Agreement with Failure Detectors

CO 347: Distributed Algorithms

Department of Computing, Imperial College London

1 Objectives

The goal of this coursework is to design and implement in the Java programming language failure detectors and use them to reach agreement in an asynchronous (or, partially synchronous) message-passing system with crash failures. We have developed an emulator for one such system, which you will extend with your solutions.

Do not make the provided code or your solutions publicly available on the Internet (§3.4).

Overall, your submission will consist of two parts:

1. Your group will be assigned a Git repository on the departmental GitLab service that you will populate with your solutions. You only submit to CATE the SHA-1 hash of the final commit you wish to be marked in a file named `cw1-githash.txt`. *No custom created Git repositories are allowed.*

2. Second, you must submit a PDF report to CATE, named `cw1-report.pdf`, highlighting key design and implementation choices of your solution(s). *No hard copy is required.*

The next section describes your *four* implementation tasks in more detail. More information on your submission, along with its assessment strategy, is provided in §3. Appendices A–C describe the Java-based emulator you are going to extend with your implementations.

2 Implementation Tasks

Typically, failure detectors are implemented using *periodic heartbeats.* A correct process, say p, will suspect a faulty one, say q, if it hasn't received a heartbeat message from q within a given *timeout* period. This timeout is usually a function of message delay. Assuming messages are never lost and there is an—initially, unknown—upper bound on message delay, process q will be (eventually) suspected by all correct processes.

2.1 Perfect failure detectors

▷ Design and implement an instance of (a) a perfect failure detector $\mathcal{P}$, and (b) an eventually perfect failure detector $\diamond\mathcal{P}$.

Hints. The two implementations will differ in terms of the timeout period $T = \Delta + f(d_{q \to p})$, where Δ is the periodicity of the heartbeat messages and $f(d_{q \to p})$ is a function of the message delay, $d_{q \to p}$, from process q to process p. In (a), for example, you can assume *a priori* a uniform message delay **for all** processes. In (b), however, *such an assumption no longer holds due to* *the eventually strong accuracy property.* That is, $\diamond\mathcal{P}$ may falsely suspect some correct process *temporarily*, thus the timeout period **for each** process should be adjustable in real time.

Implementation details. Set the configuration parameters in `Utils.java` as shown in the table below.

Parameter	*Value for $\mathcal{P}$*	*Value for $\diamond\mathcal{P}$*
`Utils.accuracy`	`Accuracy.DEFAULT`	`Accuracy.DEFAULT`
`Utils.delay`	`Delay.UNIFORM`	`Delay.RANDOM`
`Utils.DELAY`	`100`	`100`

Appendices A.5 and A.6 explain how to simulate crash failures and slow links, respectively. In your implementation, you can measure delay in real-time as follows:

i) The sending process, appends at each (heartbeat) message `m` the current time:

```
m.setPayload(String.format("%d", System.currentTimeMillis()));
```

ii) The receiving process calculates message `m`'s delay as the difference:

```
delay = System.currentTimeMillis() - Long.parseLong(m.getPayload());
```

2.2 Consensus with Strong Failure Detectors

▷ Design and implement a *rotating coordinators* algorithm that solves consensus using (a) a strong failure detector $\mathcal{S}$ when the maximum number F of faulty processes is bounded by $F < N$, and (b) an eventually strong failure detector $\diamond\mathcal{S}$ when the number F is bounded by $F < \lceil \frac{N}{2} \rceil$ (that is, at least $\lceil {}^{(N+1)}/_{2} \rceil$ processes are correct).

Hints. Figure 1 shows the pseudocode for the two algorithms. You can use your implementation of $\mathcal{P}$ and $\diamond\mathcal{P}$ to emulate $\mathcal{S}$ and $\diamond\mathcal{S}$, respectively.

Implementation details. The pseudocode in Figure 1 follows the following conventions:

i) `read x` implies a command line argument. In your experiments, set `x` to be the id of each process. That is, process `P1` will propose value `x = 1`, `P2` will propose `x = 2`, and so on.

ii) `send [VAL: x, r] to all` means: "broadcast a message of type `VAL` and payload variables `x` and `r` to all processes."

iii) `collect [VAL: v, r] from G[p]` means: "block until either a message `[VAL: v, r]` is received from process `p`, or `p` is suspected." The function returns `true` upon successful reception or `false` if the process is suspected. Consider the implementation of this function *carefully*:

Due to the synchronisation model of the emulator, a process cannot block in `receive(m)` waiting for a specific type of message (say, a message of type `VAL`); such an action, for example, would prevent heartbeat messages from ever reaching that process. Nonetheless, function `collect m from p` is indeed a blocking call, notified either when `p` is suspected, or when message `m` is received from `p`. You can implement such a call using *monitors* or other synchronisation mechanisms.

```
1 read x // Assume x is pid
  for r:=1 to n do
     if (i = r) then
        send [VAL: x, r] to all
     if (collect [VAL: v, r] from G[r]) then
        x := v
7 decide x // Print message
```

```
8  read x // Assume x is pid
   estimate := x
   state := undecided
   r := 0
   k := 0
   while (state = undecided) do
      r := r + 1
      c := (r mod n) + 1
16    send [VAL: estimate, r, k] to G[c]
17    if (pid = c) then
         upon receipt of $\lceil {}^{(N+1)}/_2 \rceil$ [VAL: $x_j$, r, $k_j$] messages
            stash[r] := {[VAL: $x_j$, r, $k_j$]}
            t := max $k_j$, [VAL: −, −, $k_j$] $\in$ stash[r]
            estimate := any $x_j$, [VAL: $x_j$, −, t] $\in$ stash[r]
            send [OUTCOME: estimate, r] to all
23    if (collect [OUTCOME: v, r] from G[c]) then
         estimate := v
         k := r
         send [ACK: r] to G[c]
      else // Process c is suspected
         send [NACK: r] to G[c]
29    if (pid = c) then
         upon receipt of $\lceil {}^{(N+1)}/_2 \rceil$ [NACK: r] or [ACK: r] messages
            if ($|${[ACK: r]}$|$ = $\lceil {}^{(N+1)}/_2 \rceil$) then
32             send [DECISION: estimate, r] to all

33 upon receipt of [DECISION: value, r] message from process p
      if (state = undecided) then
         decide value // Print message
36       state := decided
```

Figure 1: Rotating coordinators algorithm using a strong failure detector (lines 1–7) and an eventually strong failure detector (lines 8–37).

iv) Each round of the consensus algorithm using $\diamond\mathcal{S}$ consists of four asynchronous phases:

1) Each process sends its current estimate value, stamped with the round number (`k`) in which that value was set (line 16).
2) The coordinator `c` gathers $\lceil {}^{(N+1)}/_2 \rceil$ estimate values, selects one with the highest `k` value, and proposes it to all processes (lines 17–22).
3) A process will either receive `c`'s proposed value, in which case it responds to `c` with an acknowledgement (`ACK`) message or it will suspect `c`, in which case it responds with a negative acknowledgement (`NACK`) (lines 23–28).
4) The coordinator gathers $\lceil {}^{(N+1)}/_2 \rceil$ `ACK`s or `NACK`s. If all replies are `ACK`s, then a majority of the processes has adopted `c`'s proposed value and so `c` broadcasts a decision request to all processes (lines 29–32). Whenever a process receives such a request, it decides accordingly (lines 33–36).

The rest of the pseudocode is self-explanatory. For these two tasks, configure `Utils.java` with the following parameters:

Parameter	*Value for $\mathcal{S}$*	*Value for $\diamond\mathcal{S}$*
`Utils.accuracy`	`Accuracy.WEAK`	`Accuracy.EVENTUALLY_WEAK`
`Utils.delay`	`Delay.UNIFORM`	`Delay.RANDOM`
`Utils.DELAY`	`100`	`100`

The configuration for $\mathcal{S}$ will simulate one of N processes that is never suspected, while the one for $\diamond\mathcal{S}$ will simulate $\lceil {}^{N}/_2 \rceil$ faulty processes.

3 Assessment

Task	%
§2.1 (a)	30
§2.1 (b)	20
§2.2 (a)	20
§2.2 (b)	30

Table 1: Allocation of marks.

The allocation of marks for this coursework is summarised in Table 1. Implementation tasks will be marked according to the correctness and style of the programs submitted, as well as their presentation—e.g., design choices, evaluation, examples—both in your report and during the interview process. Code that does not compile will be limited to a maximum of 60% of the allocated marks.

The submission deadline is February 16, at 23:59.

Feedback will be returned by February 23.

