

## AC2 Distributed Mutual Exclusion

Lucas Azcue

Alex Blay

## Main

```
public class Main {
    Lucas Azcue Abello
    public static void main(String[] args) throws IOException {
        int NUM_LIGHTWEIGHTS = 3;
        int NUM_HEAVYWEIGHTS = 2;
        int[][] lwPorts = {{6868, 6869, 6870},{6872, 6873, 6874}};
        int[] hwPorts = {6871, 6867};
        Thread[] lwThreadA = new Thread[3];
        Thread[] lwThreadB = new Thread[3];
        Thread[] hwThread = new Thread[2];

        for(int i = 0; i < NUM_HEAVYWEIGHTS; i++) hwThread[i] = new Thread(new HeavyweightThread(i, initialToken: i == 0, lwPorts[i], hwPorts[i]));

        for(int i = 0; i < NUM_LIGHTWEIGHTS; i++) lwThreadA[i] = new Thread(new LightweightThread(i, lamportMutexFlag: true));
        for(int i = 0; i < NUM_LIGHTWEIGHTS; i++) lwThreadB[i] = new Thread(new LightweightThread(i, lamportMutexFlag: false));

        for(int i = 0; i < NUM_HEAVYWEIGHTS; i++) hwThread[i].start();

        try{
            Thread.sleep(6000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        for(int i = 0; i < NUM_LIGHTWEIGHTS; i++) lwThreadA[i].start();
        for(int i = 0; i < NUM_LIGHTWEIGHTS; i++) lwThreadB[i].start();
    }
}
```

As we can see from our main method, we have a very simple creation of the different threads that represent the Heavyweights and Lightweights that will compose our exercise. The ports are hardcoded beforehand to our chosen values.

The lightweightThread has a Boolean parameter that determines if the process is A or B, to have a Lamport Mutex or a RAMutex. The heavyweightThread has a similar Boolean property that initializes the token value so that it can give access to its lightweights first.

It is important to note that we have a 6 second wait between the start of the heavyweight threads and the lightweights. The reason for this is that to guarantee that the first accept is from the heavyweight we need to give it this time advantage. This way we can have the first accept for every lightweight be the heavyweight. This is important because there are different behaviors when receiving from another lightweight or a heavyweight. Maybe 6 seconds won't be necessary, but this time penalty will only happen once and we can guarantee the integrity of the sockets.

## LightweightThread

```

public class LightweightThread implements Runnable {
    2 usages
    private LightweightProcess process;
    3 usages
    private LockWithSockets mutex;

    2 usages  Lucas Azcue Abello
    public LightweightThread(int threadID, boolean lamportMutexFlag) throws IOException {
        if(lamportMutexFlag) mutex = new LamportMutex(threadID);
        else mutex = new RAMutex(threadID);

        this.process = new LightweightProcess(threadID, mutex);
    }

    Lucas Azcue Abello
    @Override
    public void run() {
        try {
            process.lightweight();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}

```

With this class we define the thread behavior, which will call to the LightweightProcess. Notice that depending on the lamportMutexFlag value the mutex will be a lamportMutex or a RAMutex.

## LightweightProcess

This method defines the behavior in the statement. It will have the same behavior for both mutex types, which will be passed as a parameter. We will execute the create socket functionality which makes the setup for the connections. We will also create a different thread that calls the ClientHandler method, through it we will receive messages through the lightweight sockets continuously independently of the process behavior. After this we have a sleep that makes sure the socket creation has enough time to set up appropriately. From there on we enter an infinite loop that waits for the start signal from the heavyweight, and after does a request to manage access to the CS. All the lightweights will have this same behavior, the only difference is that the RAMutex will have a different request, release and handle message implementation. The separation between heavyweight and lightweight sockets allows us to manage them separately. The print is conditioned by an if statement that checks if mutex is an instance of LamportMutex (A) or RAMutex (B).

