Tentamen
EDA698 – Realtidssystem (Helsingborg)

2016–01–09, 08.00–13.00

Det är tillåtet att använda Java snabbreferens och miniräknare, samt ordbok.
Det går bra att använda både engelska och svenska uttryck för svaren.

Den här tentamen består av två delar: *teori*, frågor 1-5 (18 poäng), och *programmering & design*, frågor 6-7 (12 poäng). För godkänd (betyg 3) krävs sammanlagt ca hälften av alla 30 möjliga poäng samt att det krävs ca en tredjedel av poängen i varje del för sig. För högsta betyg (5) krävs dessutom att en del av poängen uppnås med (del)lösningar till båda uppgifterna i *programmering & design* (preliminärt).

---

1. **Concurrent programming**
   Within a large open-source project, error reports are exchanged via e-mail. The following quote was sent by one of the main programmers as a reply to an error report:

   *I spent a long time looking at this about a year ago and couldn't come up with anything definitive. It's especially difficult to debug since running it under the debugger works. It may be that **we can fix things by introducing some delays in the code.***

   a) What do we usually call this type of fault or problem? *(1p)*

   b) Mention two drawbacks if the problem is solved as proposed in the quote. *(1p)*

   c) In Java there is a keyword providing language support for solving this type of problem. What keyword is that? *(1p)*

2. **Resource allocation, deadlock analysis**
   A real-time system is implemented by three separate threads, T1, T2, and T3. These share five common resources, each protected by a binary semaphore in order to achieve mutual exclusion when using the corresponding resource. The binary semaphores are denoted A, B, C, D, and E. They are all unlocked when the three threads are started. The threads execute the code fragments below each time they run:

T1	T2	T3
A.take();	A.take();	E.take();
B.take();	B.take();	D.take();
C.take();	useAB();	useDE();
useABC();	B.give();	E.give();
B.give();	A.give();	A.take();
E.take();	C.take();	useAD();
useACE();	D.take();	A.give();
E.give();	useCD();	D.give();
C.give();	D.give();	
A.give();	C.give();	

   a) Draw a resource allocation graph for the system. *(2p)*

   b) The system may experience deadlock. Motivate why. *(1p)*

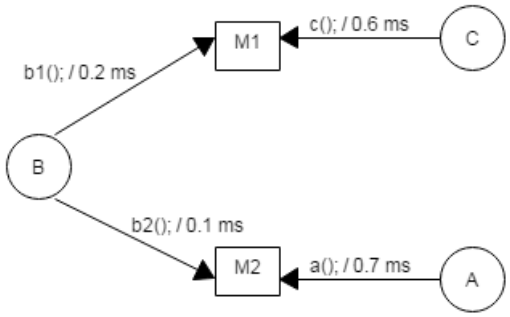
   c) Suggest a way to modify the code fragments above so that deadlock cannot occur no matter how many copies of the threads T1, T2 and T3 are running simultaneously. When a resource is used (call to a useXY(); method) the corresponding semaphore(s) must be locked. *(1p)*

---

3. **Response times, schedulability**
A real-time system is implemented using three periodic threads, A, B, and C. The threads are assigned priorities according to the principle rate monotonic scheduling (RMS). The worst-case execution times (C) and periods (T) for each thread are given below:

Thread	C (ms)	T (ms)
A	2	50
B	3	20
C	4	60

Thread B communicates with threads A and C via two monitors called M1 and M2 as shown in the figure. Each time A, B and C execute, they call the monitor methods indicated in the figure (once and only once per period). The maximum execution times for the two different monitor methods are shown in the figure.



