashmap.contains("pieceofdata") {
   data = hashmap.get("pieceofdata");
   ...
}
```

In this scenario: 'hashmap' could be an instance of a HashMap that has been wrapped within a synchronized wrapper which makes it "Thread-safe". The problem is that the code logic is not thread safe as it is entirely possible that another thread could remove the "pieceofdata" value from the hashmap in-between the thread running the code above checking if "pieceofdata" exists and retrieving it. We took into account this when researching threading and there are two solutions to correcting the above: You could wrap the whole block within a 'synchronized' statement with 'hashmap' as the lock or you could simply skip the check and see if you were returned something – We preferred the latter as if the collection was already synchronized, we did not require another synchronized block which can affect the performance and is much cleaner.

CopyOnWriteArrayList

Throughout the development of Denobo, we use the observer pattern to notify observers of particular changes to subject objects. So the subject knows which observers to notify, it contains a list of observer object instances that it iterates and invokes a notification method on. Iterating a list is inherently not thread safe as it is possible for it to change whilst iterating through it. The observers in the Agent and SocketAgent classes of Denobo are notified often for things such as receiving a message, a new connection or a connection lost which meant that we required synchronization around these observer notify blocks as the internal observer list could have been modified during these notification blocks. Whilst this works, it isn't as efficient as it could be during high-throughput scenarios of many messages coming through the system.

One thing we observed about the use of observer lists is that they rarely change and most of the observers were attached just after object construction and detached (or sometimes just garbage collected) when the object was no longer used. It is well known that immutable objects are inherently thread-safe so this is where the CopyOnWriteArrayList comes into use. This collection acts like an ArrayList except that modifications to it create a new copy of the list internally. Any threads that were currently iterating at the change will still be using the old copy reference at that moment. The advantage to this is that no synchronization is necessary which makes the code more maintainable and cleaner. We also noticed an increase in message throughput during stress testing as threads were less busy trying to acquire a lock to an object for synchronization. CopyOnWriteArrayList is not free as it entails a new list copy on every modification so are only suited to small lists where there is many more reads than writes – such as a list of observers.

Semaphore

For restricting the number of connections allowed to be connected to a SocketAgent, we make use of a Sempahore. A semaphore is basically a permit manager what initially begins with a set amount of permits equal to the maximum number of connections allowed. Whenever there is a connection request or a request to create a connection, we ask the semaphore for a permit. This is an atomic operation so is inherently thread safe with no synchronization necessary. If we successfully acquire a permit, we are still within the limit but if it fails to give us one, we know we are at our limit at that point in time. Once a connection has closed, it releases the permit it acquired when it was created. This is a cleaner and more efficient solution compared to synchronizing a conditional check.