Design Tool To Express Failure Detection Protocols

Vincenzo De Florio and Chris Blondia
University of Antwerp, Department of Mathematics and Computer Science
Performance Analysis of Telecommunication Systems group
Middelheimlaan 1, 2020 Antwerp, Belgium
Interdisciplinary institute for BroadBand Technology
Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium

June 22, 2021

Abstract

Failure detection protocols—a fundamental building block for crafting fault-tolerant distributed systems—are in many cases described by their authors making use of informal pseudo-codes of their conception. Often these pseudo-codes use syntactical constructs that are not available in COTS programming languages such as C or C++. This translates into informal descriptions that call for ad hoc interpretations and implementations. Being informal, these descriptions cannot be tested by their authors, which may translate into insufficiently detailed or even faulty specifications. This paper tackles this problem introducing a formal syntax for those constructs and a C library that implements them—a tool-set to express and reason about failure detection protocols. The resulting specifications are longer but non ambiguous, and eligible for becoming a standard form.

1 Introduction

Failure detection constitutes a fundamental building block for crafting fault-tolerant distributed systems, and many researchers have devoted their efforts on this direction during the last decade. Failure detection protocols are often described by their authors making use of informal pseudo-codes of their conception. Often these pseudo-codes use syntactical constructs such as repeat periodically [1, 2, 3], at time t send heartbeat [4, 3], at time t check whether message has arrived [4], or upon receive [2], together with several variants (see Table 1). We observe that such syntactical constructs are not often found in COTS programming languages such as C or C++, which brings to the problem of translating the protocol

Construct	NFD-E [4]	φ [5]	FD [3]	GMFD [6]	$\mathcal{D} \in \Diamond\mathcal{P}$ [1]	$\mathcal{HB}$ [2]	$\mathcal{HB}$-pt [2]
Repeat periodically	no	no	yes	no	yes	yes	yes
Upon $t =$ current time	yes	no	yes	yes	no	no	no
Upon receive message	yes	yes	yes	yes	yes	yes	yes
Concurrency management	yes	yes	no	no	yes	yes	yes

Table 1: Syntactical constructs used in several failure detector protocols. φ is the accrual failure detector [5]. $\mathcal{D}$ is the eventually perfect failure detector of [1]. $\mathcal{HB}$ is the Heartbeat detector [2]. $\mathcal{HB}$-pt is the partition-tolerant version of the Heartbeat detector. By "Concurrency management" we mean coroutines, threading or forking.

specifications into running software prototypes using one such standard language. Furthermore the lack of a formal, well-defined, and standard form to express failure detection protocols often leads their authors to insufficiently detailed descriptions. Those informal descriptions in turn complicate the reading process and exacerbate the work of the implementers, which becomes time-consuming, error-prone and at times frustrating.

Several researchers and practitioners are currently arguing that failure detection should be made available as a network service [7, 8]. To the best of our knowledge no such service exists to date. Lacking such tool, it is important to devise methods to express in the application layer of our software even the most complex failure detection protocols in a simple and natural way.

In the following we introduce one such method—a class of "time-outs", i.e., objects that postpone a certain function call by a given amount of time. This feature converts time-based events into non time-based events such as message arrivals and easily expresses the constructs in Table 1 in standard C. In some cases, our class removes the need of concurrency management requirements such as coroutines or thread management libraries. The formal character of our method allows rapid-prototyping of the algorithms with minimal effort. This is proved through a Literate Programming [9] framework that produces from a same source file both the description meant for dissemination and a software skeleton to be compiled in standard C or C++.

The rest of this article is structured as follows: Section 2 introduces our tool. In Sect. 3 we use it to express three classical failure detectors. Section 4 is a case

1. /* declarations */
TOM *tom;
timeout_t t1, t2, t3;
int my_alarm(TOM*), another_alarm(TOM*);

2. /* definitions */
tom ← tom_init(my_alarm);
tom_declare(&t1, TOM_CYCLIC, TOM_SET_ENABLE, TIMEOUT1, SUBID1, DEADLINE1);
tom_declare(&t2, TOM_NON_CYCLIC, TOM_SET_ENABLE, TIMEOUT2, SUBID2, DEADLINE2);
tom_declare(&t3, TOM_CYCLIC, TOM_SET_DISABLE, TIMEOUT3, SUBID3, DEADLINE3);
tom_set_action(&t3, another_alarm);

3. /* insertion */
tom_insert(tom, &t1), tom_insert(tom, &t2), tom_insert(tom, &t3);

4. /* control */
tom_enable(tom, &t3);
tom_set_deadline(&t2, NEW_DEADLINE2);
tom_renew(tom, &t2);
tom_delete(tom, &t1);

5. /* deactivation */
tom_close(tom);

Table 2: Usage of the TOM class. In **1.** a time-out list pointer and three time-out objects are declared, together with two alarm functions. In **2.** the time-out list and the time-outs are initialized, and a new alarm is associated to time-out `t3`. Insertion is carried out at point **3.** At **4.** `t3` is enabled and a new deadline value is specified for `t2`. The latter is renewed and `t1` is deleted. The list is finally deactivated in **5.**

study describing a software system built with our tool. Our conclusions are drawn in Sect. 5.

2 Time-out Management System

This section briefly describes the architecture of our time-out management system (TOM). The TOM class appears to the user as a couple of new types and a library of functions. Table 2 provides an idea of the client-side protocol of our tool.

To declare a time-out manager, the user needs to define a pointer to a `TOM` object and then call function `tom_init`. Argument to this function is an alarm, i.e., the function to be called when a time-out expires:

```
int alarm(TOM *); tom = tom_init( alarm );
```

The first time function `tom_init` is called a custom thread is spawned. That thread is the actual time-out manager.

Now it is possible to define time-outs. This is done via type `timeout_t` and function `tom_declare`; an example follows:

```
timeout_t t; tom_declare(&t,TOM_CYCLIC, TOM_SET_ENABLE,
                TID, TSUBID, DEADLINE).
```

In the above, time-out `t` is declared as:

- A cyclic time-out (renewed on expiration; as opposed to `TOM_NON_CYCLIC`, which means "removed on expiration"),
- enabled (only enabled time-outs "fire", i.e., call their alarm on expiration; an alarm is disabled with `TOM_SET_DISABLE`),
- with a deadline of `DEADLINE` local clock ticks before expiration.

A time-out `t` is identified as a couple of integers—`TID` and `TSUBID` in the above example. This is done because in our experience it is often useful to distinguish instances of *classes* of time-outs. We use then `TID` for the class identifier and `TSUBID` for the particular instance. A practical example of this is given in Sect. 4.

Once defined, a time-out can be submitted to the time-out manager for insertion in its running list of time-outs—see [10] for further details on this. From the user point of view, this is managed by calling function

```
tom_insert( TOM *, timeout_t * ).
```

Note that a time-out might be submitted to more than one time-out manager.

After successful insertion an enabled time-out will trigger the call of the default alarm function after the specified deadline. If that time-out is declared as `TOM_CYCLIC` the time-out would then be re-inserted.

Other control functions are available: a time-out can be temporarily suspended while in the time-out list via function

```
tom_disable( TOM *, timeout_t * )
```

and (re-)enabled via function

```
tom_enable( TOM *, timeout_t * ).
```

Furthermore, the user can specify a new alarm function via `tom_set_action`) and a new deadline via `tom_set_deadline`; can delete a time-out from the list via

```
tom_delete( TOM *, timeout_t * ),
```

and renew[1] it via

```
tom_renew( TOM *, timeout_t * ).
```

Finally, when the time-out management service is no longer needed, the user should call function

```
tom_close( TOM * ),
```

which also halts the time-out manager thread should no other client be still active.

