Agreement in asynchronous distributed systems

DoC 437: 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* as a service for reaching various forms of agreement in an asynchronous (or, partially synchronous) message-passing system with crash failures. Overall, your submission will consist of two parts.

1. First, you must submit a hard-copy report, with a Cate coversheet, to the Student Administration Office. Throughout this document, instructions, comments, and questions will further guide you through the contents and structure of such a report.
2. Second, you must submit your accompanying software electronically, via Cate. You are required to extend the provided emulator (see §§A–C) with your solution(s). **The emulator was developed using Java 1.6.** You may submit your working code directory in its entirety, or only those source files you have added and/or modified.

The next section describes your coursework that is divided into two types of tasks, *implementation* (§2.1) and *analysis* (§2.2). More information on your submission, along with its assessment strategy, is provided in §3. Appendices A–C describe the Java-based framework you are going to use for your implementation tasks.

2 Failure Detectors

You are encouraged to begin your report by briefly introducing, in your own words, failure detectors and their classification according to the *correctness* and *accuracy* properties they satisfy. Failure detector properties will prove particularly useful in your argumentation for both the implementation and analysis tasks that follow.

2.1 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, e.g. twice the average message delay or the maximum 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 process p. This is an example of an "eventually perfect failure detector".

For all the implementation tasks that follow, assume $1 < N \leq 10$.

2.1.1 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 $T = \Delta + f(d)$, where Δ is the periodicity of the heartbeat messages and $f(d)$ is a function of message delay d. In (a), for example, you can assume *a priori* an average message delay $\bar{d} =$ `Utils.DELAY` for all processes (see §A.6.1).

In (b), however, such an assumption no longer holds due to the eventually strong accuracy property (see §A.6.2). That is, $\diamond\mathcal{P}$ may falsely suspect some correct process temporarily, thus the timeout period for every process should be adjustable in real time. 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()));
```

Implementation details. You should read §A and §B, if you haven't done so, as they explain how the emulator works and how to implement failure detectors. In particular, read carefully §A.5 and §A.6 as they explain how to simulate crash failures and slow links, respectively.

2.1.2 Eventual leader election

In eventual leader election, there is a time after which all correct processes elect (or, trust) the same correct process. Consider a linear ranking $(p, <)$ of n processes p, arranged in a mesh network, based on their identifiers:

$$p_1 < p_2 < \cdots < p_N.$$

Using the eventually perfect failure detector $\diamond\mathcal{P}$ of §2.1.1, derive an implementation of an eventual leader election algorithm assuming that the correct process with the highest ranking is the next leader.

Hints. The strong completeness of $\diamond\mathcal{P}$ ensures that every faulty process will be eventually suspected by all correct processes. In turn, this property ensures that a leader will always be some correct process. *Which one?* The answer is determined by $(p, <)$ and possible suspects $I(k) = \{p : p > p_k\}$.[1]

Implementation details. Calculate a (new) leader whenever the list of suspects is updated at the failure detector. You may simply extend your implementation of an eventually perfect failure detector.

[1]Given $(p, <)$, $I(N) = \emptyset$, $I(N-1) = \{p_N\}$, $I(N-2) = \{p_{N-1}, p_N\}$, and so on.

```
1 each process i executes:
    read x
    for r:=1 to n do
      if (i = r) then
        send [VAL: x, r] to all processes in G
      if (collect [VAL: v, r] from G[r]) then
        x := v
8   decide x
```