3.1 Git repository

You are provided with a Git repository, initialised with the emulator code, on GitLab. If you are not familiar with Git, the "Pro Git" book is an excellent resource, freely distributed at `http://git-scm.com/book`. If you are not familiar with GitLab, the Computing Support Group offers a short guide at `http://www.doc.ic.ac.uk/csg/guides/version-control/gitlab`. An example workflow can be the following:

1. Clone your repository to a local machine to work on the coursework:

   ```
   $ git clone git@gitlab.doc.ic.ac.uk:[your group repository]
   ```

2. Stage your files for commit:

   ```
   $ git add EventuallyPerfectFailureDetector.java
   ```

3. Commit your work to your local tree:

   ```
   $ git commit -m ‘My eventually perfect failure detector’
   ```

4. Push your work to the server:

   ```
   $ git push origin master
   ```

Ensure that your submission commit is a *descendent* of the initial commit you are provided with; avoid `git push --force` commands and related commands to that effect. Following this basic workflow, you should not run into any problems.

Once you have completed your coursework, record the SHA-1 hash of your final commit into a text file called `cw1-githash.txt`. For example, after your final commit to the GitLab server run:

```
$ git rev-parse HEAD >> cw1-githash.txt
```

You can also get the information directly from the GitLab front-end. Upload this file to CATE in the usual manner to record your solution for marking. You also need to submit your report via CATE. The report should be in PDF format and the filename should be `cw1-report.pdf`.

3.2 Code distribution

An implementation task should be associated with at least two Java class implementations, a failure detector and an accompanying process that instantiates it. Names must be:

– `PerfectFailureDetector.java` and `PFDProcess.java`;

– `EventuallyPerfectFailureDetector.java` and `EPFDProcess.java`;

– `StrongFailureDetector.java` and `SFDProcess.java`; and

– `EventuallyStrongFailureDetector.java` and `ESFDProcess.java`.

You may submit any additional `.java` files used by the above classes, based on your software design. *Do not commit* `.class` *files, or* `.out`/`.err` *files.* Your solution should compile and be ready to run simply by typing `make`.

All failure detectors must implement the `boolean isSuspect(Integer q)` method, which returns true if process `q` is in the suspect list. Print the following message when a process `q` is suspected (or unsuspected, in case of $\diamond\mathcal{P}$):

```
Utils.out(p.pid, String.format("P%d has been [un]suspected at %s",
   q.intValue(), Utils.timeMillisToDateString(System.currentTimeMillis())));
```

In addition, `SFDProcess` and `ESFDProcess` must print their final decided value `x`:

```
Utils.out(pid, String.format("Decided on %s", x));
```

For all your experiments, use the `networks/mesh-10.txt` topology file that represents a fully-connected network of $N = 10$ processes.

3.3 Report

Your report should be limited to 3 pages, with at least 1 inch margins on all sides. Do not list entire `.java` files in your report. Introduce only those code segments that are sufficient to argue about your various design and implementation choices, and their correctness:

– How are (eventually) perfect and (eventually) strong failure detectors associated in your class hierarchy?

– When and how is a suspect list updated in your detectors, and in what data structure?

– How is function `collect` implemented?

– How do rounds progress in each of the rotating coordinators algorithms, and what is their relation to timeout periods?

3.4 Non-disclosure agreement

The materials (i.e., software) being provided to you for completing this exercise are for your use *during this term only* as a means to assess your progress. Assessed exercises are not public material and should never be shared with anyone now or *at any time in the future.* In particular, please do not disclose to anyone your solution, nor the software provided to you as part of this exercise, and especially do not place your solution and the provided software in a publicly accessible repository.

4 Concluding remarks

Please report any questions, bugs, or problems you encounter with this assignment to:

`doc-staff-347@imperial.ac.uk`

Good luck!

A Emulating an asynchronous distributed system

An asynchronous (or, partial synchronous) distributed system is emulated as a set of processes connected by reliable communication channels to a software "switch", termed the `Registrar`. The Registrar provides the abstraction of a fully-connected system, i.e. every correct process can send messages to every other correct process. The Registrar can also simulate process failures and/or slow links.

The provided source code distribution emulates a distributed system of N processes, named `P1`, `P2`, . . . , and `PN`, with unique identifiers 1, 2, . . . , and N, respectively. The name `P0` is reserved for the Registrar.

A.1 The message abstraction

Processes exchange messages traversing the system as strings of the form:

```
source<|>destination<|>type<|>payload<|>
```

The `Message` class provides you with message constructors, as well as set (get) methods to modify (interrogate) its fields. You can further impose your own structure(s) in the `type` and/or `payload` fields of a message. Note, however, that the string separator `"<|>"` is reserved by the system.