2.1 System assumptions, building blocks, and algorithms

This section is to provide the reader with a clear definition of

- the system assumptions our tool builds upon,
- the architectural building blocks of our system,
- the algorithms managing the list of time-outs.

2.1.1 System assumptions

Our tool is built in C for a generic Unix-like system with threads and standard inter-process communication facilities. Two implementation exists to date—one based on Embedded Parix [11], the other using the standard Posix threads library [12]. A fundamental requirement of our model is that processes must have access to some local physical clock giving them the ability to measure time. The availability of means to control the priorities of processes is also an important factor to reducing the chances of late alarm execution. We also assume that the alarm functions are small grained both in CPU and I/O usage so as not to interfere "too much" with the tasks of the TOM. Finally, we assume the availability of asynchronous, non-blocking primitives to send and receive messages.

2.1.2 Architectural building blocks

Figure 1 portrays the architecture of our time-outs manager: in

(1), the client process sends requests to the time-out list manager; in

[1] Renewing a time-out means removing and re-inserting it.

(2), the time-out list manager accordingly updates the time-out list with the server-side protocol described in Sect. 2.1.3.

(3) Each time a time-out reaches its deadline, a request for execution of the corresponding alarm is sent to a task called alarm scheduler.

(4) The alarm scheduler allocates an alarm request to the first available process out of those in a circular list of alarm processes, possibly waiting until one of them becomes available.

Figure 2 shows the sequence diagram corresponding to the initialization of the system and the management of the first time-out request.

The presence of an alarm scheduler and of the circular list of alarm processes can have great consequences on performance and on the ability of our system to fulfil real-time requirements. Such aspects have been studied in [10]. Our system may also operate in a simpler mode, without the above mentioned two components and with the time-out list manager taking care of the execution of the alarms.

2.1.3 Algorithms

The server-side protocol is run by a component called time-out list manager (TLM). The TLM implements a well-known time-out queuing strategy that is described e.g. in [13]. TLM basically checks every TM_CYCLE for the occurrence of one of these two events:

- A request from a client has arrived. If so, TLM serves that request.
- One or more time-outs have expired. If so, TLM executes the corresponding alarms.

Each time-out t is characterized by its *deadline* t.deadline, a positive integer representing the number of clock units that must separate the time of insertion or renewal from the scheduled time of alarm execution. This field can only be set by functions tom_declare and tom_set_deadline. Each time-out t holds also a field, t.running, initially set to t.deadline.

Each time-out list object, say tom, hosts a variable representing the origin of the time axis. This variable, tom.start_time, regards in particular the time-out at the top of the time-out list—the idea is that the top of the list is the only entry whose running field needs to be compared with current time in order to verify the occurrence of the time-out-expired event. For the time-outs behind the top one, that field represents relative values, viz., distances from expiration time of the closest, preceding time-out. In other words, the overall time-out list management aims at

isolating a "closest to expiration" time-out, or head time-out, that is the one and only time-out to be tracked for expiration, and at keeping track of a list of "relative time-outs."

Let us call TimeNow the system function returning the current value of the clock register. In an ordered, coherent time-out list, residual time *for the head time-out* t is given by

$$\mathsf{t.running} - (\mathsf{TimeNow} - \mathsf{tom.start\_time}), \tag{1}$$

that is, residual time minus time already passed by. Let us call quantity (1) as r_1, or head residual. For time-out n, $n > 1$, that is for the time-out located $n - 1$ entries "after" the top block, let us define

$$r_n = r_1 + \sum_{i=2}^{n} \mathsf{t}_i\mathsf{.running} \tag{2}$$

as the n-th residual, or residual time for time-out at entry n. If there are m entries in the time-out list, let us define $r_j = 0$ for any $j > m$.

It is now possible to formally define the key operations on a time-out list: insertion and deletion of an entry.

Insertion Three cases are possible, namely insertion on top, in the middle, and at the end of the list.

Insertion on top. In this case we need to insert a new time-out object, say t, such that $t.\mathsf{deadline} < r_1$, or whose deadline is less than the head residual. Let us call u the current top of the list. Then the following operations need to be carried out:

$$\begin{cases} t.\mathsf{running} & \leftarrow & t.\mathsf{deadline} + \mathsf{TimeNow} - \mathsf{tom.start\_time} \\ u.\mathsf{running} & \leftarrow & r_1 - t.\mathsf{deadline}. \end{cases}$$

Note that the first operation is needed in order to verify relation

$$t.\mathsf{running} - (\mathsf{TimeNow} - \mathsf{tom.start\_time}) = t.\mathsf{deadline},$$

while the second operation aims at turning the absolute value kept in the running field of the "old" head of the list into a value relative to the one stored in the corresponding field of the "new" top of the list.

Insertion in the middle. In this case we need to insert a time-out t such that

$$\exists\, j : r_j \leq t.\mathsf{deadline} < r_{j+1}.$$

Let us call u time-out $j+1$. (Note that both t and u exist by hypothesis). Then the following operations need to be carried out:

$$\begin{cases} t.\mathsf{running} & \leftarrow t.\mathsf{deadline} - r_j \\ u.\mathsf{running} & \leftarrow u.\mathsf{running} - t.\mathsf{running}. \end{cases}$$

Observation 1 *Note how, both in the case of insertion on top and in that of insertion in the middle of the list, time interval* $[0, r_m]$ *has not changed its length—only, it has been further subdivided, and is now to be referred to as* $[0, r_{m+1}]$.

Insertion at the end. Let us suppose the time-out list consists of $m > 0$ items, and that we need to insert time-out t such that $t.\mathsf{deadline} \geq r_m$. In this case we simply tail the item and initialize it so that

$$t.\mathsf{running} \leftarrow t.\mathsf{deadline} - r_m.$$

Observation 2 *Note how insertion at the end of the list is the only way to prolong the range of action from a certain* $[0, r_m]$ *to a larger* $[0, r_{m+1}]$.

Deletion The other basic management operation on the time-out list is deletion. As we had three possible insertions, likewise we distinguish here deletion from top, from the middle, and from the end of the list.

Deletion from top. If the list is a singleton we are in a trivial case. Let us suppose there are at least two items in the list. Let us call t the top of the list and u the next element—the one that will be promoted to top of the list. From its definition we know that

$$\begin{aligned} r_2 &= u.\mathsf{running} + r_1 \\ &= u.\mathsf{running} + t.\mathsf{running} - (\mathsf{TimeNow} - \mathsf{tom.start\_time}). \quad (3) \end{aligned}$$

By (1), the bracketed quantity is the elapsed time. Then the amount of absolute time units that separate current time from the expiration time is given by $u.\mathsf{running} + t.\mathsf{running}$. In order to "behead" the list we therefore need to update t as follows:

$$u.\mathsf{running} \leftarrow u.\mathsf{running} + t.\mathsf{running}.$$

Deletion from the middle. Let us say we have two consecutive time-outs in our list, t followed by u, such that t is not the top of the list. With a reasoning similar to the one just followed we get to the same conclusion—before physically purging t off the list we need to perform the following step:

$$u.\mathsf{running} \leftarrow u.\mathsf{running} + t.\mathsf{running}.$$

Deletion from the end. Deletion from the end means deleting an entry which is not referenced by any further item in the list. Physical deletion can be performed with no need for updating. Only, the interval of action is shortened.

Observation 3 *Variable* tom.start_time *is never set when deleting from or inserting entries into a time-out list, except when inserting the first element: in such case, that variable is set to the current value of* TimeNow.

