COMP 30220 – Distributed Systems

Four Of A Kind Project Report

Authors:

Alan Halvey:	14465722
Alan Holmes:	14719591
Brian Dunne:	13394546
Orla Cullen:	14708689
Gavin Keaveney:	05641349

Table Of Contents

1. Overview of Problem

The purpose of this project was to implement a distributed system in which the components of the system can reside on computers that are interconnected via a network. These computers can then pass messages over the network to transmit some information or recent activity. In the distributed system each part of the system is divided into multiple smaller jobs. These jobs can be addressed by one or more networked computers by implementing a message oriented service.

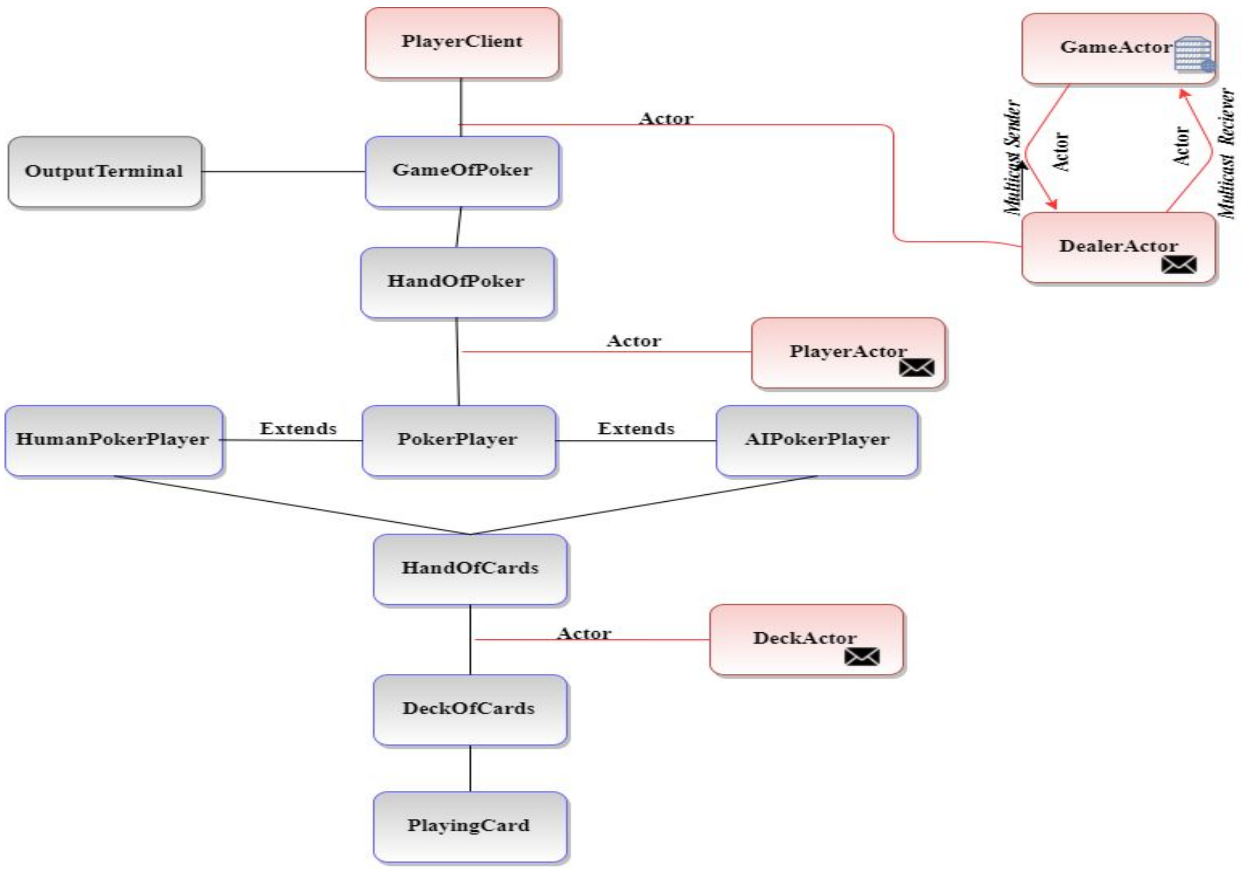
There are a number of key considerations which we must be aware of during the development of this distributed system. In the system, the components can process information concurrently. This means that messaging must be handled. In games for instance, if the concurrency is not managed, messages can be sent to the wrong player or be intercepted ahead of another message that should have been processed first. This could end up leading to inconsistent gameplay[1]. Subsequently, individual components need to work cooperatively and communicate information about the gameplay relying on a concept of shared time. However, the synchronization of clocks on and between machines is not always exactly precise. Additionally, the components in a distributed system should be able to fail individually. This failure should not affect the other components which should all remain in a running state.