Assume that the system is scheduled according to strict priority-based scheduling with dynamic priority inheritance (basic priority inheritance protocol).

a) What are the (maximum) blocking times for each thread? *(2p)*

b) What will the worst-case response times be for the three threads? *(2p)*

c) Is the system schedulable as suggested? *(1p)*

Hint:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

4. **Real-time system issues**
Debugging real-time systems can be problematic. Several new problems may arise that do not exist in single-threaded programs. Below, several real-time problems are stated. For each item, explain the problem and give an example.

a) Priority inversion *(1p)*

b) Race condition *(1p)*

c) Missed deadline *(1p)*

5. **Multitasking / preemption**
Concurrent activities executing on one processor must share the CPU time. Suppose we employ dynamic scheduling for reasons of flexibility. Explain using at most two sentences (or a short example) the meaning of the following concepts:

a) Cooperative multitasking (no preemption) *(1p)*

b) Multitasking using preemption with preemption points *(1p)*

c) Multitasking using preemption without preemption points (native preemption) *(1p)*

6. **Programming – Robot Arm Joint Control**

Your are asked to construct a part of the software required to control the motors and therefore the movement of a robot arm (each motor, or servo, has its own controller similar to the controller threads of the Washing Machine programming exercise in the course). It is important that the code is efficient and also that it is concurrency-correct as any errors from race conditions, data corruption, etc, could have serious consequences.

The system architecture is given. Each motor is to be controlled by its own servo controller thread (one thread per motor). One motion coordinator thread is responsible for the overall path of the robot, it communicates the motor specific part of the path to the respective servo controller thread. A supervisor thread periodically receives status messages from all servo controller threads to know that the thread is still alive and well and to learn about the current status of the respective servo. If a thread fails to send a report message in time or if there is an error condition reported for one of the motors, a controlled stop of the system is initiated. All communication between threads must occur using message-based communication, similar again to the Washing Machine exercise.

The system should behave as follows:

The system / each motor (servo) can be in three modes: RUNNING, JAMMED or STOPPED.

- RUNNING mode is the normal operating mode of the system. It means that all servos are functioning with the servo controllers receiving trajectory segments from the motion coordinator. The supervisor thread is periodically receiving messages from each servo controller reporting the joint's current state. If the supervisor does not receive status updates within a specified time period from each thread, it should propagate STOPPED to all threads but those that did not send their report.
- JAMMED mode is an error condition that occurs when a servo gets stuck. The servo controller thread of that particular servo should report JAMMED to the supervisor that in turn propagates JAMMED to all threads in the system, excluding the one that reported the issue. All servo controllers should then set the velocities of their joints to 0.0, but keep reading the joint's state and report that back to the supervisor. If the jammed servo gets unstuck (RUNNING), the respective controller thread should report that to the supervisor, which then can send RUNNING to all threads again. If a JAMMED servo keeps reporting JAMMED over a certain time, the supervisor should propagate STOPPED to all threads.
- STOPPED mode indicates that the system is halted. Servo controller threads should switch their motors off, but still read and report their status. The motion coordinator thread stops its work. There is at the moment no recovery from this condition.

There are three thread classes in the system: ServoController, MotionCoordinator and Supervisor.

- The ServoController thread exists in several running instances in the system, one for each motor. The thread class has a mailbox and defines a periodic thread (similar to the PeriodicThread for the Washing Machine exercise). It receives messages from the motion coordinator (PoseMsg) and from the supervisor (AliveReqMsg or StateMsg), to which it should react according to the specification of system states above.
- The MotionCoordinator thread class has a mailbox, is instantiated only once and has no specific period. It coordinates the different servos by sending pose requests to the respective threads and handling reports of success or error conditions.
- The Supervisor thread class has a mailbox, is instantiated only once and has no specific period. It supervises the servo threads and initiates global state changes, in case a servo error (jam) is reported or some servo controller thread is not reporting its state properly.

Your task is to implement the three thread classes' run() / perform() methods as outlined in the code skeleton (see page 5 and following). You may change / add attributes or alter the message classes in case you need to, but if you do, indicate that clearly. Indicate which relation to the period of periodic threads you find suitable for servo velocities or timeouts, in case that is not obvious in the code parts you write.

