Sockets

Lecture Question

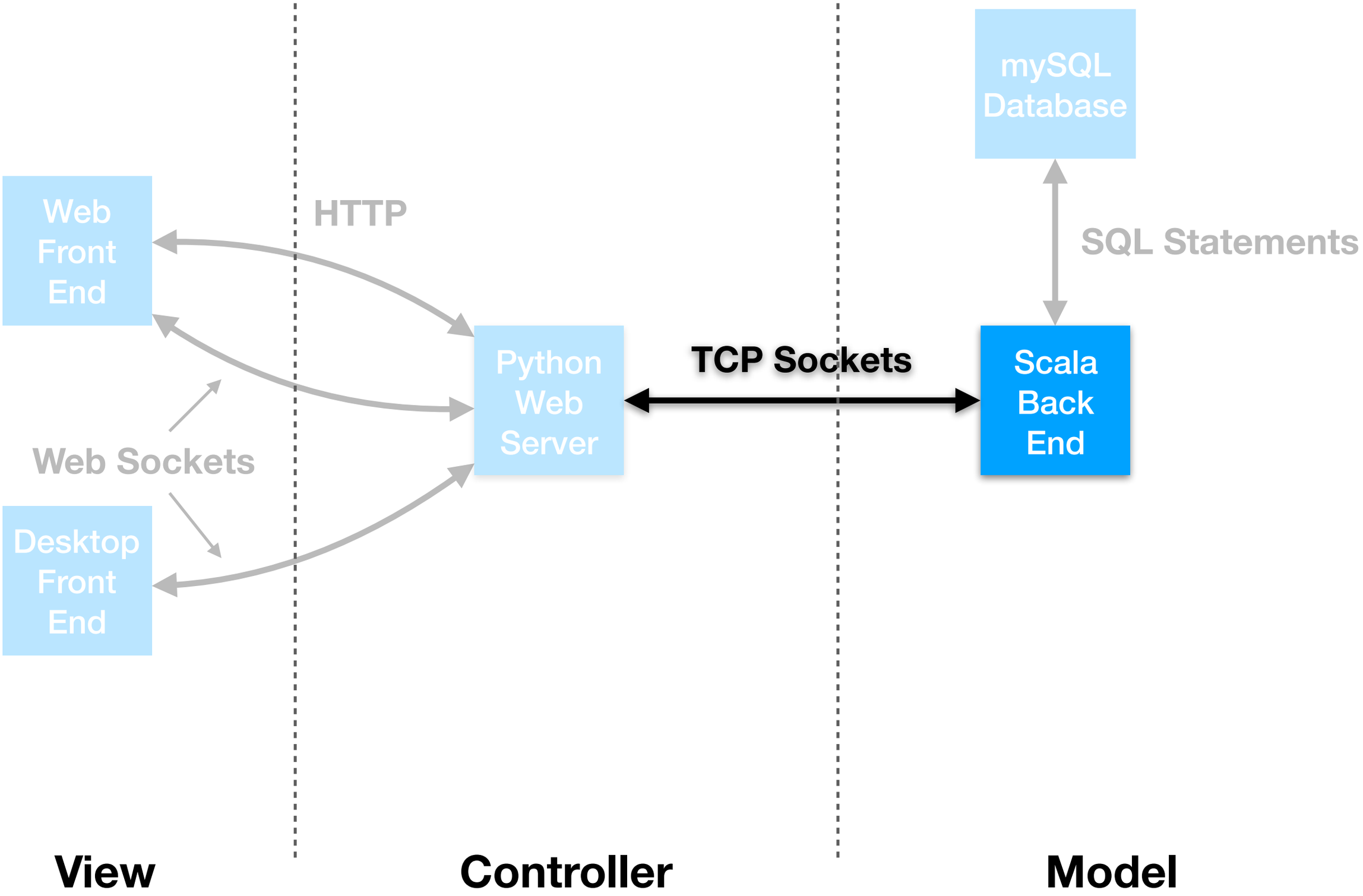
Task: Write a Scala program that functions as a TCP socket server

- Write a Scala program that opens a socket server on localhost port 8000
- The server listens for connections and messages
- The server responds to all messages with "ACK"

There is no testing for this question, though you are strongly encouraged to test your submission to ensure you are comfortable with this concept

* This question will be open until midnight

CSE116 - End Game



Sockets

- Two-way communicate between programs
- Send byte strings
  - Hardware only handles bits/bytes 0/1
  - Whenever a value leaves your program it bust be converted to bits/bytes

Sockets

- Server-Client model
  - Server opens a socket and listens for connections
  - Client connects to a server socket
- Once connected, client and server can send byte strings to each other

Socket Server

Socket Server

- Let's work through the code

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Create a case class that takes a parameter

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Extend Actor and implement receive to react to messages

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Additional import needed for sockets

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Listen for connections on port 8000

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- We'll store references to each connection in a Set

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Set does not allow duplicates and can remove by value

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- A Bound message is received when Bind is ready

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- A Connected message is received when a client connects