All messages flow through the Registrar who has sufficient knowledge to relay messages based on just the destination process identifier. E.g., given message `m` and `m.getDestination() == 1`, message `m` will be delivered to process `P1`.

A.2 The process abstraction

Custom processes are implemented by extending the `Process` class, the provided process abstraction. E.g.,

```
37 class P extends Process {

     public P (String name, int id, int size) {
40      super(name, id, size);
     }

     public void begin() {}

     public synchronized void receive (Message m) {}

     public static void main(String [] args) {
48      P p = new P ("P1", 1, 2);
49      p.registeR();
50      p.begin();
     }
52 }
```

Following the order of the arguments in the super-class constructor (lines 40, 48), the code above creates a process named `P1` with identifier 1 in a system of $N = 2$ processes.

A Process offers three basic communication methods. Methods `unicast(m)` and `receive(m)` allow a process to send a message `m` to another process and receive a message `m` from another process, respectively. The third method is `broadcast(type, payload)`.

A.3 Communication channels

Methods `unicast` and `receive` are implemented using Java TCP sockets. Let us first consider `unicast`. When a process is instantiated, it opens a TCP connection (a channel) to the Registrar. This action is performed by the `super` constructor (line 40). All outgoing messages from a process to the Registrar are multiplexed over this channel.

Calls to `unicast(m)` are blocking. The function returns once message `m` has been successfully delivered to the Registrar. The duration of this blocking call is determined by two factors: the scheduling delay, imposed by the machine(s) on which the emulator runs, and the simulated message delay, imposed by the emulator itself (see §A.6).

Incoming messages are handled by a `Listener` thread (see `Listener.java`) associated with each process at creation time. The Listener accepts only one connection, from the Registrar, and will notify its process whenever a new message arrives. Thus, `receive(m)` is an asynchronous call. *A process should never block on* `receive(m)`.

A.4 Process registration and inter-process communication

When a new process is instantiated, and after it successfully connects to the Registrar, it must register itself against the Registrar's "switch board". This is achieved by a call to `registeR()` (line 49), after which inter-process communication is enabled.

The `registeR` call is associated with a synchronisation barrier, implemented at the Registrar. In essence, a registering process blocks until all N processes of the system have registered as well. This prevents a process from sending messages to others before the system is completely initialised. Use the `begin` method (line 50) to instantiate any objects that call one the three communication methods.

At the Registrar, there are two threads associated with each process p: a thread that handles incoming messages from p (`Worker.java`), and a thread that handles outgoing messages to p (`Worker$MessageHandler.class`). A snapshot of overall system architecture is illustrated in Figure 2.

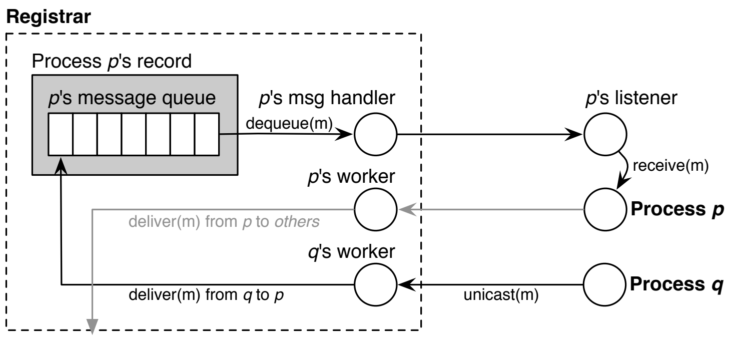

Figure 2: Inter-process communication at work. Process q's worker handles q's incoming messages—in this case, a message m to process p. This worker then enqueues m to p's message queue, an action that will notify p's message handler. In turn, the message will be delivered to p's listener. Upon receipt, the listener calls `p.receive(m)`. The message queue size is controlled by the `Utils.MSG_QUEUE_SIZE` variable.

A process p can send a message to another process q only if there is a link between the two processes. Links are encoded in topology files (under `networks/` directory), read by the Registrar when the system starts.

For a system of N processes, topology files encode process connectivity as an $N \times N$ matrix. If cell $[p][q]$ and $[q][p]$ is set to `1`, then processes p and q can communicate. Figures 3 and 4 illustrate two such sample topologies.

For the first coursework, use only the `mesh-10.txt` topology file.