The Poker distributed system will consist of a client side implementation and a server side implementation. It will endeavour to incorporate the key ideas of the Akka distributed system as the core technology. This will help the components act on information received and communicate changes in the game state by having the actors transmit the messages between the components.

2. System Architecture

The system architecture consists of the core game play classes which make up the client side and the server side classes which are the classes shaded in red which provides the distribution of the poker system over a network (see *figure 1)*. The core client side classes consist of the GameOfPoker.java, PokerPlayer.java, HandOfPoker.java, OutputTerminal.java, HumanPokerPlayer.java, AIPlayer.java, HandOfCards.java, DeckOfCards.java and PlayingCard.java. These are depicted by the grey components. However, for the purpose of this system architecture we will place more focus on the server side classes consisting of the GameActor.java MulticastSender.java MulticastReciever.java DealerActor.java PokerPlayer.java DeckActor.java and PlayerClient.java. All the actors within the system have the functionality to simply send a message asynchronously and return immediately using the tell method but also they can use the ask method [2] which instead of return immediately it returns a future which represents a possible reply. In order to reply to a message we must call getsender() this returns the actorref when you reply to the actorref and use tell method it sends a message back to the relevant sender. Using the tell ,method is preferably to ask due to the need to find a way to monitor timeouts. In addition message ordering can be assured on a per sender basis when we employ the use of the tell and ask builtin methods[2].

figure 1: System Architecture



The diagram *(see figure 1)* shows how the components interact or are connected to each other. In order for the system to start first the Game Actor must be run this listens for connections when it receives a connection request the multicast receiver it get the ip address and sends it via the multicast sender to the dealer actor which responds to the request by creating a game of poker and starting a round. The game play then continues with the deck actor taking care of the deck of cards resource and the poker player actor ensures the continuity of the message transfer between players ensuring that the correct player at the corresponding ip address obtains the relevant message from the game play. Consequently due to the ip address being held in the program for the duration of the game we can have many games running concurrently as long as we ensure that the messages are being communicated to the correct player.

3. Technology Choices

Technology	**Motivation for Use**
AKKA	Akka is based on the actor model and supports a range of features that allow for the ease of programming concurrent, parallel and distributed systems. It provides APIs for Java.

4. The Team

These are our team members and their assigned tasks.

Student Number	Name	Assigned Task(s)
14465722	Alan Halvey	GameOfPoker, GameActor, MulitcastReciever, PokerPlayer
14719591	Alan Holmes	HandOfCards, DealerActor, PlayerActor
13394546	Brian Dunne	DeckOfCards, DeckActor, HandOfPoker
05641349	Gavin Keaveney	OutputTerminal, MulticastSender, HumanPlayer
14708689	Orla Cullen	PlayingCard, PlayerClient, AIPlayer

5. Task List

There was just one main task within the project as we decided as a group to focus our attention on implementing akka into the poker game. This provides us with a way of distributing the poker game over a network using the actor model as a way of communicating between the components. The main task will be divided into subtasks or classes and divided among the team member as specified in chapter 4.

5.1 Task 1: Implement The Actor Model into the Poker Game

The following seven classes consist of server side of the poker game they are the classes which are paramount to the program design when the actor model is implemented using akka. These are the classes that provide the distribution to the program which allows the components to reside on different nodes and interact with each other by passing messages by routing through the network. It is important to note that each of the actors in this program is operating on a different port and where there is more than 1 game in progress the game actor is incremented which means with concurrent games the game Actor will be using a port unique to that particular game for the actor.

GameActor.java

The Game Actor is the server this must be started before anything else when the configuration is created it is given your local ip address; this is the host. Once this is started it waits for a connection by sending the ip address to listen for requests.

MulticastSender.java

The multicast sender is the component of the system that keeps listening on the network for something to make a connection it does this by sending out its ip address and if someone responds to it's ip address then the multicast

MulticastReciever.java

Once the multicast sender has received a response the multicast receiver reads the bytes of the message received in form of a datagram and tries to accept the response to the socket . The ip address of the message received will become the player information that will be carried into the game

NB: The important thing to note is that in both the sender and the receiver we are using the multicast this identifier allows for the identification of a set of interfaces, not just one like in unicasts and also the multicast is that stretches of multiple nodes for instance in our case we would like the system to be able to function in a distributed manner . Subsequently when the address receiver is in an active state, with the result that any packet received to a multicast address is transmitted to all the components that hold the identified address this means the game is updated in all components as the changes occur. The sender and the receiver form a kind of loop that repeatedly checks for connections and the work simultaneously.

