CSS533: Program 3

Shreevatsa Ganapathy Hegde

University of Washington, Bothell

sghegde@uw.edu

Prof. Munehiro Fukuda

15th May, 2025

Table of Contents

Documentation:

This homework implements a mobile agent platform in Java, where agents are capable of autonomously migrating between distributed execution environments known as Places. The agents carry their execution state and logic with them, allowing them to resume computation seamlessly on remote nodes.

To implement the above functionality, the following methods were modified. Each method is explained in detail below.

Agent.hop(): This method is a core function in the mobile agent program that enables an agent to migrate from its current Place (execution environment) to a remote Place on another host. It takes three arguments: the destination hostname, the name of the method to execute upon arrival, and an optional array of string arguments. Internally, hop() serializes the agent's current state and bytecode, sends it over the network using Java RMI to the specified destination Place, and requests the Place to deserialize and resume the agent by invoking the specified method. This method not only facilitates physical mobility of agents across distributed nodes but also ensures logical continuity by enabling method-level control flow at each destination. After initiating the hop, the current thread is terminated, transferring full control to the destination system.

Agent.run(): This method is the main execution entry point for a mobile agent once it has been deserialized at its destination Place. It is invoked by the Place's thread after receiving and reconstructing the agent via RMI. The method uses Java reflection to dynamically invoke the agent's specified behavior method (such as init, step, jump, etc.), which was indicated during the call to hop(). This method name, along with any arguments, is passed to run() and executed within the agent's new environment. agent.run() enables the agent to resume its lifecycle at the appropriate method point, allowing stateful execution across multiple hosts. After the target method completes, run() terminates, completing that phase of the agent's execution before it initiates another migration.

Place.transfer(): The Place.transfer() method is responsible for receiving and launching a mobile agent that has arrived from another node. When an agent invokes hop(), it serializes itself and sends its bytecode, state, target method name, and arguments to the destination Place using Java RMI. Upon arrival, transfer() is called on the receiving Place, where it deserializes the agent's state, dynamically loads its class using a custom class loader, and restores the agent instance. It then assigns necessary runtime properties such as the port number and starts the agent in a new thread by calling its run() method. This mechanism allows the Place to seamlessly accept and activate agents arriving from remote systems, preserving both their logic and internal state for continued execution.

MyAgent Class: This class extends the base Agent class and defines a structured migration path across multiple remote nodes. The agent maintains a hopCount to track its current position in the journey and a destination array that specifies the sequence of hosts it should visit. Its lifecycle begins with the init() method, which triggers a hop to the next node and invokes the step() method upon arrival. This process continues through the jump() and fall() methods, with each method printing the agent's ID, hop count, destination, and a message such as "Hello!", "Oi!", or "Bye!". The class includes safety checks to prevent out-of-bounds errors when the number of destinations is less than expected.

These are the modifications needed as part of base functionality. Additional Features will be discussed in discussion section.

Source Code:

Agent.java:

```
package Mobile;
import java.io.*;
import java.rmi.*;
import java.lang.reflect.*;

/**
 * Mobile.Agent is the base class of all user-define mobile agents. It carries
 * an agent identifier, the next host IP and port, the name of the function to
 * invoke at the next host, arguments passed to this function, its class name,
 * and its byte code. It runs as an independent thread that invokes a given
 * function upon migrating the next host.
 *
 * @author  Munehiro Fukuda
 * @version %I% %G%
 * @since   1.0
 */
public class Agent implements Serializable, Runnable {
    // live data to carry with the agent upon a migration
    protected int agentId        = -1;    // this agent's identifier
    private String _hostname     = null;  // the next host name to migrate
    private String _function     = null;  // the function to invoke upon a move
    private int _port            = 0;     // the next host port to migrate
    private String[] _arguments  = null;  // arguments pass to _function
    private String _classname    = null;  // this agent's class name
    private byte[] _bytecode     = null;  // this agent's byte code

    /**
     * setPort( ) sets a port that is used to contact a remote Mobile.Place.
     *
     * @param port a port to be set.
     */
    public void setPort( int port ) {
    this._port = port;
    }

    public int getPort( ) {
        return _port;
    }

    public String getHost( ) {
        return _hostname;
```

```
}

/**
 * setId( ) sets this agent identifier: agentId.
 *
 * @param id an idnetifier to set to this agent.
 */
public void setId( int id ) {
this.agentId = id;
}

/**
 * getId( ) returns this agent identifier: agentId.
 *
 * @param this agent's identifier.
 */
public int getId( ) {
return agentId;
}

/**
 * getByteCode( ) reads a byte code from the file whosename is given in
 * "classname.class".
 *
 * @param classname the name of a class to read from local disk.
 * @return a byte code of a given class.
 */
public static byte[] getByteCode( String classname ) {
// create the file name
String filename = classname + ".class";

// allocate the buffer to read this agent's bytecode in
File file = new File( filename );
byte[] bytecode = new byte[( int )file.length( )];

// read this agent's bytecode from the file.
try {
    BufferedInputStream bis =
    new BufferedInputStream( new FileInputStream( filename ) );
    bis.read( bytecode, 0, bytecode.length );
    bis.close( );
} catch ( Exception e ) {
    e.printStackTrace( );
    return null;
}
```

```
// now you got a byte code from the file. just return it.
return bytecode;
}

/**
 * getByteCode( ) reads this agent's byte code from the corresponding file.
 *
 * @return a byte code of this agent.
 */
public byte[] getByteCode( ) {
if ( _bytecode != null ) // bytecode has been already read from a file
    return _bytecode;

// obtain this agent's class name and file name
_classname = this.getClass( ).getName( );
_bytecode = getByteCode( _classname );

return _bytecode;
}

/**
 * run( ) is the body of Mobile.Agent that is executed upon an injection
 * or a migration as an independent thread. run( ) identifies the method
 * with a given function name and arguments and invokes it. The invoked
 * method may include hop( ) that transfers this agent to a remote host or
 * simply returns back to run( ) that termiantes the agent.
 */
public void run( ) {
    try {
        // get class and method to invoke
        Class<?> cls = this.getClass();
        if (_arguments == null) {
            Method method = cls.getMethod(_function);
            method.invoke(this); // invoke method with no arguments
        } else {
            Method method = cls.getMethod(_function, String[].class);
            method.invoke(this, (Object) _arguments);  // invoke method with arguments
        }
    } catch (InvocationTargetException e) {
        if (!(e.getCause() instanceof ThreadDeath)) {
            e.getCause().printStackTrace();
        }
    } catch (Exception e) {
        e.printStackTrace();
```

```
    }

}

/**
 * hop( ) transfers this agent to a given host, and invoeks a given
 * function of this agent.
 *
 * @param hostname the IP name of the next host machine to migrate
 * @param function the name of a function to invoke upon a migration
 */
public void hop( String hostname, String function ) {
hop( hostname, function, null );
}

/**
 * hop( ) transfers this agent to a given host, and invoeks a given
 * function of this agent as passing given arguments to it.
 *
 * @param hostname the IP name of the next host machine to migrate
 * @param function the name of a function to invoke upon a migration
 * @param args     the arguments passed to a function called upon a
 *                 migration.
 */
@SuppressWarnings( "deprecation" )
public void hop( String hostname, String function, String[] args ) {
    this._hostname = hostname;
    this._function = function;
    this._arguments = args;

    // get this agent's class name and byte code
    _classname = this.getClass().getName();
    _bytecode = getByteCode();
    // serialize this agent to a byte array
    byte[] entity = serialize();

    try {
        // create a place object to transfer this agent
        String url = "rmi://" + _hostname + ":" + _port + "/place";
        PlaceInterface remote = (PlaceInterface) Naming.lookup(url);
        // transfer this agent
        boolean result = remote.transfer(_classname, _bytecode, entity);
        if (!result) {
            System.err.println("Transfer failed to " + url);
        }
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }

        Thread.currentThread().stop();

    }

    /**
     * serialize( ) serializes this agent into a byte array.
     *
     * @return a byte array to contain this serialized agent.
     */
    private byte[] serialize( ) {
    try {
        // instantiate an object output stream.
        ByteArrayOutputStream out = new ByteArrayOutputStream( );
        ObjectOutputStream os = new ObjectOutputStream( out );

        // write myself to this object output stream
        os.writeObject( this );

        return out.toByteArray( ); // conver the stream to a byte array
    } catch ( IOException e ) {
        e.printStackTrace( );
        return null;
    }
    }
}
```

