Category "jms 2010/2011"

Q1 wisdoms at peer C (10 pts)

At the end of exercise **2.2.(f)**, cut and paste to the box below all wisdoms shown by peer C.

```
A@klokan: AAA
A@klokan: second wisdom
```

Q2 line sending response to NewNeighbour (10 pts)

In the box below, copy and paste the line that you identified in exercise **2.3.(b)** together with the 6 lines that precede it.

It is the command on line 139 in file `Dispatcher.java`:

```
            String senderName = nn.getMostRecentSenderName();

            // respond to the sender:
            try
            {
                neighbours.sendObjectTo(senderName, nn.constructResponse(myQueueName));
```

Q3 method waiting for response (10 pts)

In the box below, name the class and method that you identified in exercise **2.3.(c)**.

(100) equals (case sensitive):

`Dispatcher.awaitResponse`

Q4 line notifying of response (10 pts)

In the box below, write the *file name* and copy and paste the *line* that you identified in exercise **2.3.(d)** together with the 6 lines that precede it.

It is the command on line 116 in file `Dispatcher.java`:

```
        if (obj instanceof WMessageResponse)
        {
            WMessageResponse response = (WMessageResponse) obj;
            synchronized (this)
            {
                responses.put(response.getReplyTo(), response);
                this.notifyAll();
```

Q5 add via message (15 pts)

Accurately describe changes you made to the given peer code during exercise **2.4.(a)**.

For each change indicate which file and method it is in, and cut and paste the altered lines together with three preceding and three following lines of original code to easily identify the context of the change.

The task can be achieved eg by the following change in method `Neighbours.newWisdom`*:*

```
        // update message forwarding info:
        nw.registerForward(myQueueName, neighbours.size());

        // MODEL ANSWER FOR 2.4(a) START
        Wisdom w = nw.getWisdom();
        w.setText("[mk] " + w.getText());
        // MODEL ANSWER FOR 2.4(a) END

        ObjectMessage msg = sendSession.createObjectMessage(nw);
```

Q6 acknowledge NewWisdom (15 pts)

Accurately describe changes you made to the given peer code during exercise **2.5.(a)**.

For each change indicate which file and method it is in, and cut and paste the altered lines together with three preceding and three following lines of original code to easily identify the context of the change.

The task can be achieved eg by the following change in method `Dispatcher.onMessage`*:*

```
        {
            NewWisdom nw = (NewWisdom) obj;

            // TASK 2.5(a) MODEL ANSWER START
            String senderName = nw.getMostRecentSenderName();

            // respond to the sender:
            try
            {
                neighbours.sendObjectTo(senderName, nw.constructResponse(myQueueName));
            }
            catch (JMSException e)
            {
                System.out.println("failed to respond to sender of NewWisdom: "
                        + senderName);
                e.printStackTrace();
            }
            // TASK 2.5(a) MODEL ANSWER END

            if (!wisdoms.contains(nw.getWisdom()))
            {
```

Q7 wait for NewWisdom ack (15 pts)

Accurately describe changes you made to the given peer code during exercise **2.5.(b)**.

For each change indicate which file and method it is in, and cut and paste the altered lines together with three preceding and three following lines of original code to easily identify the context of the change.

The task can be achieved eg by the following change in method `Neighbours.newWisdom`*:*

```
            neighbourSender.send(msg, DeliveryMode.NON_PERSISTENT,
                    Message.DEFAULT_PRIORITY, MESSAGE_TTL);

            // TASK 2.5.(b) MODEL ANSWER START
            if(dispatcher.awaitResponse(nw, 2 * SECOND) == null)
            {
                listener.removedNeighbour(neighbourName);
                System.out.println("removing unresponsive neighbour " + neighbourName);
                neighbours.remove(neighbourName);
            }
            // TASK 2.5.(b) MODEL ANSWER END
        }
    }
```

while also adding a dispatcher parameter in the method header:

```
    public synchronized void newWisdom(final NewWisdom nw,
            Dispatcher dispatcher) throws JMSException
```

and providing the value for this parameter in both calls, ie in `Dispatcher.forwardWisdom`*:*

```
    {
        try
        {
            neighbours.newWisdom(nw, this); // CHANGED IN TASK 2.5.(b)
        }
        catch (JMSException e)
        {
```

and in `Peer.forwardWisdom`*:*

```
    public void newWisdom(String text) throws JMSException
    {
        Wisdom wisdom = new Wisdom(text, queueName);
        wisdoms.addWisdom(wisdom);
        neighbours.newWisdom(new NewWisdom(queueName, wisdom), dispatcher);
         // CHANGED IN TASK 2.5.(b)
    }
```

Q8 why waiting for ack in parallel (15 pts)

Explain why it is beneficial to implement the change required by task 2.5.(d).

While waiting for an acknowledgement, which may take up to 2 seconds, it is good to proceed and send the wisdom to another neighbour and/or return control to the user interface. Waiting in a new thread enables such processing in parallel.

Without it, the propagation of wisdoms would be significantly slower and the user interfaces could appear to freeze at times for a few seconds.

Q9 parallel waiting for ack (0 pts)

(This is an optional question — it is does not contribute to the score. It will be assessed so that you can obtain feedback.)

Accurately describe changes you made to the given peer code during exercise **2.5.(d)**.

The task can be achieved eg by the following change in method `Neighbours.newWisdom`*:*

```
        neighbourSender.send(msg, DeliveryMode.NON_PERSISTENT,
                Message.DEFAULT_PRIORITY, MESSAGE_TTL);

        // TASK 2.5.(d) MODEL ANSWER START:
        final Neighbours thisNeighbours = this;

        new Thread(new Runnable()
        {
            public void run()
            {
                // TASK 2.5.(b) MODEL ANSWER START (also added the dispatcher method parameter)
                if(dispatcher.awaitResponse(nw, 2 * SECOND) == null)
                {
                    listener.removedNeighbour(neighbourName);
                    System.out.println("removing unresponsive neighbour " + neighbourName);
                    synchronized(thisNeighbours)
                    {
                        neighbours.remove(neighbourName);
                    }
                }
                // TASK 2.5.(b) MODEL ANSWER END
            }
        }).start();
        // TASK 2.5.(d) MODEL ANSWER END
    }
}
```

Notice that since the removal of a unresponsive neighbour now occurs in a separate thread, the **`synchronized`** *keyword in the method header does not apply to it and a new* **`synchronized`** *block has to be created around it. To be able to lock the correct object, a new local variable is introduced to make* **`this`** `Neighbours` *object available also for the new thread.*

Due to limitations of the Java language, it was also necessary to declare the new local variable as well as the dispatcher method parameter **`final`***. These variables are captured inside the new thread and thus it is undesirable that the variable change value. The keyword* **`final`** *guarantees that the variable will never change its value.*

Total of 100 pt.