PlayerActor.java

The player actor stops and starts the poker player allowing the player to take turns as there will be multiple players in a game the pre start and poststop methods allow the change over of the player by referencing the actor name . Additionally, when a message object is received it checks if the message needs a reply. If it does it asks the user to input the message to be transmitted and then use the getsender().tell (msg, null) by using this it allows the player to send their message as a response to whatever the question was asked in the game by using the scanner utility to take input from the user. If a responsponse is not needed the dealer just executes the message.

DealerActor.java

The dealer actor has similar pre start and post stop methods however the dealer is always active in the game so unlike the player actor it is not changed over mid game as there will only be one dealer per game. This means that the dealer will only be started and stopped on entry and exit from the game. the name is not changed so the dealer remains the same throughout the game but is stopped and started on entry and exit from the game. When it receives a message it gets the sender's name which is the player and prints the message to the terminal and sets the boolean got player to true this allows the game to progress into the round.

DeckActor.java

The deck actor can receive a message from the game. when the message is the same as its comparison the deck actor will then take action on the Deck of cards class by invoking the method that is required for instance if the message is equal to deal next then the actor will tell the dck to deal the next card and inform the sender of the card by using the tell method.

PlayerClient.java.

The player client is the second component that is run after the game Actor this starts the client side or the game play side by sending a response to the multicast sender when it intercepts the signal allowing the dealer to know that the game now has a human player. when this is true the client will then be prompted to enter their name.

The following will give a brief summary of each of the client's side classes in the system architecture and with reference to there basic functionality. The actor system and the actor ref are passed as arguments into the constructor of some of the within the next classes . The actor classes described previously interact with these classes sending appropriate messages to the gameplay to the corresponding component. How ever most of the implementation of akka occurs in the aforementioned classes.

GameOfPoker.java

The game of Poker class takes the player and dealer reference from the dealer and puts the ip received for the host and the ip received from the player and creates a game system of the human poker player class with the dealer the player the output terminal and the deck . It allows a game to run consisting of many rounds and also keeps track of the current thread this allows multiple games to be run because each game is run is a separate thread.

PokerPlayer.java

The poker player class is an abstract class that provides abstract methods which will be implemented in the Human poker player and the AI poker player class.

HandOfPoker.java

The Hand of Poker is the class that runs the implementation of the round of poker . It uses functions from other classes to create the round of poker keeping track of the pot checking for winners adding players to the round and asking if you want to proceed to a new round when a round has been finished and won.

OutputTerminal.java

The output terminal simply takes in a dealer reference and a player reference and outputs the relevant message to a player pertinent to the message.

AIPlayer.java

This class extends the Poker player and provides the implementation for all the classes related to the AI Player . The AI player allows the game to run even when there are no other players on the network requesting to join the game. this means means that the game will be able to run even when there is only 1 human player by filling the other 3 places at the table with 3 AI Players. In the AI player the betting logic and the players responses are automated . It also incorporates a level of risk dependant on the hand of cards that is held by the AI player it will bet with respect to the hand of cards.

HumanPokerPlayer.java

This class extends the Poker player and provides the implementation for all the classes related to the Human Player. In the human player the player will be asked for a response to a question. It provides the implementation to discard the cards at a position in the player's hand.Like the AIPlayer the human player must also implement all the abstract methods from the poker player class because it is a subclass of the poker player. Also like the AI player it provides some core functionality for getting the betted amount from the player.

HandOfCards.java

This class takes the deck actor reference. This class evaluates the type of hand that the player has by scoring the hand. In poker this game of poker there are 10 types of hand a player can have ranging for high hand which has the lowest value to a royal flush having the highest possible value. This class also provides functionality for discarding cards by allowing up to three cards to be discarded and replaced by new ones.

DeckOfCards.java

This class builds the deck of 52 cards of which will contain 13 cards in each suit . In this class it is important to synchronize some of the methods the reason for this is too ensure game integrity. In terms of functionality the deck must also have methods to shuffle the deck return a card to the deck and deal the next card along with a reset when we run out of cards.

Playing Card.java

This class is to create the playing cards individually which will encompass the four suits clubs heart diamonds and clubs. This is a base class which will be used to build a deck.