```
9 read x
  r := 0
  m := 0
  while (true) do
    r := r + 1
    c := (r mod n) + 1
    send [VAL: x, r] to G[c]
    if (i = c) then
      upon (receipt of N - F [VAL: $x_j$, r] messages)
        v := majority($\forall j$: $x_j$)
        d := ($\forall j$: $x_j$ = v) # This is a boolean
            variable
        send [OUTCOME: d, v, r] to $\forall j$
21  if (collect [OUTCOME: d, v, r] from G[c]) then
      x := v
      if (d) then
        decide x
        if (m = c) then
          exit
        else if (m = 0) then
28          m := c
```

Figure 1: Rotating coordinators algorithm using a strong failure detector (lines 1–8) and an eventually strong failure detector (lines 9–28).

2.1.3 Consensus

You are required to design and implement a *rotating coordinators* algorithm that solves consensus using (a) a strong failure detector $\mathcal{S}$ when the number F of faulty processes is bounded by $F < N$, and (b) an eventually strong failure detector $\diamond\mathcal{S}$ when the number F of faulty processes is bounded by $F < \frac{N}{3}$.

Hints. The two algorithms you are required to implement are available (as pseudocode) in "Part 6: Failure Detectors" of your Lecture notes, in slides 21 and 26, respectively. For convenience, the two algorithms are reproduced in Figure 1.

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

i) `read x` implies that there is an extra command line argument for each process, apart from the default ones (see §C).

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

iii) `collect [VAL: v, r] from G[r]` means: "wait (block) until either a message `[VAL: v, r]` is received from process `r`, or `r` is suspected; this function returns true upon successful reception. 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; such an action, for example, would prevent failure detectors' heartbeat messages from ever reaching that process. Nonetheless, function `collect m from r` is indeed a blocking call, notified either when `r` is suspected, or when a message of type `VAL` is received from `r`. You can implement such a call using *monitors*.

The rest of the pseudocode is self-explanatory.

2.2 Analysis tasks

Consider the completeness and accuracy properties of a perfect failure detector ($\mathcal{P}$), a strong failure detector ($\mathcal{S}$), an eventually perfect failure detector ($\diamond\mathcal{P}$), and an eventually strong failure detector ($\diamond\mathcal{S}$).

2.2.1 Relationships between classes

A failure detector $\mathcal{D}$ emulates another failure detector $\mathcal{D}'$ if its outputs (namely, suspected processes) are also outputs of $\mathcal{D}'$. For instance, $\mathcal{P}$ emulates $\mathcal{S}$, but $\mathcal{S}$ does not emulate $\mathcal{P}$. Further identify the relationships between $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$. Explain your rationale.

Hints. There are six pairs to consider in total. Relationships amongst pairs are transitive; that is, if a failure detector $\mathcal{D}$ emulates a failure detector $\mathcal{D}'$ and $\mathcal{D}'$ emulates $\mathcal{D}''$, then $\mathcal{D}$ also emulates $\mathcal{D}''$. Transitivity also implies that if D'' does not emulate D' and D emulates D', then D'' does not emulate D either.

Furthermore, you need to show that your derived relationships are uni-directional (or, antisymmetric); that is, either $\mathcal{D}$ emulates $\mathcal{D}'$, or $\mathcal{D}'$ emulates $\mathcal{D}$, or $\mathcal{D}$ and $\mathcal{D}'$ do not emulate each other. Otherwise, consider the case where $\mathcal{D}$ emulates $\mathcal{D}'$ and $\mathcal{D}'$ emulates $\mathcal{D}$, which wrongly concludes that $\mathcal{D}$ and $\mathcal{D}'$ are equivalent.

All of $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$ satisfy strong completeness, but each of them satisfies a different accuracy property from which a natural hierarchy unfolds.

2.2.2 Weak completeness

The aforementioned classes of failure detectors ($\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$) satisfy strong completeness, i.e. every faulty process will be eventually suspected by *every* correct process. Consider the property of *weak completeness*, i.e. every faulty process will be eventually suspected by *some* correct process.

Describe in your report a scheme with which a failure detector that satisfies weak completeness can be augmented to emulate strong completeness. Justify your answer.

Hints. For example, revisit your implementation of $\mathcal{P}$: what is the necessary and sufficient payload that heartbeat messages must carry, and why?

3 Assessment

The allocation of marks for this coursework is summarised in Table 1. Implementation tasks (§2.1) will be marked according to the correctness and style of the programs submitted, as well as their presentation – e.g., design choices, evaluation, examples – in your report alongside the analysis tasks (§2.2).[2]

The submission deadline is March 14, at 16:00. Feedback will be returned by March 28.

[2]Code that does not compile will be limited to a maximum of 60% of the allocated marks.

Task	%
§2.1.1 (a)	10
§2.1.1 (b)	15
§2.1.2	5
§2.1.3 (a)	20
§2.1.3 (b)	20
§2.2.1	15
§2.2.2	15

Table 1: Allocation of marks.

3.1 Implementation tasks

An implementation task should be associated with at least two Java class implementations: a failure detector and an accompanying process that instantiates it.[3] Consider the following submission guidelines for your implementation tasks:

i) Do not submit `.class` files. Clearly state in a `README` file, submitted as part of your code distribution, and (or) in your report what source files are associated with what task along with instructions on how to compile and execute them.

ii) Do not list entire `.java` files in your report. Introduce only those code fragments that are necessary and sufficient to argue about your various design and implementation choices.

iii) Do not list large `.out` log files, nor include them in your code distribution. Design and implement your evaluation methodology carefully, and report on your experimental setup and those necessary and sufficient generated logs that demonstrate the correctness of your implementation.

iv) For all your experiments, $1 < N \leq 10$.

v) Pack your code distribution in `source-code.tar`; and submit it via CATE.

3.2 Analysis tasks

The two analysis tasks are short, yet they require critical thinking that will further enhance your knowledge on failure detectors. As such, each answer should be limited to a page.

[3]In fact, your code distribution should be limited to the following files:

- `PerfectFailureDetector.java` and `PFDProcess.java`
- `EventuallyPerfectFailureDetector.java` and `EPFDProcess.java`
- `EventuallyLeaderElector.java` and `ELEProcess.java`
- `StrongFailureDetector.java` and `SFDProcess.java`
- `EventuallyStrongFailureDetector.java` and `ESFDProcess.java`
- `Makefile` (or equivalent)
- `README`
- Any additional `.java` file that is used by the above classes.
- Any `.java` file from the original distribution that has been modified.

4 Concluding remarks

Please report any questions, bugs, or problems you encounter with this assignment to both Alex Wolf and Alexandros Koliousis (`a.wolf@imperial.ac.uk` and `a.koliousis@imperial.ac.uk`, respectively).

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 and its components who have 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.,