(9p)

7. **Design - Robot Arm Joint Control with Collision Detection / Obstacle Avoidance**
Your controller software works well within specifications and the core system architecture is decided to remain the same, however, some extensions are needed. You are asked to connect some touch sensors to the arm segments / joints and a camera system for general scene observation. If any of the sensors reports a light collision (basically, the arm segment bumped into something, e.g., a human user's arm that was moved too quickly into the work area of the robot to be detected as an obstacle), all joints need to be stopped immediately - but it must be possible to resume movement when the problem is solved. Here, it should be possible for the system to invoke some dialogue with a user to solve issues like a collision or a motor jam issue. Additionally, the camera system should be used to detect obstacles up-front (i.e., before the collision is there), so that it becomes possible to re-plan a trajectory (the overall movement of the robot arm). This means, that the motion coordinator might have to change a target pose for one or several joints before these have reported being done when the camera system evaluation reports a problem (i.e., an obstacle in the planned path of the robot arm). Suggest modifications to the system from task 6 in form of additional connections to sensors, cameras, modifications to messages, new message types, additional tool functionalities, new system states or changes in the handling of messages. Base your explanations on a diagram for the system.

(3p)

End of tasks, course specific Java reference and code skeleton follow

A Short, specific Java-reference with potentially appropriate methods

Tidshantering, trådhantering, väntelägen, etc

- klass Math och klass System:
  - public static double Math.random(): ger ett slump-värde mellan 0 och 1.
  - public static long System.currentTimeMillis(): aktuell systemtid i millisekunder
- klass Thread:
  - public static void sleep(long t): försätta en tråd i viloläge under t ms
  - public static void yield(): dra tråden tillbaka och sätt i kön för ny schemaläggning
  - public void join(): vänta att tråden dör
  - public void start(): starta upp tråden
  - public void run(): trådens arbete
- klass Object:
  - public final void wait(): försätta en tråd i vänteläge
  - public final void wait(long t): försätta en tråd i vänteläge under max t ms. Kan dock väckas ändå!
  - public final void notify(): meddela nästa tråden i vänte-kön att ett tillstånd har ändrats
  - public final void notifyAll(): meddela alla trådar i vänteläge att ett tillstånd har ändrats

B Code skeleton for task 6

The ServoController Thread - implement here

```
public class ServoController extends PeriodicRobotThread {
  private HW hw;               // the hardware access for setting velocities, etc.
  private int jointID;         // this instance's joint id
  private double currentPose;  // this servo's current pose
  private int jointState;      // this servo's current state

  /* Put your own attributes here */

  public ServoController( long period, HW hw, int id) {
    super( period);
    this.hw = hw;
    jointID = id;
    hw.setMotor( true);
    currentPose = hw.getPose( );
    jointState = hw.getState( );
  }

  /*
   * When the ServoController receives an AliveReqMsg
   * from the Supervisor, it should start sending status reports back periodically.
   * Upon receiving a new PoseMsg it starts controlling the joint servo's velocity
   * (e.g., so that it automatically slows down when coming close to the
   * requested pose, which results in keeping a pose around the desired
   * one if nothing new is coming in) and if requested in the message,
   * acknowledge with a status message when the requested pose is reached
   * (within some small margin).
   * If no new message is retrieved, it should keep doing what it was
   * told to do before (like the controllers in the Washing Machine exercise).
   * The servo controllers can set the velocity of their joint, set the motors
   * on or off and ask for joint status and current pose of the joint by using
   * the HW classes' respective methods.
   */
  public void perform() {
    /* YOUR TASK is to IMPLEMENT this method */
  }
}
```

The MotionCoordinator Thread - implement here

```
public class MotionCoordinator extends RobotThread {
  private RobotPath path;             // access to the robot path
  private int numOfJoints;            // number of joints / servos to handle
  private ServoController[] servos;   // the servo controllers

  /* Put your own attributes here */

  public MotionCoordinator( RobotPath p, int num, ServoController[] servos) {
    this.path = p;
    this.servos = servos;
    this.numOfJoints = num;
  }

  /*
   * As long as the MotionCoordinator can retrieve a new set of
   * joint poses (values) to work with (from RobotPath, see below the
   * methods), and no error is reported, it should send the joint
   * values with PoseMsg instances to the respective ServoController
   * instances. It should make sure that all servo controller threads
```

```
   * report back to have reached their requested pose before sending
   * out new requests (somewhat similar to a Washing Program
   * thread in the programming exercise). If it receives a StateMsg with
   * state STOPPED, it should empty its mailbox and stop (see code skeleton
   * of the thread class RobotThread for the possible mailbox operations).
   */
  public void run() {
    /* YOUR TASK is to IMPLEMENT this method! */
  }
}
```