Figure 3 shows the action of the server-side protocol: In **1.**, a 330ms time-out called **A** is inserted in the list. In **2.**, after 100ms, **A** has been reduced to 230ms and a 400ms time-out, called **B**, is inserted (its value is 170ms, i.e., 400-230ms). Another 70ms have passed in **3.**, so **A** has been reduced to 160ms. At that point, a 510ms time-out, **C** is inserted and goes at the third position. In **4.**, after 160ms, time-out **A** occurs—**B** becomes then the top of the list; its decrementation starts. In **5.** another 20ms have passed and **B** is at 150ms—at that point a 230ms time-out, called **D** is inserted. Its position is in between **B** and **C**, therefore this latter is adjusted. In **6.**, after 150ms, **B** occurs and **D** goes on top.

3 Discussion

In this section we show that the syntactical constructs in Table 1 can be expressed in terms of our class of time-outs. We do so by considering three classical failure detectors and providing their time-out based specifications.

Let us consider the classical formulation of eventually perfect failure detector $\mathcal{D}$ [1]. The main idea of the protocol is to require each task to send a "heartbeat" to its fellows and maintain a list of tasks suspected to have failed. A task identifier q enters the list of task p if no heartbeat is received by p during a certain amount of time, $\Delta_p(q)$, initially set to a default value. This value is increased when late heartbeats are received.

The basic structure of $\mathcal{D}$ is that of a coroutine with three concurrent processes, two of which execute a task periodically while the third one is triggered by the arrival of a message:

Every process p executes the following:

output$_p \leftarrow 0$
for all $q \in \Pi$
  $\Delta_p(q) \leftarrow$ default time interval

cobegin
  —— *Task 1:* **repeat periodically**
    send "p-is-alive" to all

  —— *Task 2:* **repeat periodically**
    for all $q \in \Pi$
      if $q \notin$ *output*$_p$ and p did not receive "q-is-alive" during
        the last $\Delta_p(q)$ ticks of p's clock **then**
          output$_p \leftarrow$ *output*$_p \cup \{q\}$

  —— *Task 3:* **when** received "q-is-alive" for some q
    if $q \in$ *output*$_p$
      output$_p \leftarrow$ *output*$_p - \{q\}$
      $\Delta_p(q) \leftarrow \Delta_p(q) + 1$
coend.

We call the **repeat periodically** in *Task 1* a "multiplicity 1" repeat, because indeed a single action (sending a "p-is-alive" message) has to be tracked, while we call "multiplicity q" repeat the one in *Task 2*, which requires to check q events.

Our reformulation of the above code is as follows:

Every process p executes the following:

```
timeout_t t_task1, t_task2[NPROCS];
task_t p, q;
for (q=0; q<NPROCS; q++) {
  Δ_p[q] = DEFAULT_TIMEOUT;
  output_p[q] = TRUST;
}

/* "⇝" is our symbol for the "address-of" operator */
tom_declare(⇝t_task1, TOM_CYCLIC, TOM_SET_ENABLE, p, 0, Δ_p[q]);
tom_set_action(⇝t_task1, action_Repeat_Task1);
tom_insert(⇝t_task1);
```

```
for (q=0; q<NPROCS; q++) {
  if (p ≠ q) {
    tom_declare(t_task2+q, TOM_CYCLIC, TOM_SET_ENABLE, q, 0, Δ_p[q]);
    tom_set_action(t_task2+q, action_Repeat_Task2);
    tom_insert(⇝t_task2);
  }
}

do {
  getMessage(⇝m);
  switch (m.type)  {
    TASK1;
    TASK2;
    TASK3;
  }
} forever;
```

where tasks and actions are defined as follows:

```
TASK1 ≡ case REPEAT_TASK1:
          sendAll(I_AM_ALIVE);
        break;
TASK2 ≡ case REPEAT_TASK2:
          q = m.id;
          if (output_p[q] ≡ TRUST)
            output_p[q] = SUSPECT;
        break;
TASK3 ≡ case I_AM_ALIVE:
          q = m.sender;
          if (output_p[q] ≡ SUSPECT)  {
            output_p[q] = TRUST;
            Δ_p(q) = Δ_p(q) + 1;
          }
        break;

action_Repeat_Task1() {
        message_t m;
        m.type = REPEAT_TASK1;
        Send(m, p);
```

```
}
action_Repeat_Task2(timeout_t *t) {
        message_t m;
        m.type = REPEAT_TASK2;
        m.id = t->id;
        Send(m, p);
}
```

We can draw the following observations:

- Our syntax is less abstract than the one adopted in the classical formulation. Indeed we have deliberately chosen a syntax very similar to that of programming languages such as C or C++. Behind the lines, we assume also a similar semantics.

- Our syntax is more strongly typed: we have deliberately chosen to define (most of) the objects our code deals with.

- We have systematically avoided set-wise operations such as union, complement or membership by translating sets into arrays as, e.g., in

  $$output_p \leftarrow output_p \cup \{q\},$$

  which we changed into

  $$output_p[q] = \texttt{PRESENT}.$$

- We have systematically rewritten the abstract constructs `repeat periodically` as one or more time-outs (depending on their multiplicity). Each of these time-out has an associated action that sends one message to the client process, p. This means that

  1. time-related event "it's time to send p-is-alive to all" becomes event "message `REPEAT_TASK1` has arrived."
  2. time-related events "it's time to check whether q-is-alive has arrived" becomes event "message (`REPEAT_TASK2`, id=q) has arrived."

- Due to the now homogeneous nature of the possible events (that now are all represented by message arrivals) a single process may manage those events through a multiple selection statement (a switch). In other words, no coroutine is needed anymore.

Through the Literate Programming approach and a compliant tool such as CWEB [14, 9] it is possible to further improve our reformulation. As well known, the CWEB tool allows a pretty printable TEX documentation and a C file ready for compilation and testing to be produced from a single source code. In our experience this link between these two contexts can be very beneficial: testing or even simply using the code provides feedback on the specification of the algorithm, while the improved specification may reduce the probability of design faults and in general increase the quality of the code.

Figure 4 and Figure 5 respectively show a reformulation for the $\mathcal{HB}$ failure detector for partitionable networks [2] and for the group membership failure detector [6] produced with CWEB. In those reformulations, symbols such as τ and $\mathcal{D}_p$ are caught by CWEB and translated into legal C tokens via its "@f" construct [14]. Note also that the expression $m.path[q] \leq$PRESENT in Fig. 5 means "q appears at most once in *path*". A full description of these protocols is out of the scope of this paper—for that we refer the reader to the above cited articles. The focus here is mainly on the syntactical constructs used in them and our reformulations, which include simple translations for the syntactical constructs in Table 1 in terms of our time-out API. A case worth noting is that of the group membership failure detector: here the authors mimic the availability of a cyclic time-out service but intrude its management in their formulation. This management code can be avoided altogether using our approach.

4 A development experience: the DIR net

What we call "DIR net" [15] is the distributed application at the core of the software fault tolerance strategy realized through several European projects [15, 16]. In this section we describe the DIR net and report on how we designed and developed it by means of the TOM system.

The DIR net is a fault-tolerant network of failure detectors connected to other peripheral error detectors (called "Dtools" in what follows). Objective of the DIR net is to ensure consistent fault tolerance strategies throughout the system and play the role of a backbone handling information to and from the Dtools [18].