Place.java: Contains the complete version with additional features. For all the required usecases of additional features, new methods are implemented preserving all the base functionality features as it is.

```
package Mobile;

import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Queue;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.rmi.registry.*;
```

```java
/**
 * Mobile.Place is the our mobile-agent execution platform that accepts an
 * agent transferred by Mobile.Agent.hop( ), deserializes it, and resumes it
 * as an independent thread.
 *
 * @author  Munehiro Fukuda
 * @version %I% %G$
 * @since   1.0
 */
public class Place extends UnicastRemoteObject implements PlaceInterface {
    private AgentLoader loader = null;  // a loader to define a new agent class
    private int agentSequencer = 0;     // a sequencer to give a unique agentId
    // Indirect communication
    private ConcurrentHashMap<Integer, Queue<String>> messageBox = new ConcurrentHashMap<>();
    // Agent directory to keep track of agent IDs
    private ConcurrentHashMap<String, Integer> agentDirectory = new ConcurrentHashMap<>();

    /**
     * This constructor instantiates a Mobiel.AgentLoader object that
     * is used to define a new agen class coming from remotely.
     */
    public Place( ) throws RemoteException {
    super( );
    loader = new AgentLoader( );
    }

    /**
     * deserialize( ) deserializes a given byte array into a new agent.
     *
     * @param buf a byte array to be deserialized into a new Agent object.
     * @return a deserialized Agent object
     */
    private Agent deserialize( byte[] buf )
    throws IOException, ClassNotFoundException {
    // converts buf into an input stream
        ByteArrayInputStream in = new ByteArrayInputStream( buf );

    // AgentInputStream identify a new agent class and deserialize
    // a ByteArrayInputStream into a new object
        AgentInputStream input = new AgentInputStream( in, loader );
        return ( Agent )input.readObject();
    }

    /**
```

```java
     * transfer( ) accepts an incoming agent and launches it as an independent
     * thread.
     *
     * @param classname The class name of an agent to be transferred.
     * @param bytecode  The byte code of  an agent to be transferred.
     * @param entity    The serialized object of an agent to be transferred.
     * @return true if an agent was accepted in success, otherwise false.
     */
    public boolean transfer( String classname, byte[] bytecode, byte[] entity )
    throws RemoteException {
        try {
            //  Load agent class
            loader.loadClass(classname, bytecode);

            // Deserialize agent
            Agent agent = deserialize(entity);

            //  Assign unique ID if not yet set
            if (agent.getId() == -1) {
                int id = InetAddress.getLocalHost().hashCode() + agentSequencer++;
                agent.setId(id);
            }

            //  Start agent thread
            Thread thread = new Thread(agent);
            thread.start();

            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }

    }

    /**
     * main( ) starts an RMI registry in local, instantiates a Mobile.Place
     * agent execution platform, and registers it into the registry.
     *
     * @param args receives a port, (i.e., 5001-65535).
     */
    public static void main( String args[] ) {
        if (args.length != 1) {
        System.err.println("Usage: java Mobile.Place <port>");
        System.exit(1);
```

```
        }

        int port = 0;
        try {
            port = Integer.parseInt(args[0]);
            if (port < 5001 || port > 65535) {
                throw new IllegalArgumentException("Port must be between 5001 and 65535");
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            startRegistry(port);
            Place place = new Place();
            Naming.rebind("rmi://localhost:" + port + "/place", place);
            System.out.println("Place is running at port " + port);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    /**
     * startRegistry( ) starts an RMI registry process in local to this Place.
     *
     * @param port the port to which this RMI should listen.
     */
    private static void startRegistry( int port ) throws RemoteException {
        try {
            Registry registry =
                LocateRegistry.getRegistry( port );
            registry.list( );
        }
        catch ( RemoteException e ) {
            Registry registry =
                LocateRegistry.createRegistry( port );
        }
    }

    /**
     * deliverMessage( ) delivers a message to the specified agent.
     *
     * @param receiverId The ID of the agent to receive the message.
```

```
     * @param message The message to be delivered.
     */
    @Override
    public void deliverMessage(int receiverId, String message) throws RemoteException {
        messageBox.computeIfAbsent(receiverId, k -> new
ConcurrentLinkedQueue<>()).add(message);
        System.out.println("Delivered message to agent " + receiverId + ": " + message);
    }

    /**
     * retrieveMessage( ) retrieves a message for the specified agent.
     *
     * @param receiverId The ID of the agent to retrieve the message for.
     * @return The message for the agent, or null if no message is available.
     */

    @Override
    public String retrieveMessage(int receiverId) throws RemoteException {
        Queue<String> queue = messageBox.get(receiverId);
        if (queue == null || queue.isEmpty()) {
            return null;
        }
        return queue.poll();
    }
    /**

     * registerAgent( ) registers an agent with a name and ID.
     *
     * @param name The name of the agent.
     * @param agentId The ID of the agent.
     */
    @Override
    public void registerAgent(String name, int agentId) {
        agentDirectory.put(name, agentId);
    }

    /**

     * lookupAgentId( ) looks up the ID of an agent by its name.
     * @param name
     * @return The ID of the agent, or -1 if not found.
     */
    @Override

    public int lookupAgentId(String name) {
        return agentDirectory.getOrDefault(name, -1);
```

```
    }
}
```

MyAgent.java: Agent for base functionalities.

```
import Mobile.*;

/**
 * MyAgent is a test mobile agent that is injected to the 1st Mobile.Place
 * platform to print the breath message, migrates to the 2nd platform to
 * say "Hello!", and even moves to the 3rd platform to say "Oi!".
 *
 * @author  Munehiro Fukuda
 * @version %I% %G%
 * @since   1.0
 */
public class MyAgent extends Agent {
    public int hopCount = 0;
    public String[] destination = null;

    /**
     * The consturctor receives a String array as an argument from
     * Mobile.Inject.
     *
     * @param args arguments passed from Mobile.Inject to this constructor
     */
    public MyAgent( String[] args ) {
        destination = args;
    }

    /**
     * init( ) is the default method called upon an agent inject.
     */
    public void init( ) {
        System.out.println( "agent( " + agentId + ") invoked init: " +
                    "hop count = " + hopCount +
                    ", next dest = " + destination[hopCount] );
        String[] args = new String[1];
        args[0] = "Hello!";
        // Hop to the first destination
        hop(destination[hopCount], "step", args);

    }

    /**
```

```
     * step( ) is invoked upon an agent migration to destination[0] after
     * init( ) calls hop( ).
     *
     * @param args arguments passed from init( ).
     */
    public void step( String[] args ) {
        hopCount++;
        // The agent is migrated to destination[1] after this method if
        // destination[1] is not null.
        if (hopCount < destination.length) {
            System.out.println( "agent( " + agentId + ") invoked step: " +
                    "hop count = " + hopCount +
                    ", next dest = " + destination[hopCount] +
                    ", message = " + args[0] );
            args[0] = "Oi!";
            hop(destination[hopCount], "jump", args);
        } else {
            System.out.println( "agent( " + agentId + ") invoked fall: " +
                    "hop count = " + hopCount +
                    ", message = " + args[0] );
        }

    }

    /**
     * jump( ) is invoked upon an agent migration to destination[1] after
     * step( ) calls hop( ).
     *
     * @param args arguments passed from step( ).
     */
    public void jump( String[] args ) {
        hopCount++;
        // The agent is migrated to destination[2] after this method if destination[2]
        // is not null.
        if (hopCount < destination.length) {
            System.out.println( "agent( " + agentId + ") invoked jump: " +
                    "hop count = " + hopCount +
                    ", next dest = " + destination[hopCount] +
                    ", message = " + args[0] );
            args[0] = "Bye!";
            hop(destination[hopCount], "fall", args);
        } else {
            System.out.println( "agent( " + agentId + ") invoked fall: " +
                    "hop count = " + hopCount +
                    ", message = " + args[0] );
```

```
        }
    }

    /**
     * fall( ) is invoked upon an agent migration to destination[2] after
     * jump( ) calls hop( ).
     *
     * @param args arguments passed from jump( ).
     */
    public void fall( String[] args ) {
        hopCount++;
        System.out.println( "agent( " + agentId + ") invoked fall: " +
                    "hop count = " + hopCount +
                    ", message = " + args[0] );
    }
}
```

Additional Feature classes:

Few changes to Place.java has been done as part of additional features. The above pasted Place.java contains all the additional features methods namely, deliverMessage(), retrieveMessage(), registerAgent(), and lookupAgentId(). Additionalities Agents created for each additional feature will be pasted here.