The Supervisor Thread - implement here

```
public class Supervisor extends RobotThread {
  private ServoController[] servos;    // the servo controllers to supervise
  private MotionCoordinator moCo;      // the motion coordinator
  private int numOfJoints;             // number of joints and servos

  /* Put your own attributes here */

  public Supervisor( int num, ServoController[] servos, MotionCoordinator mc) {
    this.numOfJoints = num;
    this.moCo = mc;
    this.servos = servos;
  }

  /*
   * When started, the Supervisor should send one AliveReqMsg to each
   * ServoController instance, so that those know who to report to
   * regularly. After that it should start its supervision of the servo
   * controllers according to the state specification in the task description.
   * Observe, that it should not poll its mailbox unnecessarily often, but
   * still make sure that it reads messages as quickly as possible and also
   * make sure that it is not blocked from evaluating the time it can accept
   * for a servo controller not reporting anything (see code skeleton of the
   * thread class RobotThread for the possible mailbox operations).
   */
   public void run() {
    /* YOUR TASK is to IMPLEMENT this method! */
  }
}
```

The RobotThread classes' mailbox methods - ready to use

```
public class RobotThread extends Thread {
  /* constructor etc; */

  /* putting a message into the mailbox */
  public void putMsg( RobotMsg msg) { /*... */ }

  /* non-blocking fetch, returns null, if empty */
  protected RobotMsg tryFetch() { /* ... */ }

  /* fetch a message, blocks while buffer empty */
  protected RobotMsg doFetch() { /*... */ }

  /* fetch a message, blocks max timeout ms, returns null if still empty */
  protected RobotMsg doFetch( long timeout) { /* ... */ }

  /* empties the buffer */
  protected void flush() { /* ... */ }
}
```

The RobotMsg base class methods (superclass to AliveReqMsg, PoseMsg, StateMsg)

```
public class RobotMsg {
  /* constructor etc; */
  /* each instance (also of a subclass) has a UNIQUE id based on a counter */

  /* get the source (sender) of the message */
  public Object getSource(){ /* ... */ }

  /* get the time stamp of the message */
  public long getTimestamp() { /* ... */ }

  /* get the unique message id */
  public int getID() { /* ... */ }
}
```

The AliveReqMsg class - ready to use

```
public class AliveReqMsg extends RobotMsg {
  public AliveReqMsg( Object src, long timestamp) { /* ... */ }
  /* methods exactly as RobotMsg, but easier to identify this way */
}
```

The PoseMsg class - ready to use

```
public class PoseMsg extends RobotMsg {
 /*
  * offers a field for sending a requested pose (double value),
  * and a request for ack (true, if there should be an answer)
  */
  public PoseMsg( Object src, long timestamp, double pose, boolean ack) {/*...*/}

  /* get the pose specified with this message */
  public double getPose() { /* ... * / }

  /* get the status of ack request (true - the sender waits for an answer) */
  public boolean requestsAck() { /* ... */ }
}
```

The StateMsg class - ready to use

```
public class StateMsg extends RobotMsg {
  /*
   * offers fields for sending a jointID (e.g., -1 for global/general messages),
   * an ackid (in response to a message id requesting answer,
   * e.g., -1 for original messages)
   * and a state indicator (HW.RUNNING, HW.JAMMED, or HW.STOPPED)
   */
  public StateMsg( Object src, long timestamp, int id, int ackID, int state) {/*...*/}

  /* get the jointID for which this status is reported */
  public int getJointID() { /* ... */ }

  /* get the acid, i.e., the id of the message this is an answer to */
  public int getAckID() { /* ... */ }

  /* get the state that is reported */
  public int getState() { /* ... */ }
}
```