The DIR net consists of four classes of components. Each processing node in the system runs an instance of a so-called "I'm Alive Task" (IAT) plus an instance of either a "DIR Manager" (DIR-$\mathcal{M}$), or a "DIR Agent" (DIR-$\mathcal{A}$), or a "DIR Backup Agent" (DIR-$\mathcal{B}$). A DIR-$\mathcal{A}$ gathers all error detection messages produced by the Dtools on the current processing node and forwards them to the DIR-$\mathcal{M}$ and the DIR-$\mathcal{B}$'s. A DIR-$\mathcal{B}$ is a DIR-$\mathcal{A}$ which also maintains its messages into a database located in central memory. It is connected to DIR-$\mathcal{M}$ and to the other DIR-$\mathcal{B}$'s and

Time-out	Caller	Action	Cyclic?
$t_{\text{IA\_SET}}$	DIR-x	On TimeNow + $d_{\text{IA\_SET}}$ do send $m_{\text{IA\_SET\_ALARM}}$ to Caller	Yes
$t_{\text{IA\_CLR}}$	IAT	On TimeNow + $d_{\text{IA\_CLR}}$ do send $m_{\text{IA\_CLR\_ALARM}}$ to IAT	Yes

Table 3: Description of messages $m_{\text{IA\_SET\_ALARM}}$ and $m_{\text{IA\_CLR\_ALARM}}$.

Message	Receiver	Explanation	Action
$m_{\text{IA\_SET\_ALARM}}$	DIR-x	Time to set IAF	IAF $\leftarrow$ `TRUE`
$m_{\text{IA\_CLR\_ALARM}}$	IAT k	Time to check IAF	**if** (IAF $\equiv$ `FALSE`) SendAll(m_{TEIF}, k)
			else IAF $\leftarrow$ `FALSE`,

Table 4: Description of time-outs $t_{\text{IA\_SET}}$ and $t_{\text{IA\_CLR}}$.

is eligible for election as a DIR-$\mathcal{M}$. A DIR-$\mathcal{M}$ is a special case of DIR-$\mathcal{B}$. Unique within the system, the DIR-$\mathcal{M}$ is the one component responsible for running error recovery strategies—see [15] for a description of the latter. Let us use DIR-x to address any non-IAT component (i.e. the DIR-$\mathcal{M}$, or a DIR-$\mathcal{B}$, or a DIR-$\mathcal{A}$.)

An important design goal of the DIR net is that of being tolerant to physical and design faults, both permanent or intermittent, affecting up to all but one DIR-$\mathcal{B}$. This is accomplished also through a failure detection protocol that we are going to describe in the rest of this section.

4.1 The DIR net failure detection protocol

Our protocol consists of a local part and a distributed part. Each of them is realized through our TOM class.

4.1.1 DIR net protocol: local component

As we already mentioned, each processing node hosts a DIR-x and an IAT. These two components run a simple algorithm: they share a local Boolean variable, the "I'm Alive Flag" (IAF). The DIR-x has to periodically set the IAF to `TRUE` while the IAT has to check periodically that this has indeed occurred and reverts IAF to `FALSE`. If the IAT finds the IAF set to `FALSE` it broadcasts message m_{TEIF} ("this entity is faulty").

The cyclic tasks mentioned above can be easily modeled via two time-outs, $t_{\text{IA\_SET}}$ and $t_{\text{IA\_CLR}}$, described in Table 3 and Table 4 (TimeNow being the system function returning the current value of the clock register.)

Note that the time-outs' alarm functions do not clear/set the flag—doing so a hung DIR-x would go undetected. On the contrary, those functions trigger the

transmission of messages that once received by healthy components trigger the execution of the meant actions.

The following is a pseudo-code for the IAT algorithm:

The IAT k executes as follows:

```
timeout_t  t_IA_CLR ;
msg_t  activationMessage , m;

tom_declare(⇝t_IA_CLR , TOM_CYCLIC,
    TOM_SET_ENABLE, IAT_CLEAR_TIMEOUT, 0, d_IA_CLR);
tom_set_action(⇝t_IA_CLR , actionSendm_IA_CLR_ALARM);
tom_insert(⇝t_IA_CLR);

Receive(activationMessage);

forever {
  Receive(m);
  if (m.type ≡ m_IA_CLR_ALARM)
    if (IAF ≡ TRUE) IAF ← FALSE;
    else SendAll(m_TEIF , k); delete_timeout(⇝t_IA_CLR);
}

actionSendm_IA_CLR_ALARM() { Send(m_IA_CLR_ALARM , IAT k); }
```

The time-out formulation of the IAT algorithm is given in next section.

4.1.2 DIR net protocol: distributed component

The resilience of the DIR net to crash faults comes from the DIR-$\mathcal{M}$ and the DIR-$\mathcal{B}$'s running the following distributed algorithm of failure detection:

Algorithm DIR-$\mathcal{M}$ Let us call `mid` the node hosting the DIR-$\mathcal{M}$ and b the number of processing nodes that host a DIR-$\mathcal{B}$. The DIR-$\mathcal{M}$ has to send cyclically a m_{MIA} ("Manager-Is-Alive") message to all the DIR-$\mathcal{B}$'s each time time-out $t_{\text{MIA\_A}}$ expires—this is shown in the right side of Fig. 6. Obviously this is a multiplicity b "repeat" construct, which can be easily managed through a cyclic time-out with an action that signals that a new cycle has begun. In this case the action is "send a message of type $m_{\text{MIA\_A\_ALARM}}$ to the DIR-$\mathcal{M}$."

The manager also expects periodically a (m_{TAIA}, i) message ("This-Agent-Is-Alive") from each node where a DIR-$\mathcal{B}$ is expected to be running. This is easily

accomplished through a vector of $(t_{\text{TAIA\_A}}, \mathbf{i})$ time-outs. The left part of Fig. 6 shows this for node i. When time-out $(t_{\text{TAIA\_A}}, p)$ expires it means that no (m_{TAIA}, p) message has been received within the current period. In this case the DIR-$\mathcal{M}$ enters what we call a "suspicion period". During such period the manager needs to distinguish the case of a late DIR-$\mathcal{B}$ from a crashed one. This is done by inserting a non-cyclic time-out, namely $(t_{\text{TEIF\_A}}, p)$.

During the suspicion period only one out of the following three events may occur:

1. A late (m_{TAIA}, p) is received.
2. A (m_{TEIF}, p) from IAT at node p is received.
3. Nothing comes in and the time-out expires.

In case 1. we get out of the suspicion period, conclude that DIR-$\mathcal{B}$ at node p was simply late and go back waiting for the next (m_{TAIA}, p).

It is the responsibility of the user to choose meaningful values for the time-outs' deadlines. By "meaningful" we mean that those values should match the characteristics of the environment and represent a good trade-off between the following two risks:

overshooting, i.e., choosing too large values for the deadlines. This decreases the probability of false negatives (regarding a slow process as a failed process; this is known as accuracy in failure detection terminology) but increases the detection latency;

undershooting, namely under-dimensioning the deadlines. This may increase considerably false negatives but reduces the detection latency of failed processes.

Under the hypotheses of properly chosen time-outs' deadlines, and that of a single, stable environment[2], the occurrences of late (m_{TAIA}, p) messages should be exceptional. This event would translate in a false deduction uncovered in the next cycle. Further late messages would postpone a correct assessment, but are considered as an unlikely situation given the above hypotheses. An alternative and better approach would be to track the changes in the environment. For the case at hand this would mean that the time-outs' deadlines should be adaptively adjusted. This could be possible, e.g., through an approach such as in [19].

[2]We call an environment "stable" when it does not change drastically its characteristics except under erroneous and exceptional conditions. Single environments are typical of fixed (non-mobile) applications.

If 2. is the case we assume the remote component has crashed though its node is still working properly as the IAT on that node still gives signs of life. Consequently we initiate an error recovery step. This includes sending a "`WAKEUP`" message to the remote IAT so that it spawns another DIR-$\mathcal{B}$ on that node.

In case 3. we assume the entire node has crashed and initiate node recovery.

Underlying assumption of our algorithm is that the IAT is so simple that if it fails then we can assume the whole node has failed.

Algorithm DIR-$\mathcal{B}$ This algorithm is also divided into two concurrent tasks. In the first one DIR-$\mathcal{B}$ on node i has to cyclically send (m_{TAIA}, i) messages to the manager, either in piggybacking or when time-out $t_{\text{TAIA\_B}}$ expires. This is represented in the right side of Fig. 7.