Indirect Inter-Agent Communication via Place:

The central communication node was hardcoded to ensure all agents interact with a single shared Place, guaranteeing consistent message delivery and agent directory visibility. This eliminates synchronization issues that arise when agents register or communicate across separate, isolated Place instances.

TestAgentReceiver.java:

```
import Mobile.*;
import java.rmi.*;

public class TestAgentReceiver extends Agent {
    private String[] destination;

    public TestAgentReceiver(String[] args) {
        destination = args;
    }

    public void init() {
        System.out.println("Receiver agent (" + getId() + ") invoked init");
```

```
        try {
            // central node
            PlaceInterface remote = (PlaceInterface)
Naming.lookup("rmi://cssmpi6:50777/place");
            // Register itself with the local Place under a known name
            remote.registerAgent("receiverAgent", getId());
            System.out.println(getId() +" Registered as 'receiverAgent'");
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Migrate to the destination to receive messages
        hop(destination[0], "receive", new String[0]);

    }

    /**
     * receive( ) is invoked when the agent has completed its task.
     *
     * @param args The arguments passed to the receive method.
     */
    public void receive(String[] args) {
        System.out.println( getId() + " invoked receive");
        try {
            // Register itself with the local Place under a known name
            PlaceInterface remote = (PlaceInterface) Naming.lookup("rmi://localhost:" +
getPort() + "/place");
            for (int i = 0; i < 5; i++) {
                // retrieve message
                String msg = remote.retrieveMessage(getId());
                if (msg != null) {
                    System.out.println("Received: \"" + msg + "\"");
                    return;
                } else {
                    System.out.println("No message yet, retrying...");
                    Thread.sleep(2000); // wait 2 second before retry
                }
            }

            System.out.println("No message received after retries.");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
}
```

TestAgentSender.java:

```
import Mobile.*;
import java.rmi.*;

public class TestAgentSender extends Agent {
    private String[] destination;

    public TestAgentSender(String[] args) {
        destination = args;
    }

    public void init() {
        System.out.println("Sender agent (" + getId() + ") invoked init");

        try {
            // Register itself
            // hardcoding the central node
            PlaceInterface remote = (PlaceInterface)
Naming.lookup("rmi://cssmpi6:50777/place");

            // Register itself with the local Place under a known name
            remote.registerAgent("senderAgent", getId());
            System.out.println("Registered as 'senderAgent'");

            // Try to look up the receiver agent by name
            int receiverId = -1;
            for (int i = 0; i < 5; i++) {
                receiverId = remote.lookupAgentId("receiverAgent");
                if (receiverId != -1) break;

                System.out.println("receiverAgent not found, retrying...");
                Thread.sleep(1000);
            }
            // If receiver agent is found, send a message
            if (receiverId != -1) {
                String msg = "Hello from senderAgent("+ getHost() +": "+ getId() + ")";
                System.out.println("Sending: \"" + msg + "\" to receiverAgent (ID: " +
receiverId + ")");
                remote.deliverMessage(receiverId, msg);
                System.out.println("Message sent successfully.");
            } else {
```

```
                System.out.println("receiverAgent not found after retries. Message not
sent.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        // Hop to next destination to complete lifecycle
        hop(destination[0], "done", new String[0]);

    }
    /**
     * done( ) is invoked when the agent has completed its task.
     *
     * @param args The arguments passed to the done method.
     */
    public void done(String[] args) {
        System.out.println("Sender agent (" + getId() + ") done.");
    }
}
```

Agent Spawning (Child Agent Creation):

ParentAgent.java:

```
import Mobile.*;
import java.rmi.*;
import java.lang.reflect.*;
import java.net.*;

public class ParentAgent extends Agent {
    private String[] destination;

    public ParentAgent(String[] args) {
        destination = args; // args[0] = child destination
    }

    public void init() {
        System.out.println("Parent agent (" + getId() + ") invoked init");

        try {
            // Load the bytecode for ChildAgent
            String className = "ChildAgent";
            byte[] bytecode = Agent.getByteCode("ChildAgent");
            if (bytecode == null) {
                System.err.println("ERROR: Could not load ChildAgent bytecode.");
```

```
                return;
            }
            System.out.println("Bytecode loaded for ChildAgent, size: " + bytecode.length);

            // Create an instance of ChildAgent using reflection
            AgentLoader loader = new AgentLoader();
            Class<?> cls = loader.loadClass(className, bytecode);
            Constructor<?> ctor = cls.getConstructor(String[].class);
            // Pass the destination as an argument to the constructor
            String[] childArgs = new String[] { destination[0] };
            Agent child = (Agent) ctor.newInstance((Object) childArgs);
            child.setPort(getPort());

            System.out.println("Calling child.hop(...) to " + destination[0]);
            child.hop(destination[0], "run", new String[0]);  // 'run' method in child
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

ChildAgent.java: This was used to just demonstrate the child spawning. So, Implementation of child class is irrelevant as long as it's aa proper class.

```
import Mobile.*;
import java.rmi.*;
import java.net.*;

public class ChildAgent extends Agent {
    public ChildAgent(String[] args) {
        // Just for demonstration, args[0] is the destination
    }

    public void run(String[] args) {
        System.out.println(" Child agent (" + getId() + ") created and running.");
    }
}
```

Direct Inter-Agent Communication via RMI:

DirectAgent.java:

```
import Mobile.*;
import java.rmi.*;
import java.net.*;

public class DirectAgent extends Agent implements AgentInterface {
    private String[] destination;

    public DirectAgent(String[] args) {
        destination = args; // args[0] = cssmpiX destination, args[1] = "send" or "receive"
    }