```
29 class P extends Process {

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

     public void begin() {}

     public synchronized void receive (Message m) {}

     public static void main(String [] args) {
40      P p = new P ("P1", 1, 2);
41      p.registeR();
42      p.begin();
     }
44 }
```

Following the order of the arguments in the super-class constructor (lines 32, 40), 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 32). 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 (cf. §A.6).

Incoming messages are handled by a `Listener` thread (cf. `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 41), 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 42) 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.

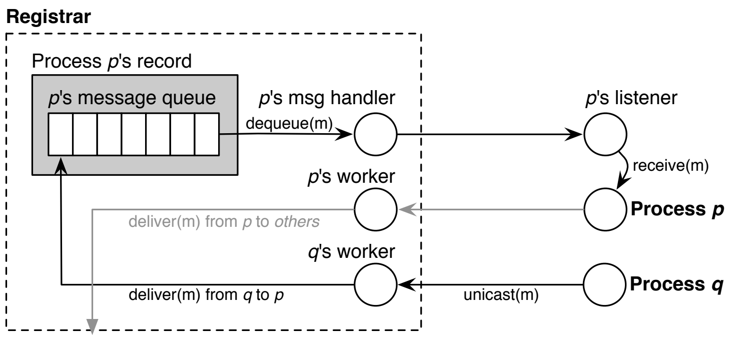

Figure 2: Inter-process communication at works. Process q's worker handles q's incoming messages – in this case, a message m to process p. This worker then enqueues m to p's message queue, an action that will notify p's message handler. In turn, the message will be delivered to p's listener. Upon receipt,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

A.5 Simulating crash failures

You can simulate crash failures (and crash-recory 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:

`P`i`<|>ON` or `P`i`<|>OFF`, for $0 < i \leq N$.

For example, after the following session all messages to and from process `P1` will be dropped by the Registrar:

```
$ 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
- `boolean GAUSSIAN`.