The DIR-$\mathcal{B}$'s in turn periodically expect a m_{MIA} message from the DIR-$\mathcal{M}$. As evident when comparing Fig. 6 with Fig. 7, the DIR-$\mathcal{B}$ algorithm is very similar to the one of the manager: also DIR-$\mathcal{B}$ enters a suspicion period when its manager does not appear to respond quickly enough—this period is managed via time-out $t_{\text{TEIF\_B}}$, the same way as in DIR-$\mathcal{M}$. Also in this case we can get out of this state in one out of three possible ways: either

1. a late $(m_{\text{MIA\_B\_ALARM}}, \text{mid})$ is received, or
2. a $(m_{\text{TEIF}}, \text{mid})$ sent by the IAT at node mid is received, or
3. nothing comes in and the time-out expires.

In case 1. we get out of the suspicion period, conclude that the manager was simply late, go back to normal state and start waiting for the next $(m_{\text{MIA}}, \text{mid})$ message. Also in this case, a wrong deduction shall be detected in next cycles. If 2. we conclude the manager has crashed though its node is still working properly, as its IAT acted as expected. Consequently we initiate a manager recovery phase structured similarly to the DIR-$\mathcal{B}$ recovery step described in Sect. 4.1.2. In case 3. we assume the node of the manager has crashed, elect a new manager among the DIR-$\mathcal{B}$'s, and perform a node recovery phase.

Table 5 summarizes the DIR-$\mathcal{M}$ and DIR-$\mathcal{B}$ algorithms.

We have developed the DIR net using the Windows TIRAN libraries [16] and the CWEB system of structured documentation.

4.2 Special services

4.2.1 Configuration

The management of a large number of time-outs may be an error prone task. To simplify it, we designed a simple configuration language. Figure 8 shows an ex-

Time-out	Caller	Action	Cyclic?
$t_{\text{MIA\_A}}$	DIR-$\mathcal{M}$	Every $d_{\text{MIA\_A}}$ do send $m_{\text{MIA\_A\_ALARM}}$ to DIR-$\mathcal{M}$	Yes
$t_{\text{TAIA\_A}}[i]$	DIR-$\mathcal{M}$	Every $d_{\text{TAIA\_A}}$ do send $(m_{\text{TAIA\_A\_ALARM}}, i)$ to DIR-$\mathcal{M}$	Yes
$t_{\text{TEIF\_A}}[i]$	DIR-$\mathcal{M}$	On TimeNow + $d_{\text{TEIF\_A}}$ do send $(m_{\text{TEIF\_A\_ALARM}}, i)$ to DIR-$\mathcal{M}$	No
$t_{\text{TAIA\_B}}$	DIR-$\mathcal{B}$ j	Every $d_{\text{TAIA\_B}}$ do send $m_{\text{TAIA\_B\_ALARM}}$ to DIR-$\mathcal{B}$ j	Yes
$t_{\text{MIA\_B}}$	DIR-$\mathcal{B}$ j	Every $d_{\text{MIA\_B}}$ do send $m_{\text{MIA\_B\_ALARM}}$ to DIR-$\mathcal{B}$ j	Yes
$t_{\text{TEIF\_B}}$	DIR-$\mathcal{B}$ j	On TimeNow + $d_{\text{TEIF\_B}}$ do send $m_{\text{TEIF\_B\_ALARM}}$ to DIR-$\mathcal{B}$ j	No

Message	Receiver	Explanation	Action
(m_{TAIA}, i)	DIR-$\mathcal{M}$	DIR-$\mathcal{B}$ i is OK	(Re-)Insert or renew $t_{\text{TAIA\_A}}[i]$
$m_{\text{MIA\_A\_ALARM}}$	DIR-$\mathcal{M}$	A new heartbeat is required	Send m_{MIA} to all DIR-$\mathcal{B}$'s
$m_{\text{TAIA\_A\_ALARM}}$	DIR-$\mathcal{M}$	Possibly DIR-$\mathcal{B}$ i is not OK	Delete $t_{\text{TAIA\_A}}[i]$, insert $t_{\text{TEIF\_A}}[i]$
(m_{TEIF}, i)	DIR-$\mathcal{M}$	DIR-$\mathcal{B}$ i crashed	Declare DIR-$\mathcal{B}$ i crashed
$(m_{\text{TEIF\_A\_ALARM}}, i)$	DIR-$\mathcal{M}$	Node i crashed	Declare node i crashed
m_{MIA}	DIR-$\mathcal{B}$ j	DIR-$\mathcal{M}$ is OK	Renew $t_{\text{MIA\_B}}$
$m_{\text{TAIA\_B\_ALARM}}$	DIR-$\mathcal{B}$ j	A new heartbeat is required	Send (m_{TAIA}, j) to DIR-$\mathcal{M}$
$m_{\text{MIA\_B\_ALARM}}$	DIR-$\mathcal{B}$ j	Possibly DIR-$\mathcal{M}$ is not OK	Delete $t_{\text{MIA\_B}}$, insert $t_{\text{TEIF\_B}}$
m_{TEIF}	DIR-$\mathcal{B}$ j	DIR-$\mathcal{M}$ crashed	Declare DIR-$\mathcal{M}$ crashed
$m_{\text{TEIF\_B\_ALARM}}$	DIR-$\mathcal{B}$ j	DIR-$\mathcal{M}$'s node crashed	Declare DIR-$\mathcal{M}$'s node crashed

Table 5: Time-outs and messages of DIR-$\mathcal{M}$ and DIR-$\mathcal{B}$.

ample of configuration script to specify the structure of the DIR net (in this case, a four node system with three DIR-$\mathcal{B}$'s deployed on nodes 1–3 and the DIR-$\mathcal{M}$ on node 0) and of its time-outs. A translator produces the C header files to properly initialize an instance of the DIR net (see Fig. 9).

4.2.2 Fault injection

Time-outs may also be used to specify fault injection actions with fixed or pseudo-random deadlines. In the DIR net this is done as follows. First we define the time-out:

```
#ifdef INJECT
 tom_declare(&inject, TOM_NON_CYCLIC, TOM_SET_ENABLE,
             INJECT_FAULT_TIMEOUT, i, INJECT_FAULT_DEADLINE);
 tom_insert(tom, &inject);
#endif
```

The alarm of this time-out sends the local DIR-x a message of type "`INJECT_FAULT_TIMEOUT`". Figure 10 shows an excerpt from the actual main loop of the DIR-$\mathcal{M}$ in which this message is processed.

4.2.3 Fault tolerance

A service such as TOM is indeed a single-point-of-failure in that a failed TOM in the DIR net would result in all components being unable to perform their failure detection protocols. Such a case would be indistinguishable from that of a crashed node by the other DIR net components. As well known from, e.g., [20], a single design fault in TOM's implementation could bring the system to a global failure. Nevertheless, the isolation of a *service* for time-out management may pave the way for a cost-effective adoption of multiple-version software fault tolerance techniques [21] such as the well known recovery block [22], or N-version programming [23]. Another possibility would be to use the DIR net algorithm to tolerate faults in TOM. No such technique has been adopted in the current implementation of TOM. Other factors, such as congestion or malicious attacks might introduce performance failures that would impact on all modules that depend on TOM to perform their time-based processing [10].

5 Conclusions

We have introduced a tentative *lingua franca* for the expression of failure detection protocols. TOM has the advantages of being simple, elegant and not ambiguous. Obvious are the many positive relapses that would come from the adoption of a standard, semi-formal representation with respect to the current Babel of informal descriptions—easier acquisition of insight, faster verification, and greater ability to rapid-prototype software systems. The availability of a tool such as TOM is also one of the requirements of the timed-asynchronous system model [25].