    public void init() {
        System.out.println("DirectAgent (" + getId() + ") invoked init, mode: " +
destination[1]);
        try {
            if ("receive".equals(destination[1])) {
                // Register this agent to the registry
                AgentRegistry registry = (AgentRegistry)
Naming.lookup("rmi://cssmpi5:50999/registry");
                registry.registerAgent(getId(), this);
                System.out.println("DirectAgent registered for receiving, ID: " + getId());
            } else if ("send".equals(destination[1])) {
                // Look up the receiver and send message
                AgentRegistry registry = (AgentRegistry)
Naming.lookup("rmi://cssmpi5:50999/registry");
                int receiverId = Integer.parseInt(destination[2]);
                AgentInterface receiver = registry.getAgent(receiverId);
                // Check if the receiver is registered, if registered, send a message
                if (receiver != null) {
                    receiver.receiveMessage("Direct message from Agent " + getId());
                    System.out.println("Message sent directly to Agent " + receiverId);
                } else {
                    System.out.println("Receiver agent not found.");
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void receiveMessage(String msg) throws RemoteException {
```

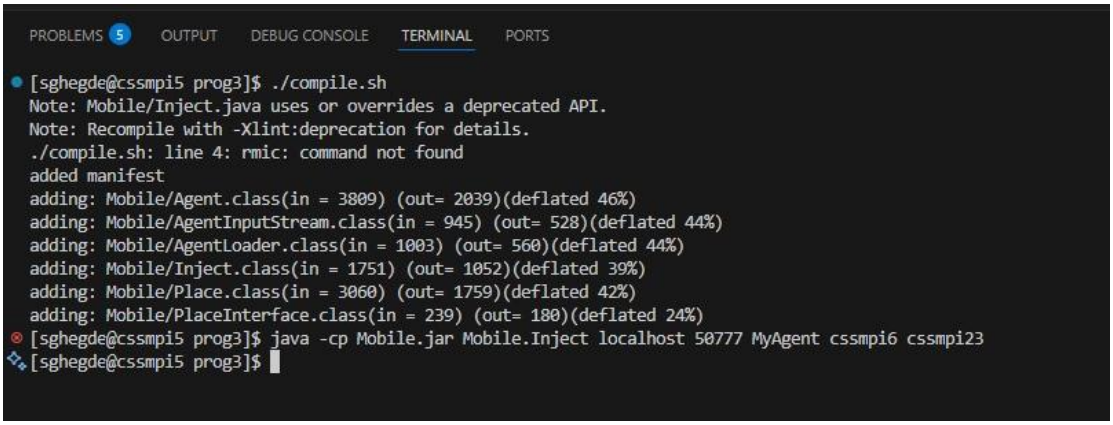
```
        System.out.println("DirectAgent (" + getId() + ") received on " + getHost() + ": \"" +
msg + "\"");
    }
}
```

Execution Outputs:

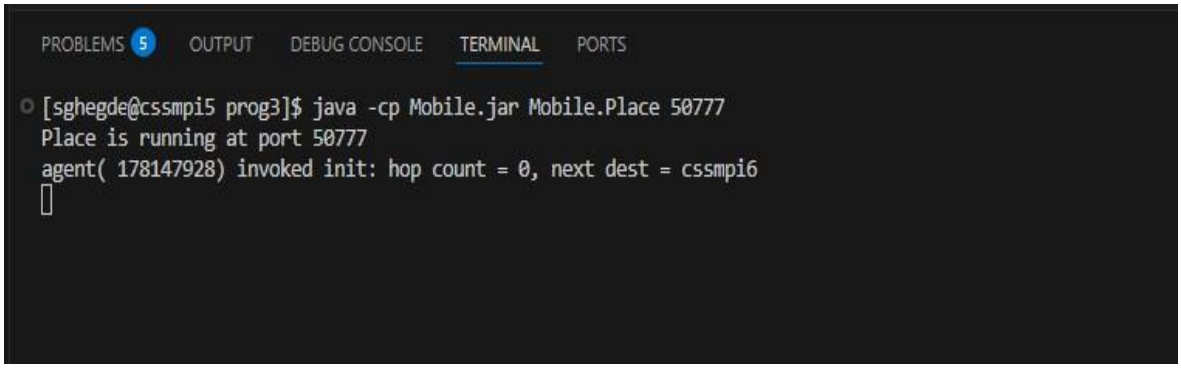
Base Implementation:

Two scenarios were given for the testing of base implementation. The following is the output of both respectively

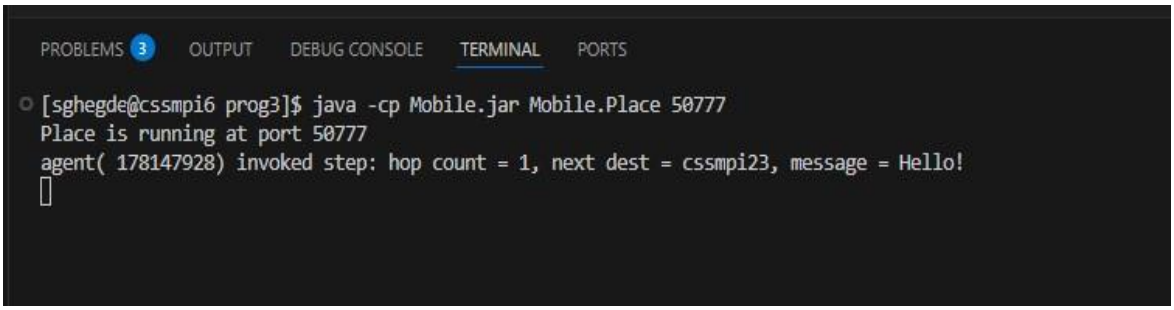
1. Execution with 3 different Places
   Starting point:



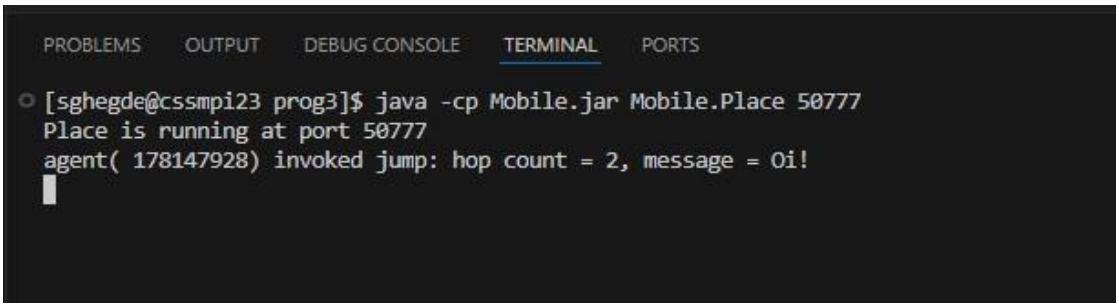
   Execution of Init():



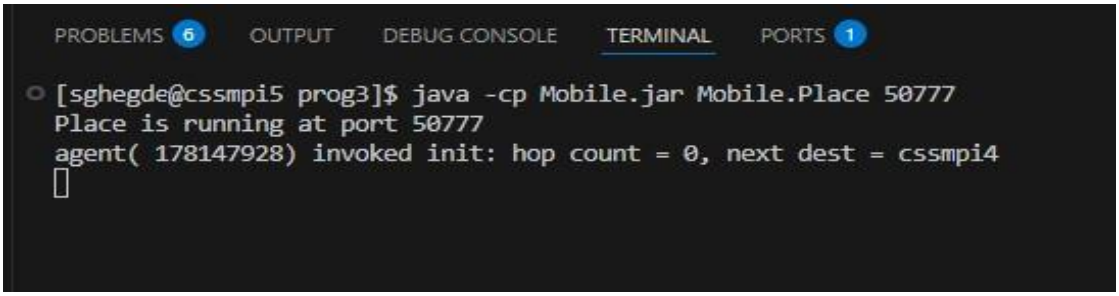
   2nd destination:



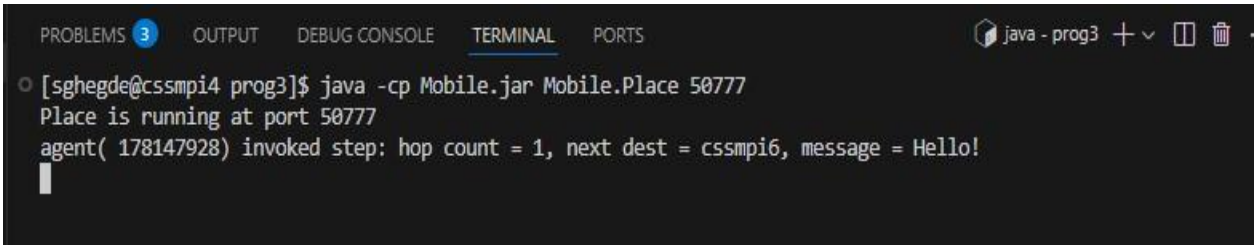
Final Destination:



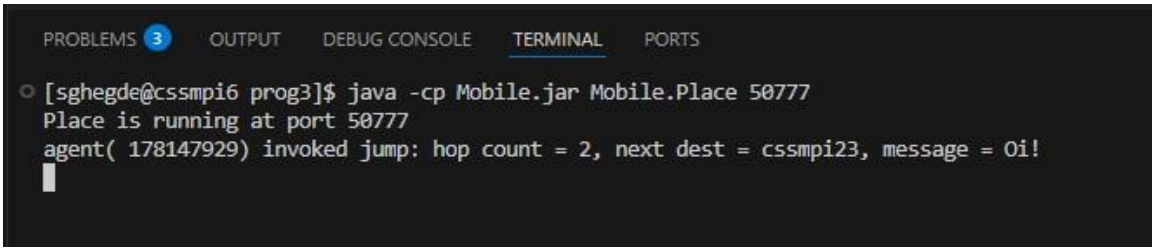
2. Execution with 4 different places:
   Execution of Init:



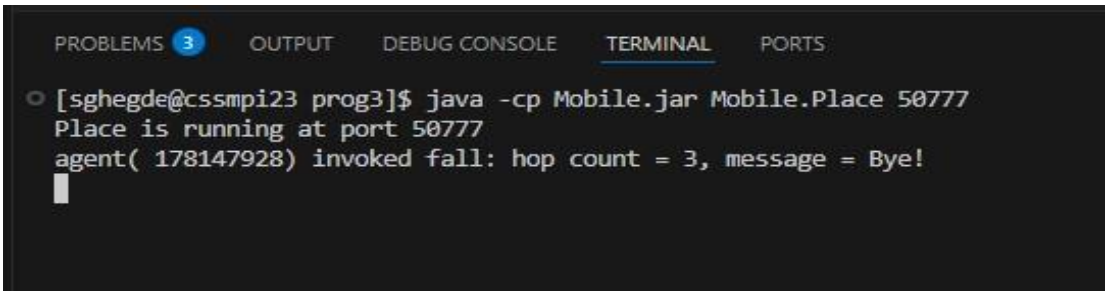
2nd Destination:



3rd Destination:



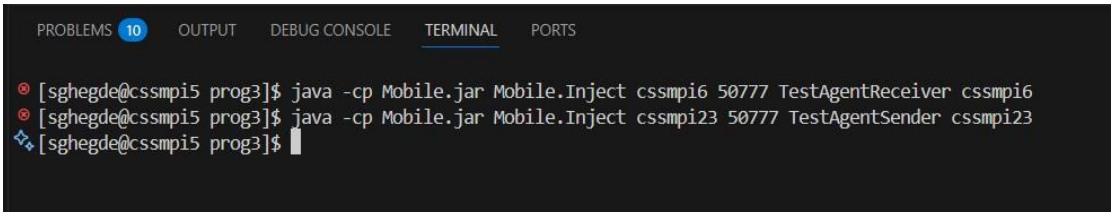
Last destination:



Additional features:

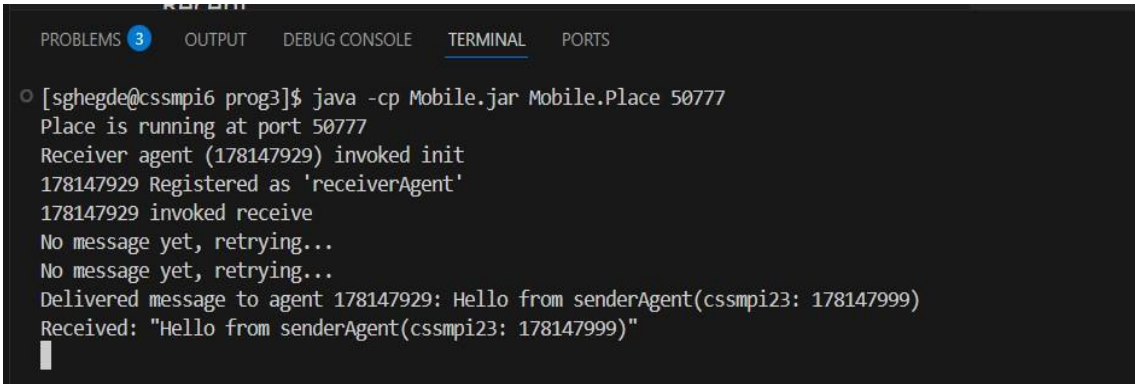
Indirect Communication:

Command Screen:



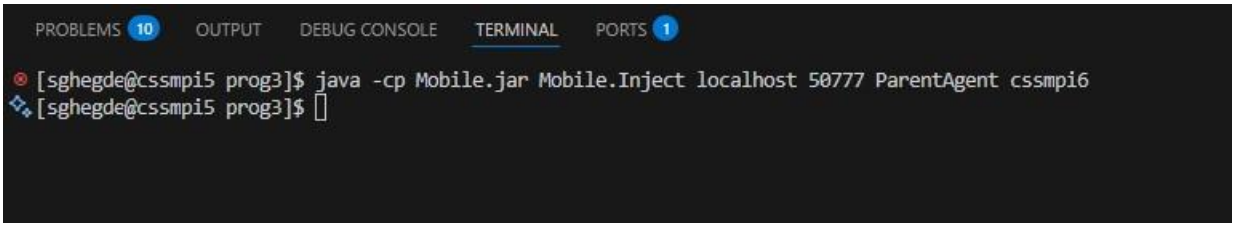
Sender agent:

Receiver Agent:

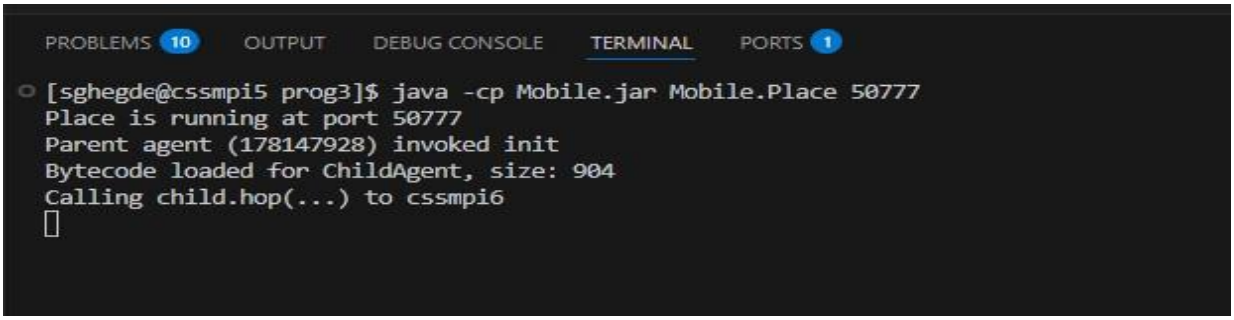


Child Creation:

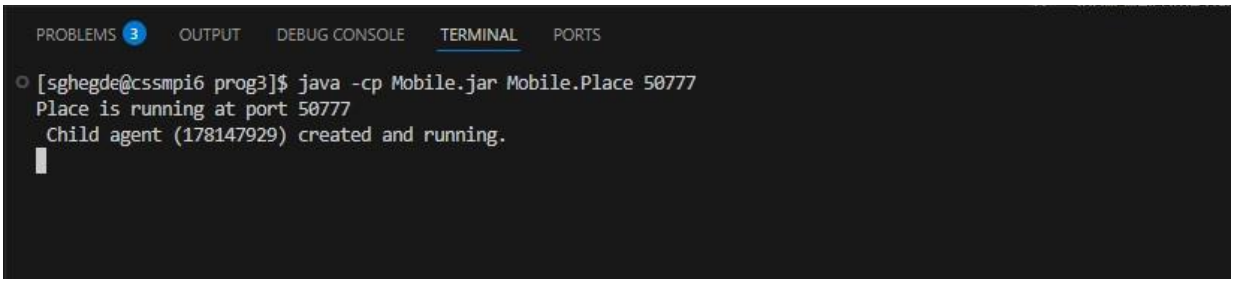
Commands:



Parent:

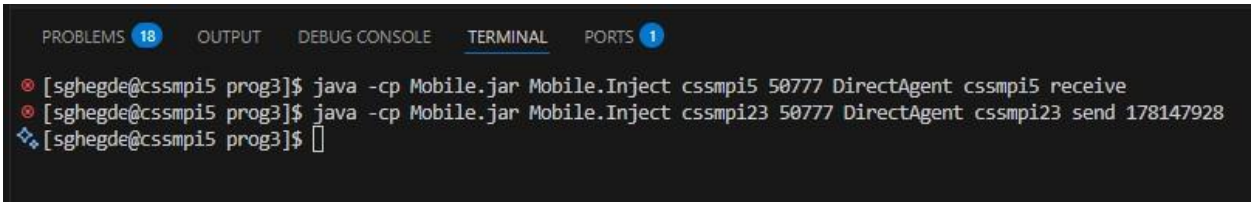


Child:



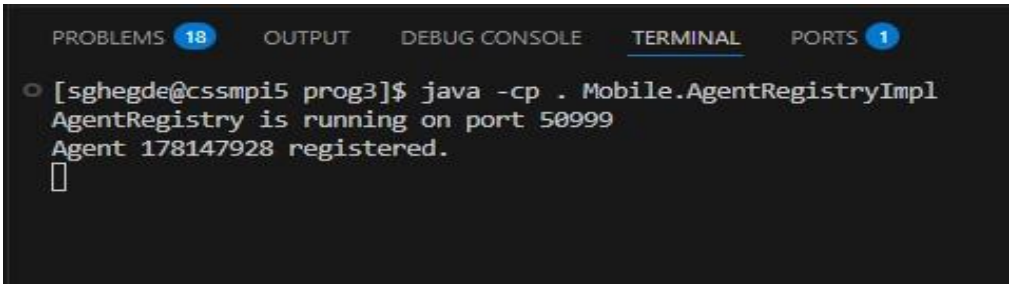
Direct Communication:

Commands:



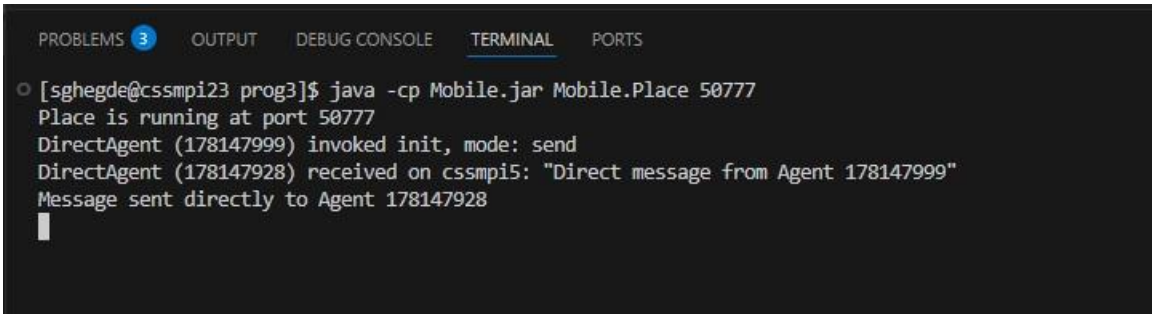
Agent Registry:

I have designed the communication where only receiver has to register. So, sender has not registered.

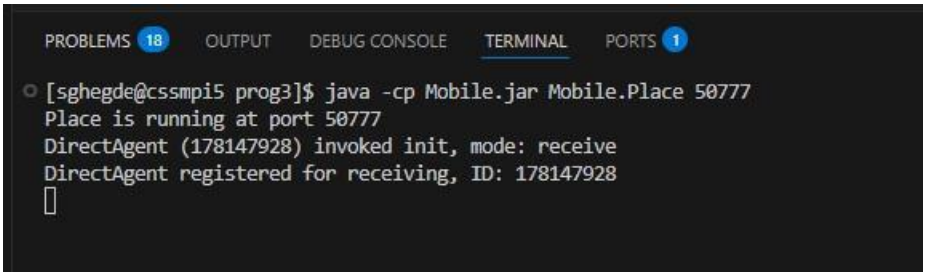


Sender:

The received message is printed in the sender because in my approach, I am not using hop. My goal was to achieve direct communication. So, I am just injecting. In the output, it can be clearly seen that the hostname on which it is received is cssmpi5 rather than the sender's hostname which is cssmpi23.



Receiver:



Discussions:

We will be talking about the additional features in this section. I have implemented 3 additional features. They are listed as below.

Indirect Inter-Agent Communication via Place: The implemented indirect communication feature enables mobile agents to exchange messages asynchronously through a central Place acting as a shared message relay. Each agent can register itself with the Place using a unique name, allowing other agents to look up its ID via name-based resolution. Messages are delivered using the deliverMessage(int receiverId, String message) method, which queues the message in a thread-safe map, and the receiving agent polls its queue using retrieveMessage(int receiverId) to access incoming messages. This design decouples sender and receiver agents in time and space, supporting communication even when the agents are not co-located or active simultaneously. To ensure consistency and avoid synchronization issues across multiple Places, all communication is routed through a hardcoded central Place node. Hardcoding the central Place node was necessary to ensure that all agents interact with a single, consistent message and registration context. This avoids message delivery failures caused by agents registering or polling from different, isolated Place instances running on separate hosts.

