```java
package lamport;

import utils.Constants;

import java.net.MalformedURLException;
import java.rmi.AlreadyBoundException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * This class defines a remote object of the {@link IValueManager} implementation.
 *
 * DESCRIPTION:
 * When the {@link ValueManager} is launched, arguments of the main program are parsed first.
 * The first argument argument (args[0]) is the port on which the current {@link ValueManager} is listening on.
 * The second parameter (args[1]) is the number N of the nodes of the system.
 * The next N-1 arguments are the ports on which the other {@link ValueManager}s are listening on.
 * Every {@link ValueManager} must know the other nodes' ports to execute the mutual exclusion Lamport algorithm
 * when several nodes send the value modification requests.
 *
 * LAMPORT:
 * - When the client demands the modification of the value stored in the {@link IValueManager} attributed to him,
 * the {@link ValueManager} first pushes the request to the request queue and informs all other nodes of this request,
 * so that they also store it in their proper request queues.
 * - When a {@link ValueManager} recieves such a modification request, it responds to the emitter with an
 * acknowledgement message.
 * - Once the {@link ValueManager} collects N-1 acknowledgements, it checks if it is able to get the critical section
 * and change the value. It is the case when it's request is the oldest one in the request queue. If it is not,
 * {@link ValueManager} continues waiting for it's turn.
 * - When the value is modified by the {@link ValueManager}, the critical section is liberated and the the liberation
 * messages are diffused to other nodes of the system. Once this message is received by a node, it also checks if it
 * it's turn to get the critical section.
 *
 * Authors: Samuel Mayor, Alexandra Korukova
 */
public class ValueManager extends UnicastRemoteObject implements IValueManager {

    private static final Logger LOG = Logger.getLogger(ValueManager.class.getName());

    /**
     * The value stored by {@link ValueManager}
     */
    private int value;

    /**
     * Local timestamp which is updated with every incoming/emitted {@link Message}
     */
    private int localTime;

    /**
     * The port on which {@link ValueManager} is listening for the incoming requests
     */
    private int port;

    /**
     * Number of the nodes of the system
     */
    private int nbNodes;

    /**
     * Ports of all other {@link ValueManager}s of the system
     */
    private int[] ports;

    /**
     * This map associates the port with the {@link IValueManager} which is listening on this port
     */
    private Map<Integer, IValueManager> portManager;

    /**
     * This map represents a request queue for the Lamport algorithm
     * The keys of the map are the port of the {@link IValueManager} holding the request
     * The values are the {@link Message}s containing the request
```

```
81       */
82      private Map<Integer, Message> pendingRequests;
83
84      /**
85       * Current number of acknowledgements received by the {@link ValueManager}
86       */
87      private int nbAcks;
88
89      /**
90       * This boolean is true if the current node has requested the access to the critical section
91       */
92      private boolean criticalSectionRequested;
93
94      /**
95       * In case if the critical section is requested by the {@link ValueManager}, this integer stores the
    value
96       * which the {@link ValueManager} desires to set
97       */
98      private int newValue;
99
100     /**
101      * @param args
102      *       * args[0] - the port on which the registry accepts the requests
103      *       * args[1] - number of the nodes of the system - N
104      *       * args[2]..args[N-1] - ports of the other {@link ValueManager}s of the system
105      * @throws RemoteException
106      * @throws AlreadyBoundException
107      */
108     public static void main(String ...args) throws RemoteException, AlreadyBoundException {
109
110         // parse the main arguments
111         int port = Integer.parseInt(args[0]);
112         int nbNodes = Integer.parseInt(args[1]);
113         int[] ports = new int[nbNodes-1];
114         for(int i  = 0; i < nbNodes-1; i++) {
115             ports[i] = Integer.parseInt(args[i+2]);
116         }
117
118         // create and exports a Registry instance on the localhost that accepts requests
119         Registry registry = LocateRegistry.createRegistry(port);
120         // bind the remote reference to the name in the registry
121         registry.bind(Constants.REMOTE_OBJ_NAME, new ValueManager(port, nbNodes, ports));
122         LOG.log(Level.INFO, () -> Constants.REMOTE_OBJ_NAME + " bound");
123         LOG.log(Level.INFO, () -> "Listening on incoming remote invocations on port: " + port);
124
125     }
126
127     /**
128      * Constructor
129      * @param port the port the {@link ValueManager} is listening on
130      * @param nbNodes total number of the nodes ({@link client.Site} - {@link ValueManager} couples) of the
     system: N
131      * @param ports the array containing N-1 ports of other {@link ValueManager}s of the system
132      * @throws RemoteException
133      */
134     public ValueManager(int port, int nbNodes, int[] ports) throws RemoteException {
135         super();
136         localTime = 0;
137         this.port = port;
138         this.nbNodes = nbNodes;
139         this.ports = ports;
140         pendingRequests = new TreeMap<>();
141         portManager = new HashMap<>();
142         criticalSectionRequested = false;
143     }
144
145     /**
146      * This method is used to link the current {@link ValueManager} with the other fot the further
    execution of the
147      * Lamport mutual exclusion algorithm
148      * @throws RemoteException
149      * @throws NotBoundException
150      * @throws MalformedURLException
151      */
152     public void lookup() throws RemoteException, NotBoundException, MalformedURLException {
153         // reference the other ValueManagers of the system
154         for (int p : ports) {
155             String toLookup = "rmi://" + Constants.SERVER_HOST
156                     + ":" + p + "/" + Constants.REMOTE_OBJ_NAME;
157             Registry registry = LocateRegistry.getRegistry(Constants.SERVER_HOST);
158             IValueManager manager = (IValueManager) Naming.lookup(toLookup);
159             portManager.put(p, manager);
160             LOG.log(Level.INFO, () -> Constants.REMOTE_OBJ_NAME + " is linked with other nodes of the
    system");
161         }
162     }
163
164     /**
```

```
165      * The implementation of the remote method
166      * @throws RemoteException
167      */
168     @Override
169     public void setValue(int value) throws RemoteException {
170         sendRequest(value);
171         this.value = value;
172     }
173 
174     /**
175      * This method is called when the user desires to modify the value stored by {@link ValueManager}s.