The messages will have the following format between lightweights both for A and B functionalities:

```
{ "timestamp" "src" "tag" }
```

## HeavywayThread

```
1 usage  ↳ Lucas Azcue Abello
public class HeavyweightThread implements Runnable {
    2 usages
    private final HeavyweightProcess process;

    1 usage  ↳ Lucas Azcue Abello
    public HeavyweightThread(int processID, boolean initialToken, int[] lwPorts, int hwPort) throws IOException {
        process = new HeavyweightProcess(processID, initialToken, lwPorts, hwPort);
    }

    ↳ Lucas Azcue Abello
    @Override
    public void run() {
        try {
            process heavyweight();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The heavyweightProcess has a very simple functionality. It just connects the Runnable functionality with the HeavyweightProcess class.

## HeavyweightProcess

The heavyweightProcess starts by creating the sockets. We see that first we manage the heavyweights before waiting for some seconds to make sure that the hwSocket corresponds to the heavyweight. After that it manages all the lightweight connections, notice that these do not need the time delay because we do not need to distinguish between them, each message will have its src in the message. If we did not send the ID, then a time delay will have to be applied to separate connections.

The heavyweight does not need a separate thread to receive message. Its own behavior has defined listening functionalities to check when a lightweight has finished and when it receives the token from the other heavyweight.

For heavyweight messages the format will be the following: {"start/finish/token"}. We do not need to know the src of the message.

## ClientHandler

```

1 usage  Lucas Azcue Abello
public void readSocket(){
    try {
        // Read data from the client
        for(int i = 0; i < num_clients; i++){
            if(this.mutex.getSockets().serverSocketReceive[i].getInputStream().available() > 0) {
                String line = utils.readMessage(this.mutex.getSockets().serverSocketReceive[i]);
                String[] parts = line.split(regex: " ");
                this.mutex.handleMsg(Integer.parseInt(parts[0]), Integer.parseInt(parts[1]), parts[2]);
            }
        }
        // Close the socket
        //clientSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

As we can see, the ClientHandler reads messages from the different client sockets and splits them into its different components to pass them to the handleMsg. Both mutex types have a handleMsg functionality and so by extending the LockWithSockets class we can reuse the ClientHandler for both lightweight processes.

## Lamport Mutex

```

public LamportMutex(int myID) throws IOException {

    clock = new DirectClock(N, myID);
    requestQ = new int[N];
    numOkay = 0;
    this.myID = myID; //Add process ID
    for (int i = 0; i < N; i++) {
        requestQ[i] = Integer.MAX_VALUE;
    }
    sockets = new LwSockets( hwPort: 6867, myID);
    msg = new Msg( numMessages: N - 1);
}

```

As we can see, the Lamport Mutex uses a direct clock of size N. With a request array of the same size that stores the value of the request timestamp. The values are initialize to infinite, in this case that will be the maximum for the Integer data type. In addition, we also initialize the Msg and LwSockets properties that we will use during the synchronization of lightweight processes.

```

public synchronized void requestCS() throws InterruptedException {
    clock.tick(); //Increase timer before sending event

    requestQ[myID] = clock.getValue(myID);
    String message = requestQ[myID] + " " + this.myID + " request";
    msg.broadcastMsg(message, this.sockets);
    while (!okayCS()) Thread.sleep(1000); //Thread.sleep(1000) or wait?
}

1 usage  ▲ Lucas Azcue Abello +1
public synchronized void releaseCS() {
    clock.tick(); // Increase timer before sending event
    requestQ[myID] = Integer.MAX_VALUE;
    String message = clock.getValue(myID) + " " + this.myID + " release";
    msg.broadcastMsg(message, this.sockets);
    numOkay = 0;
}

1 usage  ▲ ablaysitges +1
private boolean okayCS() {

    if (numOkay < N - 1) return false;

    for (int i = 0; i < N; i++) {
        if (isGreater(requestQ[myID], myID, requestQ[i], i)) {
            return false;
        }
        if (isGreater(requestQ[myID], myID, clock.getValue(i), i)){
            return false;
        }
    }
    return true;
}

```

The request method uses a sleep instead of a wait. This could be changed to better optimize the code, but the functionality is the same. Until the client handler receives all the necessary acknowledgements and checks the other conditions related to the okay state, the lightweight process will wait while the client handler continues to read messages.