Child Creation: The child agent creation, or spawning feature, enables a mobile agent to dynamically generate and dispatch another agent at runtime. This is achieved using Java reflection and bytecode loading. The parent agent retrieves its class bytecode using the getByteCode() method, loads it via a custom AgentLoader, and instantiates a new agent by invoking its constructor with the required arguments. The parent then initiates a hop on the newly created child agent, specifying its target host and the method to be invoked upon arrival. The child is passed a string array as arguments, and upon arrival at the destination Place, executes its method (e.g., run()) as defined.

Direct Communication: The direct communication feature enables one mobile agent to invoke a method on another agent using Java RMI, without involving the Place as a message relay. Each agent that wishes to be directly contacted implements a shared remote interface, AgentInterface, which defines the receiveMessage(String msg) method. Agents that intend to receive messages register themselves with a centralized AgentRegistry, an RMI-based registry service running on a known host and port. This registry stores a mapping between agent IDs and their corresponding remote object references. When a sender agent wants to communicate, it uses RMI to look up the receiver's reference by ID through the registry and directly invokes the receiveMessage() method. In this implementation, both sender and receiver agents remain on their original injected Places without migration. As a result, the message prints on the terminal of the Place where the receiver was injected, reflecting that the method call was indeed executed remotely.

Limitations and Improvements: One limitation of the current system is that direct communication relies on a centralized AgentRegistry, creating a single point of failure and limiting scalability in large distributed environments. Additionally, agents do not currently support name-based addressing in direct communication, requiring manual tracking of numerical IDs. Message delivery in indirect communication also depends on agents actively polling, which may delay reception or cause message loss if agents terminate early. Possible improvements include implementing a decentralized registry or discovery mechanism, adding name-to-ID mapping for direct communication, enabling agent-to-agent replies, and supporting event-based message delivery with callbacks or listener patterns to reduce polling overhead.

Lab 5:

Source Code:

Client.java