The HW class - ready to use

```
public class HW {
  public static final int RUNNING = 0;
  public static final int JAMMED = 1;
  public static final int STOPPED = 2;

  /* constructor and stuff */

  /* methods for access to the motor / joint parameters of each servo */

  /*
   * set the requested velocity for the joint motor,
   * return the current status of the joint
   * (RUNNING, if everything is fine,
   * JAMMED if there is a problem,
   * STOPPED, if the motors are off).
  */
  public synchronized int setVel( double vel) { /* ... */ }

  /* get the current pose value */
  public synchronized double getPose() { /*... */ }

  /* get the current state (without change of velocity) */
  public synchronized int getState( ) { /* ... */ }

  /*
   * set the motors according to onOff (true for on).
   * If true and not JAMMED, state is set to RUNNING, if
   * false, state is set to STOPPED in any case. Return current state of joint.
  */
  public synchronized int setMotor( boolean onOff) { /* ... */ }
}
```

The RobotPath class - ready to use

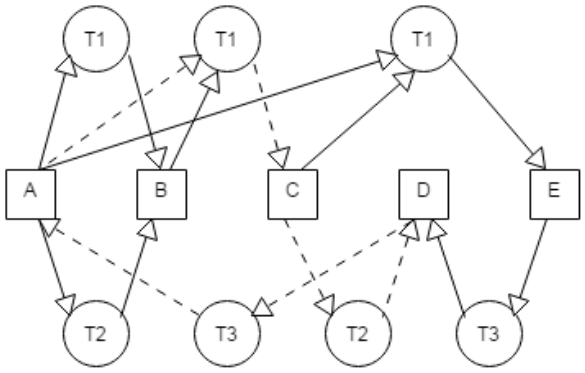
```
public class RobotPath {
  /* attributes, constructor and other methods*/

  /*
   * A tool method for generating joint values
   * generates the joint configuration
   * (a double value for each of the numOfJoints joints)
   * from an internally known desired path of the robot hand
   * You can assume that the servos are sorted in the
   * same order as their corresponding
   * values in this array!
   */
  public double[] generateJointValues( int numOfJoints) { /* ... */ }
}
```

Kortfattad lösning

1. a) Concurrency error or race condition.
   b) The error is still present even if it is not manifested. The fix becomes dependent on the underlying system.
   c) The keyword is synchronized. Also volatile as it might be used to create tread-safe solutions.

2. a) Resource allocation graph



   b) The system may deadlock. See the dashed cycle in the graph.
   c) For instance, reverse the hold-wait situation D->T3->A

```
A.take();
E.take();
D.take();
useDE();
E.give();
useAD();
A.give();
D.give();
```

3. RMS priority order:
   B
   A
   C

   RMS blocking:
   $B_B = 0.7 + 0.6 = 1.3$
   $B_A = 0.6$
   $B_C = 0$

   RMS response times:
   $R_B = 3 + 1.3 = 4.3$
   $R_A = 2 + 0.6 = 2.6$
   $R_A = 2 + 0.6 + \lceil \frac{2.6}{20} \rceil 3 = 5.6$
   $R_A = 2 + 0.6 + \lceil \frac{5.6}{20} \rceil 3 = 5.6$
   $R_C = 4$
   $R_C = 4 + \lceil \frac{4}{20} \rceil 3 + \lceil \frac{4}{50} \rceil 2 = 9$
   $R_C = 4 + \lceil \frac{11}{20} \rceil 3 + \lceil \frac{11}{50} \rceil 2 = 9$

   Yes, as all response times are well below periods / deadlines

4. a) Occurs when a high-priority thread is blocked by a lower-priority thread. For instance, consider three threads of high (H), medium (M) and low (L) priority. Lets say the L thread enters a monitor and blocks the H thread from entering. Now the M thread runs and stops the L thread from executing, blocking the H thread for an indeterminate period of time.

b) The scheduling of the threads determine the outcome of the code. For instance, consider a global variable a and let two threads write to this variable. The value of the variable then becomes determined by execution order.

c) In Java, each thread caches variables thread-locally. This means one thread might not see changes performed to the variable by another thread. Volatile means changes to a variable will therefore be visible to all threads.