A.6.1 Fixed delay

Variable `DELAY` represents the simulated message delay; in your distribution, `DELAY` $= 100ms$. The emulator delays the delivery of each message by this value. 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, consider the following experiment.

Ten instances of the `Broadcaster` class, running on a 2.4GHz Intel Core i7 with 8GB of RAM, broadcast 20,000 messages in total amongst each other. Each process receives approximately 88 messages/s; 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.

In the above experiment, variable `GAUSSIAN` was set to `false`. This allows us to make *a priori* assumptions on an upper bound of message delay; such an assumption can be used when evaluating a perfect failure detector.

A.6.2 Gaussian delay

When `GAUSSIAN` is `true`, however, the simulated message delay is a Gaussian distribution with mean $\mu = \texttt{DELAY}$ and standard deviation $\sigma = \frac{\texttt{DELAY}}{2}$. To demonstrate this, the above experiment was repeated with `GAUSSIAN = true`. Results indicate an average delivery rate of approximately

87.5messages/s; the average delay is $\mu \simeq 102.2ms$, as in the previous case; the standard deviation, however, is $\sigma \simeq 48.6ms$.

Thus, when `GAUSSIAN` is `true`, a link can be either “slow” or “fast” and *a priori* assumptions on an upper bound of message delay no longer hold. When evaluating eventually strong accuracy (e.g., in the case of an eventually perfect failure detector), assume that `GAUSSIAN` is `true`.

B Implementing failure detectors

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

```
51 interface IFailureDetector {

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

     /* Handles in-coming (heartbeat) messages */
     void receive(Message m);

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

     /* Returns the next leader of the system; used only for §2.1.2 */
     int getLeader();

     /* Notifies a blocking thread that 'process' has been suspected.
      * Used only for tasks in §2.1.3 */
     void isSuspected(Integer process);
68 }
```

Using this interface, a simplistic implementation of a failure detector and its accompanying process is shown below (lines 69–108 and 109–134, respectively).

```
69 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 int getLeader() {
        return -1;
     }

     public void isSuspected(Integer process) {
        return ;
     }
108 }
```

```
109 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();
      }
134 }
```

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

a) a list of suspects;[4] and

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 – they are left as part of your implementation tasks. 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.

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

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:

```
135 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) {
144       String name = args[0];
145       int id = Integer.parseInt(args[1]);
146       int  n = Integer.parseInt(args[2]);
          P p = new P(name, id, n);
148       p.registeR();
          p.begin();
       }
151 }
```

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:

```
$ java Registrar 2
[000] Registrar started; n = 2.
_
```

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:

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

Process `P1` now blocks, since its registration (line 148) will not complete until a second process registers as well (cf. §A.4). So, in the third terminal, run:

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

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:

```
[001] Registered.
_
```

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 144–146). For example, the following command is equivalent to starting the Registrar (`P0`), `P1`, and `P2` manually, as demonstrated in the previous section:[5]

```
164 $ ./sysmanager.sh start P 2
    [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.
    _
175 $ _
```

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:

```
176 $ ./sysmanager.sh stop
    [DBG] stop
    [DBG] pid is 6214
    [DBG] pid is 6216
    [DBG] pid is 6217
    [DBG] clear
182 $ _
```

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` (line 8 in `sysmanager.sh`).

[5]In line 164, the `sysmanager` is instructed to start 2 instances of `P.class`. Subsequent command-line arguments to `sysmanager` are passed to the Java class as arguments `args[3]`, `args[4]`, and so on.