```
import java.io.*;
import java.net.InetAddress;
import java.rmi.*;

public class Client {

    public static void main( String args[] ) {
    // verify arguments
    int port = 0;
    try {
        if ( args.length == 2 ) {
        port = Integer.parseInt( args[1] );
        if ( port < 5001 || port > 65535 )
            throw new Exception( );
        }
        else
        throw new Exception( );
    } catch ( Exception e ) {
        System.err.println( "usage: java Client serverIp port" );
        System.exit( -1 );
    }
    String serverIp = args[0];

    try {
            // Get the local hostname
            String hostname = InetAddress.getLocalHost().getHostName();
            ServerInterface server = (ServerInterface) Naming.lookup("rmi://" + serverIp + ":"
+ port + "/Server");
            ClientInterface client = new ClientImplementation();
            server.registerForCallback(client); // Register the client for callbacks

            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            String line;
            while ((line = br.readLine()) != null) {
                server.broadcast(hostname + ": " + line, client);
            }

            server.unregisterForCallback(client);
            System.out.println("Unregistered and exiting.");
```

```
        } catch (Exception e) {
            System.out.println("Client exception: " + e);
        }

    }
}
```

ServerImplementation.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class ServerImplementation extends UnicastRemoteObject
    implements ServerInterface {


    private final ArrayList<ClientInterface> clientList;
    public ServerImplementation( ) throws RemoteException {
        super( );
        clientList = new ArrayList<>();
    }

    public String greetings( ) throws RemoteException {
        return( "hello" );
    }

    public synchronized void registerForCallback( ClientInterface client )
    throws RemoteException {

        if ( !clientList.contains( client ) ) {
            clientList.add( client );
            System.out.println( "Registered: " + client );
        }
    }

    public synchronized void unregisterForCallback( ClientInterface client )
    throws RemoteException {
        if ( clientList.remove( client ) ) {
            System.out.println( "Unregistered: " + client );
        } else {
            System.out.println( "Didnot unregister: " + client );
        }
```

```
    }

    // Broadcast message to all clients except the sender
    public synchronized void broadcast(String message, ClientInterface sender) throws
RemoteException {
        for (ClientInterface client : new ArrayList<>(clientList)) {
            try {
                    if (!client.equals(sender)) {
                        client.receiveMessage(message); // Send message to all clients except
the sender
                    }
            } catch (Exception e) {
                clientList.remove(client); // auto-cleanup dead clients
                System.out.println("Removed a dead client.");
            }
        }
    }
}
```

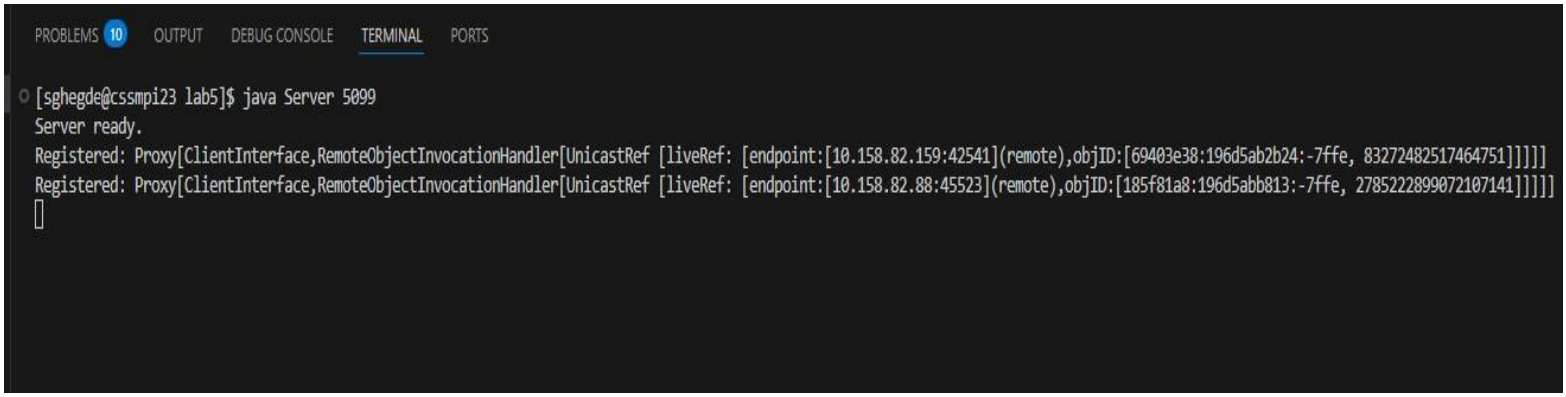
Server.java:

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
import java.io.*;

public class Server {
    public static void main( String args[] ) {
    // verify arguments
    int port = 0;
    try {
        if ( args.length == 1 ) {
        port = Integer.parseInt( args[0] );
        if ( port < 5001 || port > 65535 )
            throw new Exception( );
        }
        else
        throw new Exception( );
    } catch ( Exception e ) {
        System.err.println( "usage: java Server port" );
        System.exit( -1 );
    }
```

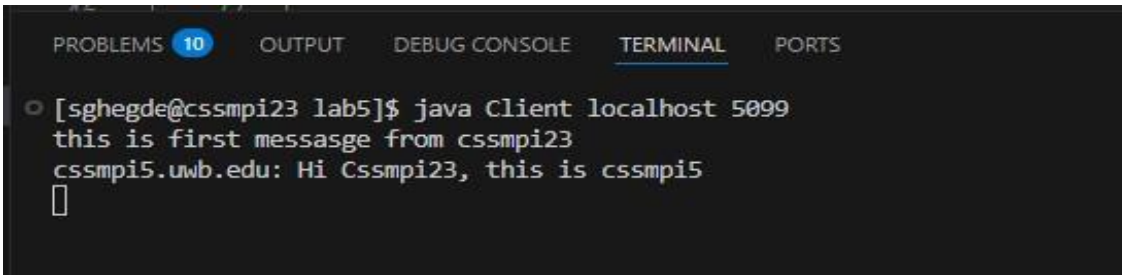
```
    try {
        startRegistry( port );
        ServerImplementation serverObject = new ServerImplementation( );
        Naming.rebind("rmi://localhost:" + port + "/Server", serverObject);
        System.out.println( "Server ready." );
    } catch ( Exception e ) {
        e.printStackTrace( );
        System.exit( -1 );
    }
    }