Given the current lack of a network service for failure detection, the availability of standard methods to express failure detectors in the application layer is an important asset: a tool like the one described in this paper isolates and crystallizes a part of the complexity required to express failure detection protocols. This complexity may become transparent of the designer, with tangible savings in terms of development times and costs, if more efforts will be devoted to time-outs configuration and automatic adjustments through adaptive approaches such as the one described in [19]. Such optimizations will be the subject of future research. Future plans also include to port our system to AspectJ [24] so as to further enhance programmability and separation of design concerns.

As a final remark we would like to point out how, at the core of our design choices, is the selection of C and literate programming, which proved to be invaluable tools to reach our design goals. Nevertheless we must point out how these choices may turn into intrinsic limitations for the expressiveness of the resulting language. In particular, they enforce a syntactical and semantic structure, that of

the C programming language, which may be regarded as a limitation by those researchers who are not accustomed to that language. At the same time we would like to remark also that those very choices allow us a straightforward translation of our constructs into a language like Promela [26], which resembles very much a C language augmented with Hoare's CSP [27]. Accordingly, our future work in this framework shall include the adoption of the Promela extension of Prof. Bošnački, which allows the verification of concurrent systems that depend on timing parameters [28]. Interestingly enough, this version of Promela includes new objects, called discrete time countdown timers, which are basically equivalent to our non-cyclic time-outs. Our goal is to come up with a tool that generates from the same literate programming source (1) a pretty printout in TEX, (2) C code ready to be compiled and run, and (3) Promela code to verify some properties of the protocol.

Acknowledgment

We acknowledge the work by Alessandro Sanchini, who developed the communication library used by our tool, and the many and valuable comments of our reviewers.

References

[1] Chandra, T. D., and Toueg, S.: 'Unreliable failure detectors for reliable distributed systems', Journal of the ACM, 1996, 43, pp. 225–267.

[2] Aguilera, M. K., Chen, W., and Toueg, S.: 'Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks', Theoretical Computer Science, 1999, 1, pp. 3–30.

[3] Bertier, M., Marin, O., and Sens, P.: 'Implementation and performance of an adaptable failure detector'. Proceedings of the International Conference on Dependable Systems and Networks (DSN '02), June 2002. IEEE Society Press.

[4] Chen, W., Toueg, S., and Aguilera, M. K.: 'On the quality of service of failure detectors', IEEE Trans. on Computers, 2002, 51, pp. 561–580.

[5] Hayashibara, N.: 'Accrual Failure Detectors'. PhD thesis, School of Information Science, Japan Advanced Institute of Science and Technology, 2004.

[6] Raynal, M., and Tronel, F.: 'Group membership failure detection: a simple protocol and its probabilistic analysis', Distributed Systems Engineering, 1999, 6, pp. 95–102.

[7] Hayashibara, N., Défago, X., Yared, R., and Katayama, T.: 'The φ accrual failure detector'. Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04). Florianopolis, Brazil, October 2004, pp. pp. 66–78.

[8] van Renesse, R., Minsky, Y., and Hayden, M.: 'A gossip-style failure detection service', Proceedings of Middleware '98, Davies, N., Seitz, J. and Raymond, K. (Eds.), The Lake District, UK, September 1998, pp. 55–70. Springer.

[9] Knuth, D. E.: 'Literate programming', The Comp. Journal, 1984, 27, pp. 97–111.

[10] De Florio, V., and Blondia, C.: 'Dynamics of a time-outs management system', Complex Systems, 2006, 16, pp. 209–223.

[11] Anonymous: 'Embedded Parix Programmer's Guide', in Parsytec: 'Parsytec CC Series Hardware Documentation' (Parsytec GmbH, Aachen, Germany, 1996).

[12] http://www-128.ibm.com/developerworks/linux/library/l-posix1.html. Accessed September 19, 2007.

[13] Tanenbaum, A. S.: 'Computer Networks' (Prentice-Hall, London, 1996, 3rd edn.)

[14] Knuth, D. E., and Levy, S.: 'The CWEB System of Structured Documentation' (Addison–Wesley, Reading, MA, 1993, 3rd edn.)

[15] De Florio, V.: 'Application-layer Fault-Tolerance Protocols' (IGI-Global, Hershey, PA, 2009), ISBN 1-60566-182-1.

[16] Botti, O., De Florio, V., Deconinck, G., Lauwereins, R., Cassinari, F., Donatelli, S., Bobbio, A., Klein, A., Kufner, H., Thurner, E., and Verhulst, E.: 'The TIRAN approach to reusing software implemented fault tolerance'. Proc. of the 8th Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP'00). Rhodos, Greece, January 1999, pp. 325–332, IEEE Comp. Soc. Press.

[17] Deconinck, G., De Florio, V., Dondossola, G., and Szanto, J.: 'Integrating recovery strategies into a primary substation automation system'. Proc. of the International Conference on Dependable Systems and Networks (DSN-2003). 2003, IEEE Comp. Soc. Press.

[18] De Florio, V., Deconinck, G., and Lauwereins, R.: 'An algorithm for tolerating crash failures in distributed systems'. Proc. of the 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS). Edinburgh, Scotland, April 2000, pp. 9–17. IEEE Comp. Soc. Press.

[19] De Florio, V., and Blondia, C.: 'Adaptive data integrity through dynamically redundant data structures'. Proc. of the third international Conference on Availability, Reliability and Security (ARES 2008). Barcelona, Spain, March 2007, IEEE Computer Society.

[20] http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, accessed April 2009.

[21] Lyu, M. R.: 'Reliability-oriented software engineering: Design, testing and evaluation techniques', IEE Proceedings – Software, 1998, 145, pp. 191–197, special issue on Dependable Computing Systems.

[22] Randell, B., and Xu, J.: 'The evolution of the recovery block concept', in Lyu, M. (Ed.): 'Software Fault Tolerance' (John Wiley & Sons, New York, 1995), Chapter 1, pp. 1–21.

[23] Avižienis, A.: 'The methodology of N-version programming', in Lyu, M. (Ed.): 'Software Fault Tolerance' (John Wiley & Sons, New York, 1995), Chapter 2, pp. 23–46.

[24] http://www.eclipse.org/aspectj, accessed October 2009.

[25] Cristian, F., and Fetzer, C.: 'The Timed Asynchronous Distributed System Model', IEEE Trans. on Parallel and Distributed Systems, June 1999, 10, 6, pp. 642–657.

[26] Holzmann, G. J.: 'Design and Validation of Computer Protocols' (Prentice-Hall, 1991).

[27] Hoare, C. A. R.: 'Communicating sequential processes', Comm. ACM, 1978, 21, pp. 667–677.

[28] Bošnački, D., and Dams, D.: 'Discrete-time Promela and Spin'. Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '98). 1998, Lecture Notes in Computer Science 1486, pp. 307–310, Springer-Verlag.

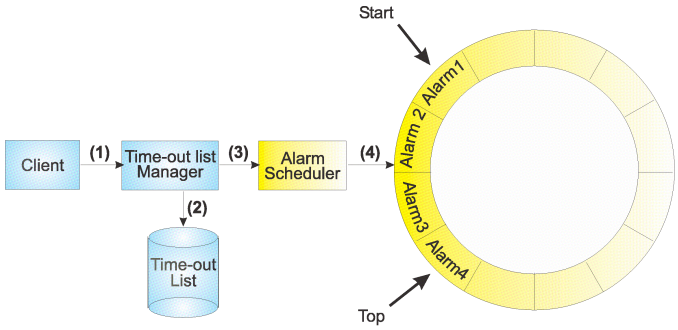


Figure 1: Architecture of the time-out management system.