5. a) In cooperative multitasking switching between threads occur on a voluntary basis.

b) Pre-emption points mean that thread switches occur at well-defined points, but it is up to the real-time kernel to decide if a switch occurs or not.

c) The real-time kernel decides on thread switches. As such the switches can occur at any time.

6. These are sketches of the relevant methods with no explicit guarantee for completeness. However, a very similar implementation (slightly more complex than requested) has been tested.

The ServoController class:

```
public class ServoController extends PeriodicRobotThread {
  private RobotThread aliveMsgReceiver;
  private RobotThread msgSource;
  private HardwareSim HW;

  private double currentPose, poseReq;
  private int jointState;
  private int mode;
  private int jointID;

  private RobotMsg msg;
  private boolean shouldAck, ackOK;
  private int ackid;


  public ServoController( long period, HardwareSim hw, int id) {
    super( period);
    msg = null;
    shouldAck = false;
    poseReq = 0.0;
    HW = hw;
    jointID = id;
    currentPose = HW.getPose( jointID);
    ackid = -1;
    aliveMsgReceiver = null;
    HW.setMotors( true, jointID);
    jointState = HW.getState( jointID);
    mode = STOP;
  }


  public void perform() {
    if( aliveMsgReceiver != null) {
      aliveMsgReceiver.putMsg(
          new StateMsg( this, System.currentTimeMillis(), jointID, -1, jointState));
    }

    msg = tryFetch();

    if( msg != null) {
      if( msg instanceof AliveReqMsg) {
        aliveMsgReceiver = (RobotThread)msg.getSource();
      } else if( msg instanceof PoseMsg) {
        poseReq = ((PoseMsg)msg).getPose();
```

```
          shouldAck = ((PoseMsg)msg).requestsAck();
          ackOK = false;
          if( shouldAck) {
            ackid = msg.getID();
            msgSource = (RobotThread)msg.getSource();
          }
          mode = MOVE;
        } else if( msg instanceof StateMsg) {
          ackOK = false;
          mode = (StateMsg)msg).getState();
        }
      }

      switch( mode) {
        case HardwareSim.JAMMED:
          jointState = HW.setVel( jointID, 0.0);
          break;
        case HardwareSim.STOPPED:
          jointState = HW.setVel( jointID, 0.0);
          HW.setMotors( false, jointID);
          break;
        case HardwareSim.RUNNING:
          currentPose = HW.getPose( jointID);
          if( Math.abs( poseReq - currentPose) < 0.02*Math.PI) {
            ackOK = true;
          }
          jointState = HW.setVel( jointID, (poseReq - currentPose) / (2.0* (double) period));

          if( jointState == HardwareSim.RUNNING && shouldAck && ackOK) {
            StateMsg ack = new StateMsg(
                this, System.currentTimeMillis(), jointID, ackid, jointState);
            msgSource.putMsg( ack);
            shouldAck = ackOK = false;
          } else if( jointState != HardwareSim.RUNNING) {
            mode = jointState;
          }
          break;
        default:
          break;
      }
    }
  }
```

The MotionCoordinator class

```
  public class MotionCoordinator extends RobotThread {
    private RobotPath path;
    private int numOfJoints;
    private ServoController[] servos;

    public MotionCoordinator( RobotPath p, int numOfJoints, ServoController[] servos) {
      this.path = p;
      this.servos = servos;
      this.numOfJoints = numOfJoints;
    }

    public void run() {
      double[] jointValues;
      int[] waitingForPoseAckID = new int[numOfJoints];
      int i;
      boolean error = false;
      int sumPoseAck = 0;
```

```
      RobotMsg msg = null;
      while( (jointValues = path.generateJointValues( numOfJoints)) != null && !error) {
        for( i=0; i<numOfJoints; i++) {
          PoseMsg m = new PoseMsg( this, System.currentTimeMillis(), jointValues[i], true);
          waitingForPoseAckID[i] = m.getID();
          servos[i].putMsg( m);
        }
        sumPoseAck = 0;
        while( !error && sumPoseAck != numOfJoints) {
          msg = doFetch();

          if( msg instanceof StateMsg) {
            if( msg.getSource() instanceof ServoController
                && ((StateMsg)msg).getState() == HardwareSim.RUNNING) {
              //making sure to get an ack for the right PoseMsg...
              for( i=0; i<numOfJoints; i++) {
                if( ((StateMsg)msg).getAckID() == waitingForPoseAckID[i]) {
                  sumPoseAck++;
                }
              }
            } else if( ((StateMsg)msg).getState() == HardwareSim.STOPPED){
              flush();
              error = true;
            }
          }
        }
      }
    }
```