    private static void startRegistry( int port ) throws RemoteException {
    try {
        Registry registry =
        LocateRegistry.getRegistry( port );
        registry.list( );
    }
    catch ( RemoteException e ) {
        Registry registry =
        LocateRegistry.createRegistry( port );
    }
    }
}
```

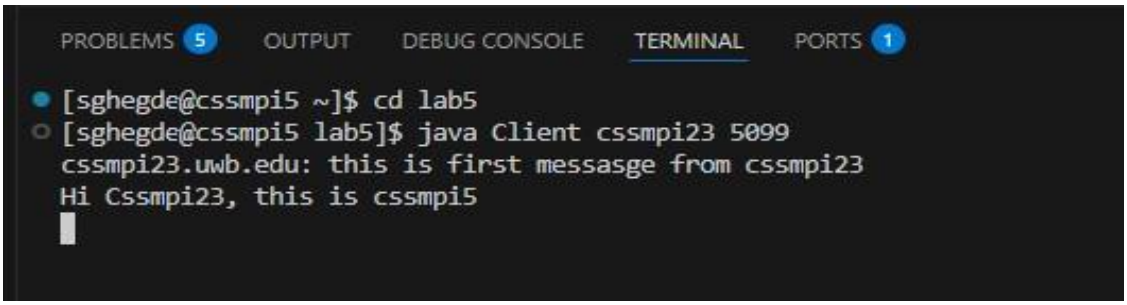
Outputs:

Server:



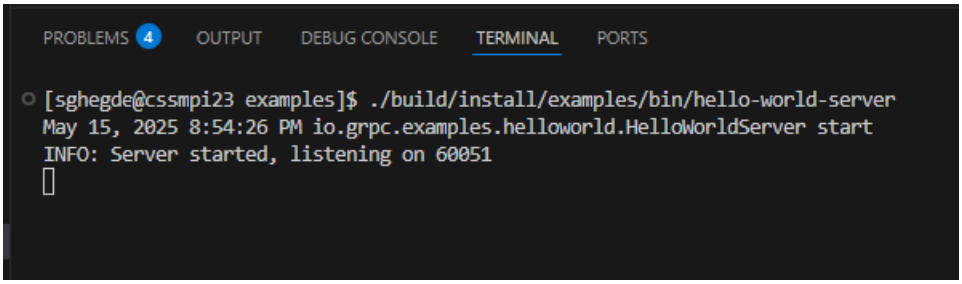
Clients:


PROBLEMS 10
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
[sghegde@cssmpi23 lab5]$ java Client localhost 5099
this is first messasge from cssmpi23
cssmpi5.uwb.edu: Hi Cssmpi23, this is cssmpi5

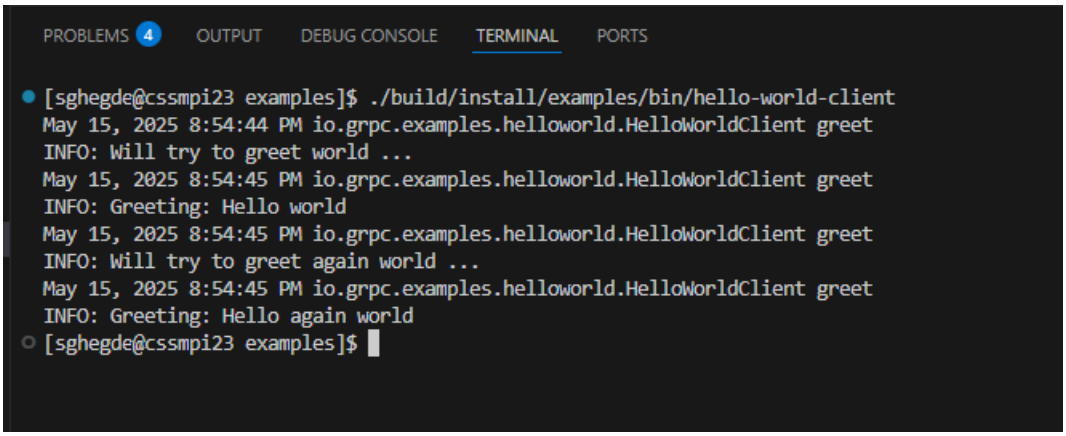

PROBLEMS 5
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS 1
[sghegde@cssmpi5 ~]$ cd lab5
[sghegde@cssmpi5 lab5]$ java Client cssmpi23 5099
cssmpi23.uwb.edu: this is first messasge from cssmpi23
Hi Cssmpi23, this is cssmpi5

Lab 6:

Server:



Client:



Lab 7:

Source Code:

MyAgent.java

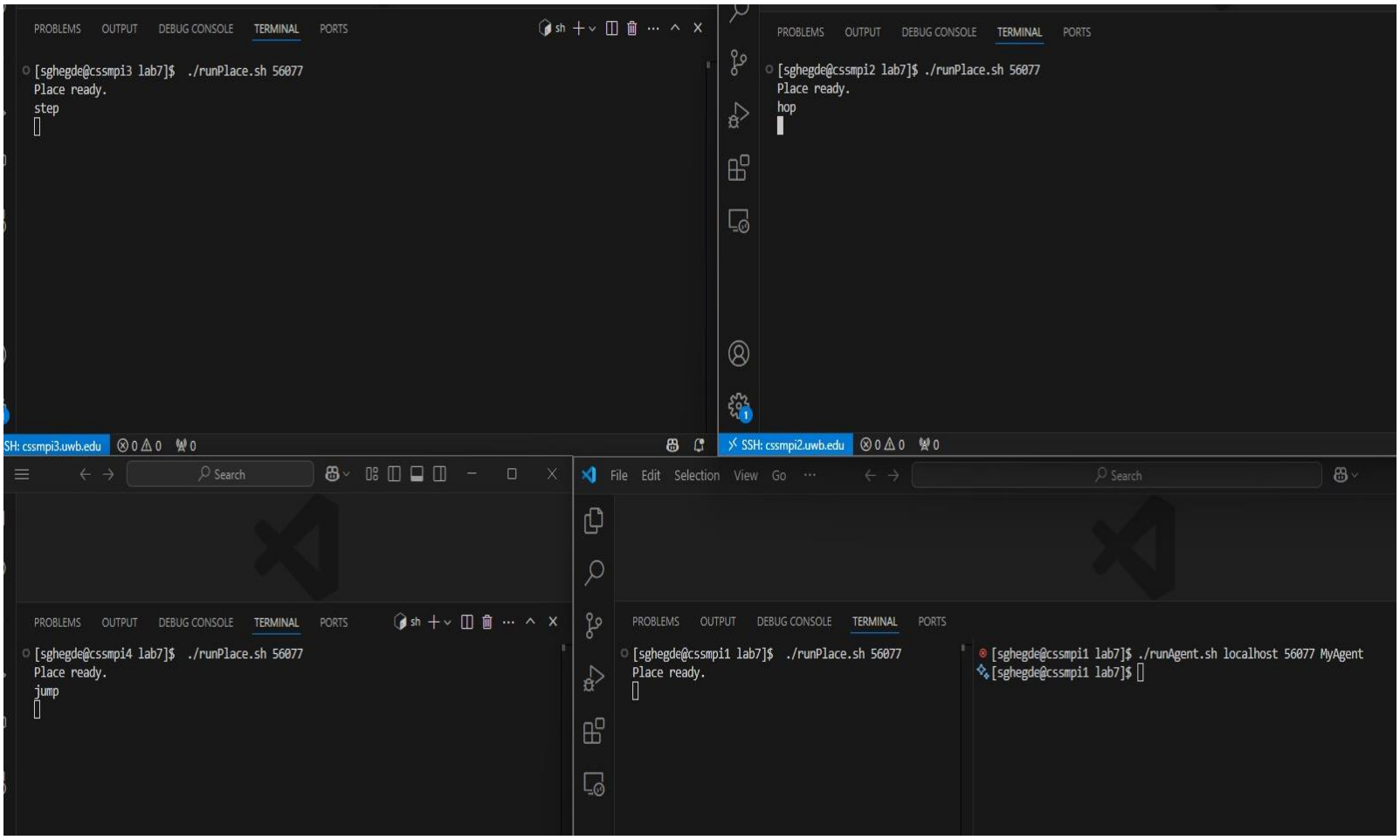
```
import Mobile.*;

public class MyAgent extends Agent {
    public int hopCount = 0;
    public String[] destination = new String[] { "cssmpi2", "cssmpi3", "cssmpi4" };

    /**
     * Constructor for MyAgent.
     *
     * @param args arguments passed from the command line.
     */
    public MyAgent( ) {

    }

    /**
     * init( ) is the default method called upon an agent inject.
     */
    public void init( ) {
        String[] args = new String[1];
        args[0] = "hop";

        hop(destination[hopCount], "step", args);

    }

    /**
     * step( ) is invoked upon an agent migration to destination[0] after
     * init( ) calls hop( ).
     *
     * @param args arguments passed from init( ).
     */
    public void step( String[] args ) {
        hopCount++;
        System.out.println( args[0] );
        args[0] = "step";
        hop(destination[hopCount], "jump", args);
```

```
    }

    /**
     * jump( ) is invoked upon an agent migration to destination[1] after
     * step( ) calls hop( ).
     *
     * @param args arguments passed from step( ).
     */
    public void jump( String[] args ) {
        hopCount++;
        System.out.println( args[0] );
        args[0] = "jump";
        hop(destination[hopCount], "fall", args);


    }


    public void fall( String[] args ) {
        hopCount++;
        System.out.println( args[0] );
    }


}
```

Output:

In the next page. Cssmpi1 is the starting point. Then follows incrementally as shown in the example.

cssmpi1 -> cssmpi2 -> cssmpi3-> cssmpi4.


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[sghegde@cssmpi3 lab7]$ ./runPlace.sh 56077
Place ready.
step
SH: cssmpi3.uwb.edu
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[sghegde@cssmpi2 lab7]$ ./runPlace.sh 56077
Place ready.
hop
SSH: cssmpi2.uwb.edu
Search
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[sghegde@cssmpi4 lab7]$ ./runPlace.sh 56077
Place ready.
jump
File Edit Selection View Go
Search
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[sghegde@cssmpi1 lab7]$ ./runPlace.sh 56077
Place ready.
[sghegde@cssmpi1 lab7]$ ./runAgent.sh localhost 56077 MyAgent
[sghegde@cssmpi1 lab7]$