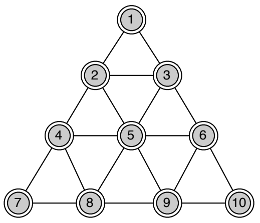

Figure 3: `tri-graph-10.txt`

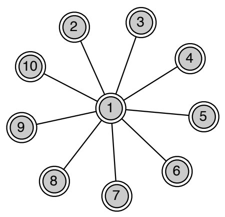

Figure 4: `star-10.txt`

A.5 Simulating crash failures

You can simulate crash failures (and crash-recovery failures) by turning ON/OFF registered processes via a user-interactive program, the `FaultInjector`:

```
$ java FaultInjector
> _
```

The FaultInjector connects to `FaultManager`, a component of the Registrar, and awaits your command(s). Commands are of the form:

$$\texttt{P}i\texttt{<|>ON} \text{ or } \texttt{P}i\texttt{<|>OFF}, \text{ for } 0 < i \leq N.$$

For example, after the following session all messages to and from process `P1` will be dropped:[1]

```
$ java FaultInjector
> P1<|>OFF
< OK
> _^C$
```

A.6 Simulating slow links

Consensus in an asynchronous system without failure detectors is impossible primarily because a process cannot determine whether another process has crashed or it is just "slow". This section covers how to simulate such slow links. Simulated message delay is controlled by two variables, located in `Utils.java`: `int DELAY`, and `enum Delay`.

A.6.1 Uniform delay

Variable `DELAY` represents the simulated message delay; in your distribution, `DELAY` $= 100ms$. The emulator delays the delivery of each message by this value. This allows us to make *a priori* assumptions on an upper bound of message delay that is the same **for all** processes.

However, due to the nature of the implementation, the observed message delay differs because it is also a function of the thread scheduling overhead. To demonstrate this effect, consider the following experiment. Ten instances of the `Broadcaster` class, running on a 2.4GHz Intel

[1]An equivalent, non-interactive command is `$ java FaultInjector -m "P1<|>OFF"`.

Core i7 with 8GB of RAM, broadcast 20,000 messages in total amongst each other. The average observed message delay is $\mu \simeq 102.2ms$, with a standard deviation of $\sigma \leq 5.7ms$. This difference of the observed and simulated delay is due to the overhead imposed by the operating system. The more threads run on it, the larger will be the difference. *Take this small overhead into account when you set timeouts.*

A.6.2 Random delay

When `Utils.delay` is `Delay.RANDOM`, the simulated message delay for a pair of processes is drawn from a Gaussian distribution with mean $\mu = \texttt{DELAY}$ and standard deviation $\sigma = {}^{\texttt{DELAY}}/_{2}$. Thus, when `Utils.delay` is random, a link can be either "slow" or "fast" and *a priori* assumptions on an upper bound of message delay no longer hold. Use this setting when evaluating eventually strong accuracy (e.g., in the case of an eventually perfect failure detector).

B Implementing failure detectors

Failure detector classes should implement the following basic interface. Of course, you may add additional methods.

```
59 interface IFailureDetector {

     /* Initiates communication tasks, e.g. sending heartbeats periodically */
     void begin ();

     /* Handles incoming (heartbeat) messages */
     void receive(Message m);

     /* Returns true if ‘process’ is suspected */
     boolean isSuspect(Integer process);

     /* Notifies a blocking thread that ‘process’ has been suspected.
      * Can be used for tasks in §2.2 */
     void isSuspected(Integer process);
73 }
```

Using this interface, a simple implementation of a failure detector and its accompanying process is shown below (lines 66–101 and 102–127, respectively).

```
74 class FailureDetector implements IFailureDetector {

     Process p;
     LinkedList<Integer> suspects;
     Timer t;

     static final int Delta = 1000; /* 1sec */

     class PeriodicTask extends TimerTask {
        public void run() {
           p.broadcast("heartbeat", "null");
        }
     }

     public FailureDetector(Process p) {
        this.p = p;
        t = new Timer();
        suspects = new LinkedList<Integer>();
     }

     public void begin () {
        t.schedule(new PeriodicTask(), 0, Delta);
     }

     public void receive(Message m) {
        Utils.out(p.pid, m.toString());
     }

     public boolean isSuspect(Integer pid) {
        return suspects.contains(pid);
     }

     public void isSuspected(Integer process) {
        return ;
     }
109 }
```

```
110 class P extends Process {

     private IFailureDetector detector;

     public P (String name, int pid, int n) {
        super(name, pid, n);
        detector = new FailureDetector(this);
     }

     public void begin () {
        detector.begin ();
     }

     public synchronized void receive (Message m) {
        String type = m.getType();
        if (type.equals("heartbeat")) {
           detector.receive(m);
        }
     }

     public static void main (String [] args) {
        P p = new P("P1", 1, 2);
        p.registeR ();
        p.begin();
     }
135 }
```

The provided implementation is simple because it only highlights two basic components of a failure detector:

a) a list of suspects;[2] and

[2]The list is implemented as a `LinkedList`. You can use another structure of your choice, e.g. an `ArrayList`.

b) a periodic task that sends a heartbeat message every one second.