In the release we return our array position in requestQ to infinity. We then send a release broadcast message so the other processes can do the same in order for the other processes to comply with the okayCS() conditions.

```

1 usage  Lucas Azcue Abello +1
public void handleMsg(int timestamp, int src, String tag) {

    clock.receiveAction(src, timestamp); // Sync clocks (Increase timer in reception)

    tag = tag.trim();

    if (tag.equals("request")) {

        this.requestQ[src] = timestamp;

        int socketPosition = src;

        if(src > this.myID) socketPosition--;

        String message = clock.getValue(myID) + " " + this.myID + " ack";
        msg.sendMsg(this.sockets.clientSocket[socketPosition], message);

    } else if (tag.equals("ack")) {
        this.numOkay++;
    } else if (tag.equals("release")) {
        this.requestQ[src] = Integer.MAX_VALUE;
    }
}
}

```

In the case of the hanldeMsg we see that its pretty simple. We update our clock with the new timestamp received. If a request message is received we update the requestQ position of the sender to this timestamp. We then use the socketPosition to check which of the two sockets corresponds to the sender, so that an ACK message can be returned.

In the case of an ACK received, we increase the number of okays. And if a release is received we understand that the timestamp in the requestQ for that process is obsolete and thus needs to be updated to infinity (Integer.max\_value).

These methods are different for algorithms, but the process functionality uses the same methods. That is why we created an interface, so that both instances of the mutex class can use these methods for the two different methodologies.

## RAMutex

```

1 usage  Lucas Azcue Abello
public RAMutex(int myID) throws IOException {
    myTS = Integer.MAX_VALUE;
    clock = new LamportClock();
    pendingQ = new LinkedList<>();

    this.myID = myID;
    this.msg = new Msg( numMessages: N-1);
    this.sockets = new LwSockets( hwPort: 6871, myID);
}
}

```

As we can see the constructor process is the same, we just use a Lamport Clock and a Linked List to store pending requests.

```
1 usage  ➤ Lucas Azcue Abello
public synchronized void requestCS() throws InterruptedException {
    clock.tick();
    myTS = clock.getValue();
    String message = myTS + " " + this.myID + " request";
    this.msg.broadcastMsg(message, sockets);
    while(numOkay < N - 1) wait();
}

1 usage  ➤ Lucas Azcue Abello
public synchronized void releaseCS() {
    myTS = Integer.MAX_VALUE;
    numOkay = 0;
    clock.tick();
    while(!pendingQ.isEmpty()) {
        int pid = pendingQ.remove();
        int socketPosition = pid;
        if(pid > this.myID) socketPosition--;
        String message = myTS + " " + this.myID + " okay";
        this.msg.sendMsg(this.sockets.clientSocket[socketPosition], message);
    }
}
```

The request and release method is shared for both algorithms. In this case we do implement the wait() function, to show that both have the same functionality.

In the case of the release we do not do a broadcast message, but send an okay to the waiting processes we are aware of through the pendingQ. Notice that we use the same functionality as in type A to determine the socket that corresponds to the ID of the lightweight.

```
1 usage  ➤ Lucas Azcue Abello
public synchronized void handleMsg(int timestamp, int src, String tag) {
    clock.receiveAction(timestamp);
    if (tag.trim().equals("request")) {
        if((myTS == Integer.MAX_VALUE) || (timestamp < myTS) || ((timestamp == myTS) && (src < myID))) {
            int socketPosition = src;
            if(src > this.myID) socketPosition--;
            String message = clock.getValue() + " " + this.myID + " okay";
            this.msg.sendMsg(this.sockets.clientSocket[socketPosition], message);
        } else {
            pendingQ.add(src);
        }
    } else if (tag.trim().equals("okay")) {
        numOkay++;
        if (numOkay == N - 1) notify();
    }
}
```

The handleMsg is where the priority is checked in the case of the RA algorithm. Instead of sending acknowledgments to everyone and then checking the requests timestamp, the okays



will only be sent to that process which has priority and thus can access the CS before. If it has lower priority we will add it to the pendingQ and send the okay once we have finished.