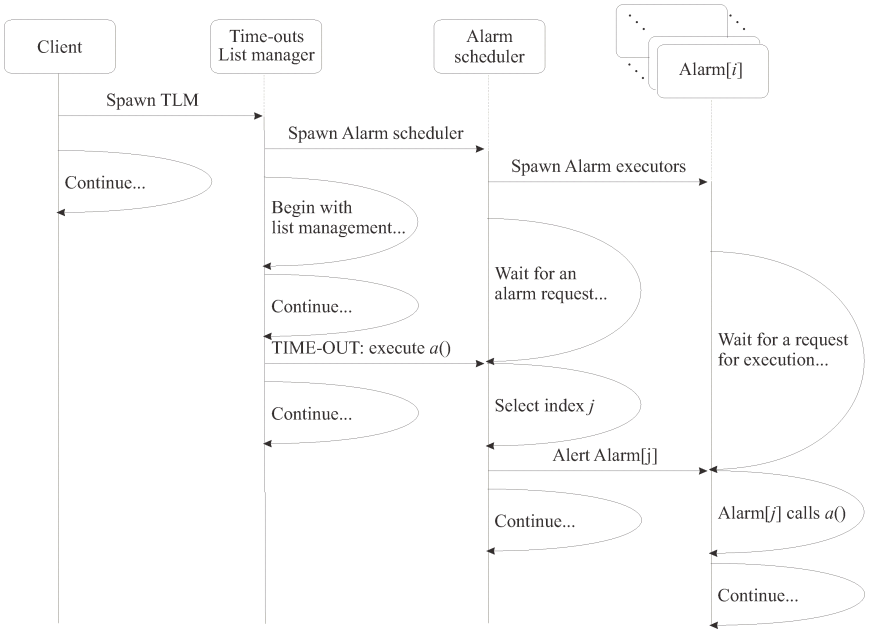


Figure 2: Sequence diagram for the tasks of the time-outs manager.

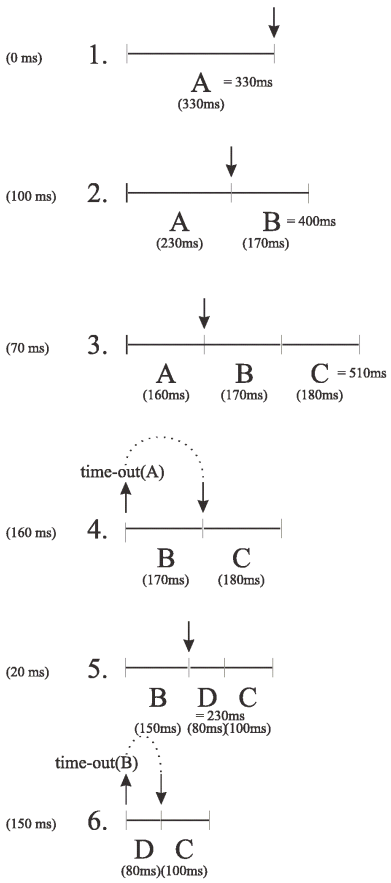


Figure 3: Operating scenario of the time-out manager.

1. Code of the $\mathcal{HB}$ failure detector for partitionable networks.
Aguilera, Chen and Toueg, Theoretical Computer Science n.1, 1999.

```
#define HEARTBEAT 1
#define ITTB 2
#define SOME_PERIOD 100000
#define FOREVER 1
#define PRESENT 1
#define ABSENT 0
```
⟨ Initialisation 2 ⟩

2. Every process p executes the following:

```
⟨ Initialisation 2 ⟩ ≡
  main( )
  {
    timeout_t τ_t2b;
    message_t m;
    for (q = 0; q < NPROCS; q++) {
      D_p[q] = 0;
      path[q] = ABSENT;
    }
    tom_declare(&τ_t2b, TOM_CYCLIC, TOM_SET_ENABLE, 1, 1, 1);
    tom_set_action(&τ_t2b, actionItsTimeToBroadcast);    /* sends ITTB */
    tom_set_deadline(&τ_t2b, SOME_PERIOD);    /* every 100000 ticks */
    tom_insert(&nextb);
    do {
      getMessage(&m);    /* sets m.date */
      switch (m.type) {
        ⟨ Task1 3 ⟩
        ⟨ Task2 4 ⟩
      }
    } while (FOREVER);
  }
```
This code is used in section 1.

3. Task 1

```
⟨ Task1 3 ⟩ ≡
case ITTB: D_p[p] = D_p[p] + 1;
  m.type = HEARTBEAT, m.path = p;
  for (q = 0; q < NPROCS; q++)
    if (isneighbor(q, p)) sendMessage(m, q);
  break;
```
This code is used in section 2.

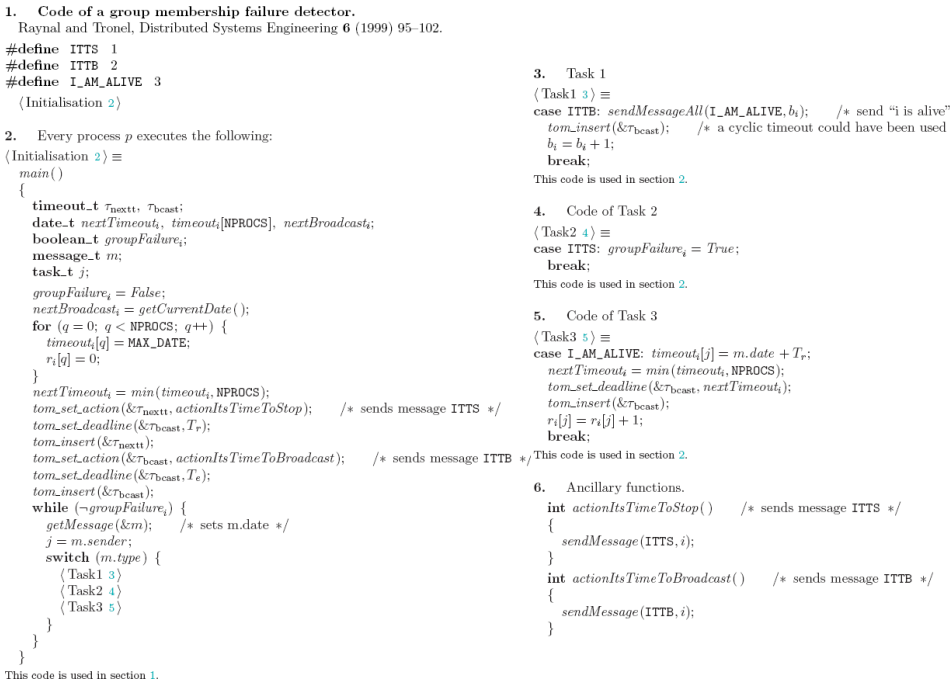
4. Code of Task 2

```
⟨ Task2 4 ⟩ ≡
case HEARTBEAT:
  for (q = 0; q < NPROCS; q++)
    if (m.path[q] ≠ ABSENT) D_p[p] = D_p[p] + 1;
  m.path[p] = m.path[p] + 1;
  for (q = 0; q < NPROCS; q++)
    if (isneighbor(q, p) ∧ m.path[q] ≤ PRESENT) sendMessage(m, q);
  break;
```
This code is used in section 2.

