Agreement with Failure Detectors

Alexandros Koliousis

Department of Computing, Imperial College London

1 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.

1.1 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`.

1.2 The process abstraction

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

```
class P extends Process {

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

   public void begin() {}

   public synchronized void receive (Message m) {}

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

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)`.

1.3 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 §1.6).

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)`.

1.4 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 1.

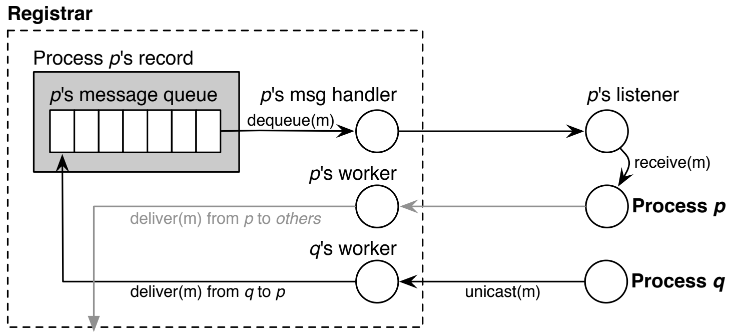

Figure 1: Inter-process communication at work. Process q's worker handles q's incoming messages—in this case, a message m to process p. This worker then enqueues m to p's message queue, an action that will notify p's message handler. In turn, the message will be delivered to p's listener. Upon receipt, 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 2 and 3 illustrate two such sample topologies.

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

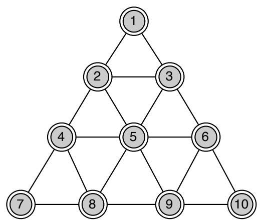

Figure 2: `tri-graph-10.txt`

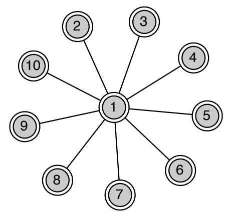

Figure 3: `star-10.txt`

1.5 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]

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

```
$ java FaultInjector
> P1<|>OFF
< OK
> _C$
```

1.6 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`.

1.6.1 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 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.*

1.6.2 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 $\mu =$ `DELAY` and standard deviation $\sigma = {}^{\mathtt{DELAY}}/_{2}$. 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).

2 Implementing failure detectors

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

```
23 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 §?? */
   void isSuspected(Integer process);
37 }
```

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

```
38 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 ;
   }
73 }
```

```
74 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();
   }
99 }
```

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

a) a list of suspects;[2] 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. 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

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

timeout period for that process should be updated; and when a time period expires, the process should be suspected.

3 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:

```
100 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) {
109      String name = args[0];
110      int id = Integer.parseInt(args[1]);
111      int  n = Integer.parseInt(args[2]);
         P p = new P(name, id, n);
113      p.registeR();
         p.begin();
      }
116 }
```

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

3.1 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:

```
117 $ java Registrar 2 networks/pair.txt
    [000] Registrar started; n = 2.
119 _
```

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:

```
120 $ java P P1 1 2
    [001] Connected.
122 _
```

Process `P1` now blocks, since its registration (line 149) will not complete until a second process registers as well (see §1.4). So, in the third terminal, run:

```
123 $ java MyProcess P2 2 2
    [002] Connected.
    [002] Registered.
126 _
```

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:

```
127 [001] Registered.
128 _
```

No further output will be generated by the system.

3.2 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:

```
129 $ ./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.
    _
140 $ _
```

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:

```
141 $ ./sysmanager.sh stop
    [DBG] stop
    [DBG] pid is 6214
    [DBG] pid is 6216
    [DBG] pid is 6217
    [DBG] clear
147 $ _
```

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`.