In this case as we have used the wait(), we will have to use the notify(). Both of these methods work together.

## LwSockets

```
public class LwSockets {
    4 usages
    private final int myID;
    4 usages
    private final int hwPort;
    2 usages
    private final String HOSTNAME = "127.0.0.1";
    2 usages
    private final int N = 2;
    3 usages
    public ServerSocket serverSocket;
    4 usages
    public Socket[] serverSocketReceive;
    3 usages
    public Socket hwSocketReceive;
    2 usages
    public Socket hwSocket;
    6 usages
    public Socket[] clientSocket;
    1 usage
    private Utils utils;

    2 usages  ▲ Lucas Azcue Abello
    public LwSockets(int hwPort, int myID) throws IOException {
        utils = new Utils();

        this.hwPort = hwPort;
        this.myID = myID;

        this.serverSocketReceive = new Socket[2];
        this.clientSocket = new Socket[2];

        //Create server socket
        this.serverSocket = new ServerSocket( port: this.hwPort + this.myID + 1);
    }
}
```

We have created a class that has all the sockets that a lightweight will use. In the constructor we only create our own server socket, all of the other sockets will be assigned in the createSockets method.

```

public void createSockets() {
    //Create server socket
    try {
        //make sure heavyweight connects first between them
        this.hwSocket = new Socket(this.HOSTNAME, this.hwPort);
        this.hwSocketReceive = this.serverSocket.accept();

        Thread.sleep( millis: 3000);

        int socketValue = 0;
        //Connect to other sockets
        for (int i = 0; i < N; i++) {
            if (i == this.myID) socketValue++;
            this.clientSocket[i] = new Socket(this.HOSTNAME, port: this.hwPort + socketValue + 1);
            socketValue++;
        }

        Thread.sleep( millis: 1000 * this.myID);

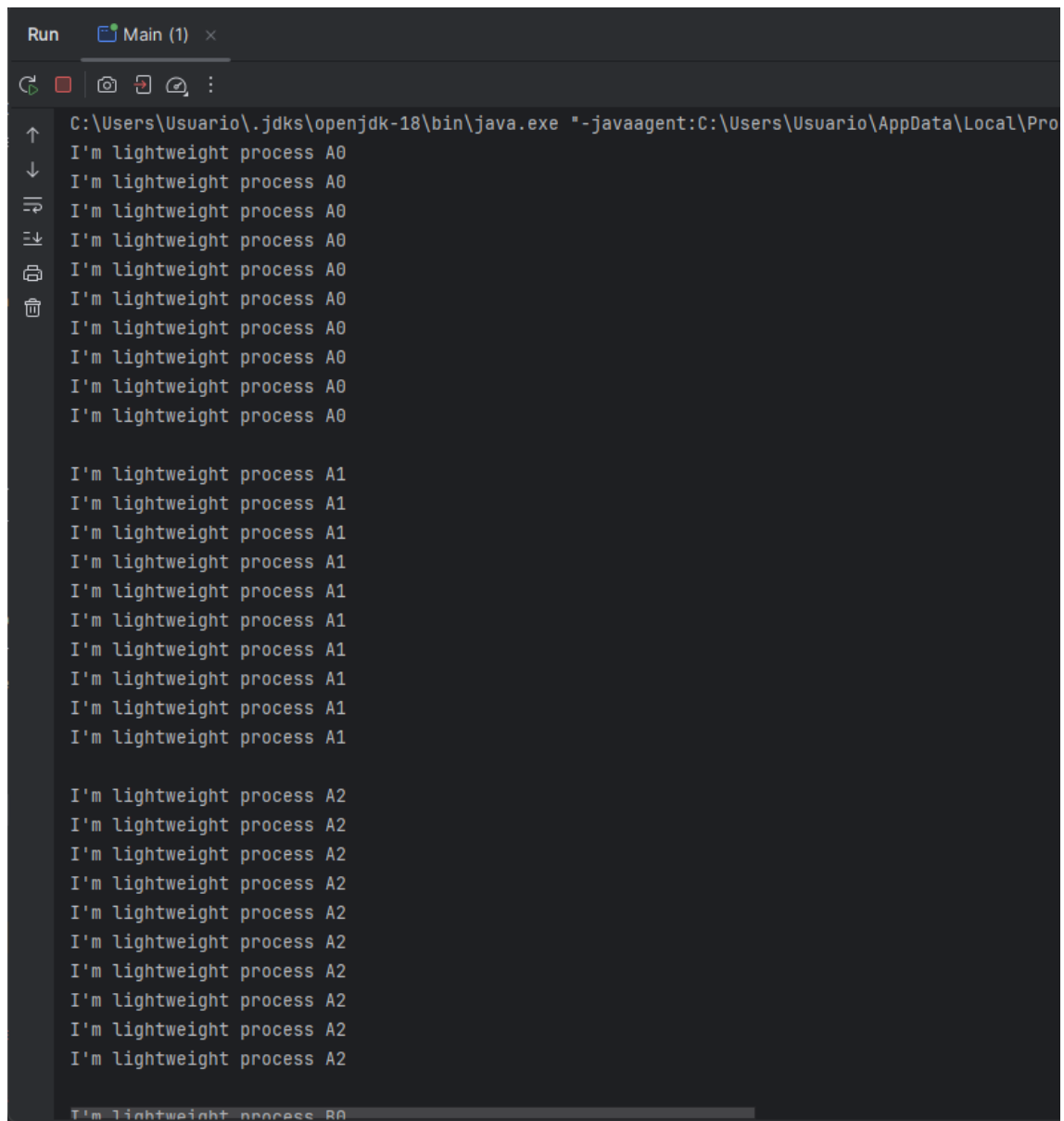
        for (int i = 0; i < N; i++) {