6. Reflections

This chapter will provide the reader with insights into the insights gained by the team in relation to the technology used. We will reflect upon the software development process and provide the team's perspective of the issues we faced during the software development the benefits and limitations of using the technologies in our system and what we have learned from the project as a team.

6.1 *Were the technologies appropriate?*

The technology used we feel was appropriate as Akka provided a framework for implementing the distributed system using a high level abstraction rather than delving into the complex intricacies of the actor model. It allowed us to develop our poker system with ease by utilizing the architectures and functionality already provided in the API. the Akka framework is purposefully built to ensure that concurrent distributed systems are easier to implement and due to the tell and ask operations the messaging service we needed was robust enough to handle our messaging on a per sender basis.

6.2 *Did distribution cause unexpected problems?*

Overall the team didn't experience any problems with akka that affected our system adversely. The akka API took some time to understand and implement however once we had a clearer view of how the akka framework worked and how it could benefit our poker system. It was relatively easy to apply the akka technology practically within our project.

6.3 *Limitations of the technology*

Akka is built off a let it crash philosophy so in the event that at message is lost in a crash akka does not guarantee the integrity and delivery of messages. This is one of the fundamental building blocks of akka that operates at the core of its technology with respect to failure[3], [4]. The Akka framework intentionally trivializes the message delivery as to not work from an standpoint of being oversimplified would lead to significant complex and low level coding of message systems which are equally not guaranteed to fulfil the message delivery process. In relation to the poker game we have implemented this would be a limitation in our system as many users may be playing concurent games of poker but for the game to continue running some of the messages must be delivered so one of the games may fail but the other games will continue to run.

6.4 *What are the benefits of the technology you used?*

The benefits of using akka as our main technology was that it allowed the team to build a system of actors and messages allowing us to reduce the duplicated code and uses pre-made functionality to process jobs multiple times without the need of a lot of programming[1]. Akka is also extremely good at handling asynchronous messaging and has an easy to use implementation making it programatically easier to set up. It tends to be stable and employs a let it crash philosophy and supervisor solution to the failure of components and it is scalable allowing for the addition of more modules and components easily. Due to messaging being delivered asynchronously the systems are naturally concurrent as the decoupling of the senders and receivers of the message allow any actors with input messages to be carried out in parallel[2], [5].

Actors in the akka toolkit send and receive messages in the same way, little importance is placed on whether the host is local or it is distributed among different host networks . This is done by utilizing routing services or through direct interaction which can be in a running state on many threads. This makes the program flexible in terms of scalability which will allow us to scale the program both up and out to make use of more powerful servers or by deploying the program across more servers. The key to this is that this can be achieved without much alteration.

6.5 *What did you learn*

As a group we have learned alot about akka and how it use s the actor model to distribute a program across a network by providing a high level abstraction. It facilitates the programmer by addressing thread management ensuring that the distributed system is coded correctly allowing the programmer to overlook the locking [2], [6]. Akka incorporates supervision hierarchies every actor is fully interactive and the built in supervisor servers to ensure adequate management and monitoring of each actor. Akka can ensure a unified programming model which employs similar semantics across clusters of nodes whether local or distributed which is highly useful when a project needs to be scalable. It provides a framework for managing failure of a node. This is achieved by informing the supervisor of the failure of the node or by restarting the node. Owing to the abstraction of message passing it permits us to concentrate on the high level technicalities instead of getting enmeshed in low level and more complex issues due to the existence of premade protocols and communication patterns [2], [5], [6]. Subsequently , Akka can provide a very useful and comprehensive toolkit for cloud based applications because of its asynchronous messaging, the manner in which it caters for scalability and the framework it applies making it both fast and simple to deploy.

References

[1] JetBrainsTV, *Introduction to Akka Actors with Java 8*. Youtube, 2016.

[2] “Akka Documentation.” [Online]. Available: https://doc.akka.io/docs/akka/2.5.5/java/index.html. [Accessed: 24-Dec-2017].

[3] “Acting asynchronously with Akka.” [Online]. Available: https://www.ibm.com/developerworks/library/j-jvmc5/index.html. [Accessed: 18-Dec-2017].

[4] M. Gupta, *Akka Essentials*. Packt Publishing Ltd, 2012.

[5] M. Thurau, “Akka framework,” *University of Lübeck*, 2012.

[6] L. Inc, “Akka - Actor-based message-driven runtime - @lightbend,” *Lightbend*. [Online]. Available: https://www.lightbend.com/akka. [Accessed: 12-Dec-2017].