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Insert the new client to our set

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Send message to the client to complete the connection

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Use this Actor to handle messages from this client

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Could defer to another ActorRef to send the client elsewhere

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- PeerClosed message received when client disconnects

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Received message received when a client sends a message

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Received has a variable "data" of type ByteString

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Convert from bytes to a utf-8 string and print the message

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- We'll send the server SendToClients messages

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Server

- Server broadcasts the message to all connected clients

```
case class SendToClients(message: String)

class SocketServer extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 8000))

  var clients: Set[ActorRef] = Set()

  override def receive: Receive = {
    case b: Bound => println("Listening on port: " + b.localAddress.getPort)
    case c: Connected =>
      println("Client Connected: " + c.remoteAddress)
      this.clients = this.clients + sender()
      sender() ! Register(self)
    case PeerClosed =>
      println("Client Disconnected: " + sender())
      this.clients = this.clients - sender()
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToClients =>
      println("Sending: " + send.message)
      this.clients.foreach((client: ActorRef) => client ! Write(ByteString(send.message)))
  }

}
```

Socket Client

Socket Client

- New case class to store message to send to the server

```
case class SendToServer(message: String)

class SocketClient(remote: InetSocketAddress) extends Actor {

  import akka.io.Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  var server: ActorRef = _

  override def receive: Receive = {
    case c: Connected =>
      println("Connected to: " + remote)
      this.server = sender()
      this.server ! Register(self)
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToServer =>
      println("Sending: " + send.message)
      if (server != null) {
        this.server ! Write(ByteString(send.message))
      }
  }

}
```

Socket Client

- When created, connect to the provided ip/port

```
case class SendToServer(message: String)

class SocketClient(remote: InetSocketAddress) extends Actor {

  import akka.io.Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  var server: ActorRef = _

  override def receive: Receive = {
    case c: Connected =>
      println("Connected to: " + remote)
      this.server = sender()
      this.server ! Register(self)
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToServer =>
      println("Sending: " + send.message)
      if (server != null) {
        this.server ! Write(ByteString(send.message))
      }
  }

}
```

Socket Client

- Create variable that will store the server ActorRef

```
case class SendToServer(message: String)

class SocketClient(remote: InetSocketAddress) extends Actor {

  import akka.io.Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  var server: ActorRef = _

  override def receive: Receive = {
    case c: Connected =>
      println("Connected to: " + remote)
      this.server = sender()
      this.server ! Register(self)
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToServer =>
      println("Sending: " + send.message)
      if (server != null) {
        this.server ! Write(ByteString(send.message))
      }
  }

}
```

Socket Client

- Connected and Received same as server

```
case class SendToServer(message: String)

class SocketClient(remote: InetSocketAddress) extends Actor {

  import akka.io.Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  var server: ActorRef = _

  override def receive: Receive = {
    case c: Connected =>
      println("Connected to: " + remote)
      this.server = sender()
      this.server ! Register(self)
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToServer =>
      println("Sending: " + send.message)
      if (server != null) {
        this.server ! Write(ByteString(send.message))
      }
  }

}
```

Socket Client

- Before sending message to server, check if we've connected

```
case class SendToServer(message: String)

class SocketClient(remote: InetSocketAddress) extends Actor {

  import akka.io.Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  var server: ActorRef = _

  override def receive: Receive = {
    case c: Connected =>
      println("Connected to: " + remote)
      this.server = sender()
      this.server ! Register(self)
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToServer =>
      println("Sending: " + send.message)
      if (server != null) {
        this.server ! Write(ByteString(send.message))
      }
  }

}
```

Socket Client

- Concurrency Concern: Might receive a SendToServer message before connection is complete

```
case class SendToServer(message: String)

class SocketClient(remote: InetSocketAddress) extends Actor {

  import akka.io.Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  var server: ActorRef = _

  override def receive: Receive = {
    case c: Connected =>
      println("Connected to: " + remote)
      this.server = sender()
      this.server ! Register(self)
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToServer =>
      println("Sending: " + send.message)
      if (server != null) {
        this.server ! Write(ByteString(send.message))
      }
  }

}
```

Socket Client

- Concurrency Concern: Could lose messages if not careful!

```
case class SendToServer(message: String)

class SocketClient(remote: InetSocketAddress) extends Actor {

  import akka.io.Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  var server: ActorRef = _

  override def receive: Receive = {
    case c: Connected =>
      println("Connected to: " + remote)
      this.server = sender()
      this.server ! Register(self)
    case r: Received =>
      println("Received: " + r.data.utf8String)
    case send: SendToServer =>
      println("Sending: " + send.message)
      if (server != null) {
        this.server ! Write(ByteString(send.message))
      }
  }

}
```

Lecture Question

Task: Write a Scala program that functions as a TCP socket server

- Write a Scala program that opens a socket server on localhost port 8000
- The server listens for connections and messages
- The server responds to all messages with "ACK"

There is no testing for this question, though you are strongly encouraged to test your submission to ensure you are comfortable with this concept

* This question will be open until midnight