            this.serverSocketReceive[i] = this.serverSocket.accept();
        }
    } catch (InterruptedException | IOException e) {
        throw new RuntimeException(e);
    }
}

```

This method connects all the sockets for a lightweight. Notice that we first connect the heavyweight socket and then wait to give it a margin for the heavyweight connection to establish and differentiate with the lightweight sockets. This is important because we listen to the hw and lw sockets in different places of the process and can create issues if we do not give it enough time. The same thing happens to each lw process, where to have the socket ID match with the desired lw ID we have to give the first lightweight an advantage.

## Results

The code has a several seconds delay at first to make sure all sockets connections work accordingly, and after the different processes start communicating to guarantee that the CS is accessed only one at a time.

A screenshot of an IDE's Run console window. The title bar shows 'Run' and 'Main (1)'. The console output displays the execution of a Java program. The first line is the command: 'C:\Users\Usuario\.jdk\openjdk-18\bin\java.exe "-javaagent:C:\Users\Usuario\AppData\Local\Pro'. The output consists of three groups of messages: 10 lines of 'I'm lightweight process A0', 10 lines of 'I'm lightweight process A1', and 10 lines of 'I'm lightweight process A2'. The last line, 'I'm lightweight process A0', is partially cut off at the bottom of the console.

```
Run Main (1) ×
C:\Users\Usuario\.jdk\openjdk-18\bin\java.exe "-javaagent:C:\Users\Usuario\AppData\Local\Pro
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0
I'm lightweight process A0

I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1
I'm lightweight process A1

I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2
I'm lightweight process A2

I'm lightweight process A0
```

This is the results of our program, as we can see everything works perfectly.