Yet, the implementation does not highlight two important tasks: handling incoming heartbeat messages, and handling timeouts. These tasks are part of your assignment. Recall that each process is associated with a timeout period: upon receipt of a message from a process, the timeout period for that process should be updated; and when a time period expires, the process should be suspected.

C Running the emulator

This section covers how to run emulations using a UNIX shell (`bash`). The example used in this section emulates a system of $N = 2$ instances of process `P.class`, implemented as follows:

```
136 class P extends Process {

       public P(String name, int pid, int n) {
          super(name, pid, n);
       }

       public void begin() {}

       public static void main(String [] args) {
145       String name = args[0];
146       int id = Integer.parseInt(args[1]);
147       int  n = Integer.parseInt(args[2]);
          P p = new P(name, id, n);
149       p.registeR();
          p.begin();
       }
152 }
```

In general, such emulations can either start manually or automatically.

C.1 Starting processes manually

In this demonstration, three terminals are required: one for each process, and one for the Registrar. Since the first action of every process is to connect to the Registrar, the latter must always start first. The command is:

```
153 $ java Registrar 2 networks/pair.txt
    [000] Registrar started; n = 2.
155 _
```

The Registrar now awaits connections from $N = 2$ processes. In the second terminal, start the first process with name `P1` and identifier 1 by running the command:

```
156 $ java P P1 1 2
    [001] Connected.
158 _
```

Process `P1` now blocks, since its registration (line 149) will not complete until a second process registers as well (see §A.4). So, in the third terminal, run:

```
159 $ java MyProcess P2 2 2
    [002] Connected.
    [002] Registered.
162 _
```

Since `P2` is the second process that registers in a system of size $N = 2$, both `P1` and `P2` now complete their registration successfully. In fact, `P1` must have also printed the following message to its console:

```
163 [001] Registered.
164 _
```

No further output will be generated by the system.

C.2 Starting processes automatically

The `sysmanager.sh` shell script can manage processes for you, given that the first three command-line arguments of your implementation are (a) the process name, (b) the process identifier, and (c) the size of the system (lines 145–147). For example, the following command is equivalent to starting the Registrar (`P0`), `P1`, and `P2` manually, as demonstrated in the previous section:

```
165 $ ./sysmanager.sh start P 2 networks/pair.txt
    [DBG] P0's pid is 6214
    [DBG] start 2 instances of class P
    [DBG] P1's pid is 6216
    [DBG] P2's pid is 6217
    [000] Registrar started; n = 2.
    [001] Connected.
    [002] Connected.
    [002] Registered.
    [001] Registered.
    _
176 $ _
```

The `sysmanager` *demonises* the three processes, keeping track of the process identifiers assigned to these programs by the operating system. In order to stop them, type the command:

```
177 $ ./sysmanager.sh stop
    [DBG] stop
    [DBG] pid is 6214
    [DBG] pid is 6216
    [DBG] pid is 6217
    [DBG] clear
183 $ _
```

In line 165, the `sysmanager` is instructed to start 2 instances of `P.class` connected in a particular topology (`pairs.txt`). Subsequent command-line arguments to `sysmanager` are passed to process `P` as arguments `args[3]`, `args[4]`, and so on.

By default, `stderr` (i.e., Java's `System.err` print stream) is directed to log files, one per process: `P0.err`, `P1.err`, `P2.err`, and so on. After stopping the system, the `sysmanager` deletes empty error logs.

You can also direct `stdout` (i.e., Java's `System.out` print stream) to files—once again, one per process—by setting variable `LOG` to `true` (line 12 in `sysmanager.sh`).

The function `Util.out(pid, s)` prints string `s` to standard output, prefixed by the identifier `pid` of your emulated process. Print statements generated by the `sysmanager` itself are prefixed by `[DBG]`; you can turn them off by setting variable `VERBOSE` to `false` in `sysmanager.sh`.