176      * Pushes the request (as a {@link Message}) to the request queue and sends the request messages to
     other
177      * {@link ValueManager}s of the system.
178      * @param newValue the value to set
179      * @throws RemoteException
180      */
181     private void sendRequest(int newValue) throws RemoteException {
182         localTime++;
183         nbAcks = 0;
184         criticalSectionRequested = true;
185         this.newValue = newValue;
186         Message requestMsg = new Message(localTime, MessageType.REQUEST, port);
187         pendingRequests.put(port, requestMsg);
188 
189         LOG.log(Level.INFO, () -> localTimeStr() + "Sending the "
190                 + requestMsg.getMessageType().name()  + " to other nodes");
191         Iterator it = portManager.entrySet().iterator();
192         while (it.hasNext()) {
193             Map.Entry pair = (Map.Entry)it.next();
194             IValueManager manager = (IValueManager) pair.getValue();
195             manager.acceptMessage(requestMsg);
196         }
197     }
198 
199     /**
200      * The implementation of the remote method
201      * @return the value
202      * @throws RemoteException
203      */
204     @Override
205     public int getValue() throws RemoteException {
206         return value;
207     }
208 
209     /**
210      * The implementation of the remote method
211      * @param message the {@link Message} sent by some remote {@link IValueManager}
212      * @throws RemoteException
213      */
214     public void acceptMessage(Message message) throws RemoteException {
215         LOG.log(Level.INFO, () -> localTimeStr() + "message received");
216         updateLocalTime(message.getTimestamp());
217         LOG.log(Level.INFO, () -> localTimeStr()
218                 + message.getMessageType().name()
219                 + " from " + message.getEmitterPort());
220         switch (message.getMessageType()) {
221             case REQUEST:
222                 pendingRequests.put(message.getEmitterPort(), message);
223                 portManager.get(message.getEmitterPort()).acceptMessage(
224                         new Message(localTime, MessageType.ACKNOWLEDGEMENT, port));
225                 break;
226             case ACKNOWLEDGEMENT:
227                 nbAcks++;
228                 checkCriticalSection();
229                 break;
230             case LIBERATION:
231                 pendingRequests.remove(message.getEmitterPort());
232                 LiberationMessage libMessage = (LiberationMessage) message;
233                 this.value = libMessage.getNewValue();
234                 checkCriticalSection();
235                 break;
236             default:
237                 LOG.log(Level.SEVERE, "Unknown message type");
238         }
239     }
240 
241     /**
242      * This method checks if the current {@link ValueManager} can get the access to the critical section.
243      * If it does, updates the value stored by all the {@link ValueManager}s of the system and sends the
     liberation
244      * messages to them.
245      * @throws RemoteException
246      */
247     private void checkCriticalSection() throws RemoteException {
248         if(criticalSectionRequested && nbAcks == nbNodes-1) {
249             int localRequestTime = pendingRequests.get(port).getTimestamp();
250             for (Map.Entry<Integer, Message> entry : pendingRequests.entrySet()) {
```

```
251                 // check if the local request is the oldest one
252                 boolean localTimeGreater = localRequestTime > entry.getValue().getTimestamp();
253                 // if there are more than one request with the same timestamps, the one with the smaller id
    (port) will
254                 // get the access to the critical section
255                 boolean localTimeEqualsIdGreater = (localRequestTime == entry.getValue().getTimestamp())
256                         && (port > entry.getValue().getEmitterPort());
257                 if (localTimeGreater || localTimeEqualsIdGreater) {
258                     return;
259                 }
260             }
261             // the current ValueManager's requests gets the access to the critical section
262             // the local request is the oldest one
263             LOG.log(Level.INFO, "Entering in the critical section");
264             this.value = newValue;
265             pendingRequests.remove(port);
266             criticalSectionRequested = false;
267             nbAcks = 0;
268             // inform other nodes of the system
269             LOG.log(Level.INFO, () -> localTimeStr() + "Updating value in other nodes");
270             Iterator it = portManager.entrySet().iterator();
271             while (it.hasNext()) {
272                 Map.Entry pair = (Map.Entry)it.next();
273                 IValueManager manager = (IValueManager)pair.getValue();
274                 manager.acceptMessage(new LiberationMessage(localTime, MessageType.LIBERATION, port,
    newValue));
275             }
276             LOG.log(Level.INFO, () -> localTimeStr() + "Value updated, new value: " + value);
277         }
278     }
279
280     /**
281      * Updates the logical timestamp oh the {@link ValueManager} when the new message is received by the
282      * {@link ValueManager}.
283      * Used to identify the oldest request by the Lamport algorithm
284      * @param remoteTimestapm the timestamp of the {@link Message} received
285      */
286     private void updateLocalTime(int remoteTimestapm) {
287         localTime =  Math.max(localTime+1, remoteTimestapm+1);
288     }
289
290     /**
291      * Local timestamp String representation for log printing
292      * @return ocal timestamp String representation
293      */
294     private String localTimeStr() {
295         return "[" + localTime + "] ";
296     }
297
298 }
299
```