The Supervisor class

```
  public class Supervisor extends RobotThread {
    private RobotMsg msg;
    private ServoController[] servos;
    private MotionCoordinator moCo;
    private int numOfJoints;

    public Supervisor( int numOfJoints, ServoController[] servos, MotionCoordinator moCo) {
      msg = null;
      this.numOfJoints = numOfJoints;
      this.moCo = moCo;
      this.servos = servos;
    }

    public void run() {
      boolean allStop = false, jammed = false;
      int state = 0;
      int jointID = -1;
      long lastTimeAlive[] = new long[numOfJoints];
      boolean alive[] = new boolean[numOfJoints];
      long jammedSince[] = new long[numOfJoints];

      for( int i=0; i<numOfJoints; i++) {
        servos[i].putMsg( new AliveReqMsg( this, System.currentTimeMillis()));
        lastTimeAlive[i] = System.currentTimeMillis();
      }

      while( !isInterrupted()) {
        msg = doFetch( 2000); // this gives immediate response to messages,
                                          // otherwise a period of approx. 2000ms

        if( msg != null && msg instanceof StateMsg) {
```

```
      state = ((StateMsg)msg).getState();
      jointID = ((StateMsg)msg).getJointID();

      if( state == HardwareSim.RUNNING && msg.getSource() == servos[jointID]) {
        lastTimeAlive[jointID] = msg.getTimestamp();
        alive[jointID] = true;
        jammedSince[jointID] = 0;

      } else if( state == HardwareSim.JAMMED && msg.getSource() == servos[jointID]) {
        if( jammedSince[jointID] == 0) jammedSince[jointID] = msg.getTimestamp();

        if( System.currentTimeMillis() - jammedSince[jointID] > 5000) {
          allStop = true;
        } else {
          if( !jammed) {
            for( int i=0; i<numOfJoints; i++) {
              if( i != jointID)
                servos[i].putMsg( new StateMsg(
                    this, System.currentTimeMillis(), i, -1, HardwareSim.JAMMED));
            }
            moCo.putMsg( new StateMsg(
                this, System.currentTimeMillis(), jointID, -1, HardwareSim.JAMMED));
            jammed = true;
          }
        }
      }
    }

    // checking whether a servo has not been sending
    long t = System.currentTimeMillis();
    for( int i=0; i<numOfJoints; i++) {
      if( t - lastTimeAlive[i] > 5000) {
        alive[i] = false;
        allStop = true;
      }
    }

    // checking whether a jammed joint has recovered or we need to stop
    if( jammed && !allStop) {
      jammed = false;
      for( int i=0; i<numOfJoints; i++) {
        if( jammedSince[i] > 0) jammed = true;
      }

      if( !jammed) {
        moCo.putMsg( new StateMsg(
            this, System.currentTimeMillis(), -1, -1, HardwareSim.RUNNING));
        for( int i=0; i<numOfJoints; i++) {
          servos[i].putMsg( new StateMsg(
              this, System.currentTimeMillis(), i, -1, HardwareSim.RUNNING));
        }
      }
    }  else if( allStop) {
      moCo.putMsg( new StateMsg(
          this, System.currentTimeMillis(), -1, -1, HardwareSim.STOPPED));
      for( int i=0; i<numOfJoints; i++) {
        if( alive[i]) servos[i].putMsg( new StateMsg(
            this, System.currentTimeMillis(), i, -1, HardwareSim.STOPPED));
      }
    }
  }
```

```
    }
}
```

7. Only a rough sketch! Your solution should of course contain explanations!