5. Extra functions

```
  int actionItsTimeToBroadcast( )    /* sends ITTB to caller */
  {
    sendMessage(ITTB, p);
  }
```

Figure 4: Reformulation of the $\mathcal{HB}$ failure detector for partitionable networks [2].



```
1.  Code of a group membership failure detector.
    Raynal and Tronel, Distributed Systems Engineering 6 (1999) 95–102.
#define ITTS  1
#define ITTB  2
#define I_AM_ALIVE  3
  ⟨ Initialisation 2 ⟩

2.  Every process p executes the following:
⟨ Initialisation 2 ⟩ ≡
  main( )
  {
    timeout_t τ_nextt, τ_bcast;
    date_t nextTimeout_i, timeout_i[NPROCS], nextBroadcast_i;
    boolean_t groupFailure_i;
    message_t m;
    task_t j;
    groupFailure_i = False;
    nextBroadcast_i = getCurrentDate( );
    for (q = 0; q < NPROCS; q++) {
      timeout_i[q] = MAX_DATE;
      r_i[q] = 0;
    }
    nextTimeout_i = min(timeout_i, NPROCS);
    tom_set_action(&τ_nextt, actionItsTimeToStop);      /* sends message ITTS */
    tom_set_deadline(&τ_bcast, T_r);
    tom_insert(&τ_nextt);
    tom_set_action(&τ_bcast, actionItsTimeToBroadcast);      /* sends message ITTB */
    tom_set_deadline(&τ_bcast, T_e);
    tom_insert(&τ_bcast);
    while (¬groupFailure_i) {
      getMessage(&m);      /* sets m.date */
      j = m.sender;
      switch (m.type) {
        ⟨ Task1 3 ⟩
        ⟨ Task2 4 ⟩
        ⟨ Task3 5 ⟩
      }
    }
  }
This code is used in section 1.

3.  Task 1
⟨ Task1 3 ⟩ ≡
case ITTB: sendMessageAll(I_AM_ALIVE, b_i);      /* send "i is alive" to all */
  tom_insert(&τ_bcast);      /* a cyclic timeout could have been used here */
  b_i = b_i + 1;
  break;
This code is used in section 2.

4.  Code of Task 2
⟨ Task2 4 ⟩ ≡
case ITTS: groupFailure_i = True;
  break;
This code is used in section 2.

5.  Code of Task 3
⟨ Task3 5 ⟩ ≡
case I_AM_ALIVE: timeout_i[j] = m.date + T_r;
  nextTimeout_i = min(timeout_i, NPROCS);
  tom_set_deadline(&τ_bcast, nextTimeout_i);
  tom_insert(&τ_bcast);
  r_i[j] = r_i[j] + 1;
  break;
This code is used in section 2.

6.  Ancillary functions.
  int actionItsTimeToStop( )      /* sends message ITTS */
  {
    sendMessage(ITTS, i);
  }
  int actionItsTimeToBroadcast( )      /* sends message ITTB */
  {
    sendMessage(ITTB, i);
  }
```

Figure 5: Reformulation of the group membership failure detector [6].

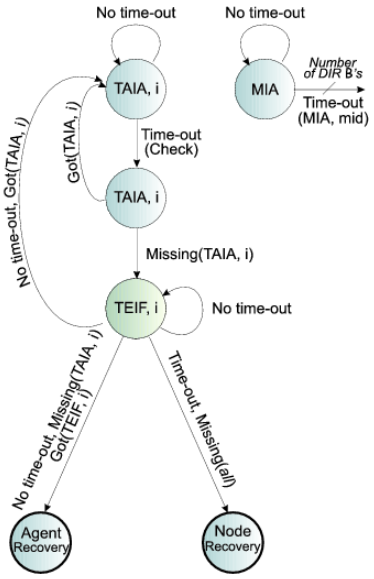


Figure 6: Algorithm of the DIR-$\mathcal{M}$.

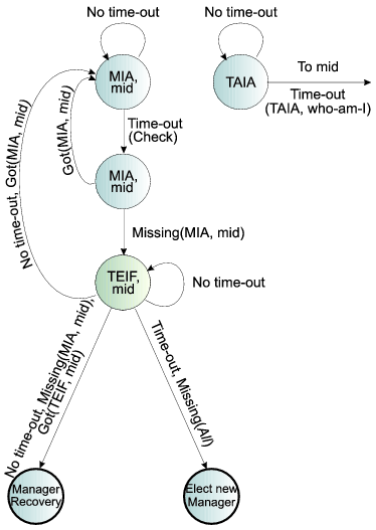


Figure 7: Algorithm DIR-$\mathcal{B}$.

```
# include files
#       defines are importable from include files via #include statements
INCLUDE "my_definitions.h"
INCLUDE "../BACKBONE.H"
# definitions
#       definitions start with the 'DEFINE' keyword, followed
#       by an integer, an interval, or a list, followed
#       by the equal sign and a role, that may be
#       ASSISTANTs or MANAGER
NPROCS = 4
DEFINE 2-4 = ASSISTANTS
DEFINE 1 = MANAGER

# NPROCS = 2
# DEFINE 2 = ASSISTANT

MIA_SEND_TIMEOUT = 800000   # Manager Is Alive -- manager side
TAIA_RECV_TIMEOUT = 1800000  # This Agent Is Alive timeout -- manager side

MIA_RECV_TIMEOUT = 1500000   # Manager Is Alive -- backup side
TAIA_SEND_TIMEOUT = 1000000  # This Agent Is Alive timeout -- backup side

TEIF_TIMEOUT     = 1800000   # after this time a suspected node is assumed
                             # to have crashed.

I'M ALIVE_CLEAR_TIMEOUT = 900000 # I'm Alive timeout -- clear IA flag
I'M ALIVE_SET_TIMEOUT = 1400000  # I'm Alive timeout -- set and checks IA flag

REQUEST_DB_TIMEOUT = 2000000
REPLY_DB_TIMEOUT = 4000000

MID_TIMEOUT = 1000000 # if a TEIF is receaved, up to MID_TIMEOUT ticks
                      # are allowed for reintegrating a new manager,
                      # otherwise, the node of the manager is considered
                      # to be dead.
```

Figure 8: Excerpt from the configuration script of the DIR net.

```
bash-2.05b$ art -s
Ariel translator, v4.0g 2-Dec-2004, (c) 2004 Universiteit Antwerpen.
Parsing file .ariel...
[ Including file `my_definitions.h' ...9 associations have been stored. ]
[ Including file `../BACKBONE.H' ...55 associations have been stored. ]
        if-then-else: ok
...done (148 lines.)
Output written in file .rcode.
Watchdogs configured.
N-version tasks configured.
Logicals written in file LogicalTable.csv.
Tasks written in file TaskTable.csv.
static version
Preloaded r-codes written in file ../trl.h.
Time-outs written in file ../timeouts.h.
Identifiers written in file ../identifiers.h.
Alpha-count parameters written in file ../alphacount.h.
Press ^C to finish processing...
```

Figure 9: Configuration tool of the DIR net.

18. This loop is the real core of the manager. It has to deal with a number of messages coming from the timeout manager, its fellow backups, the recovery thread, the remote I'm Alive Tasks. The core of the fault-tolerant strategy of the DIR net is in here.

```
⟨ manager loop (waiting for incoming messages) 18 ⟩ ≡
  while (1) {
    ⟨ wait for an incoming message 53 ⟩
    tom_dump(tom);
    switch (message.type) {
    case INJECT_FAULT_TIMEOUT:
      LogError(EC_ERROR, "Manager␣loop", "Fault␣injection");
      tom_close(tom);      /* the time-out manager is detached */
      break;
    case IA_FLAG_TIMEOUT:
      LogError(EC_ERROR, "Manager␣loop", "IA_FLAG_TIMEOUT␣message␣->␣clear␣IA-flag.");
        /* time to clear the IA-flag! */
      ⟨ clear IA-flag 16 ⟩
      break;
    case MIA_TIMEOUT:
      LogError(EC_ERROR, "Manager␣loop",
          "MIA_TIMEOUT␣message␣(time␣to␣send␣a␣MIA␣to␣Backup␣%d).", message.subid);
        /* time to send a MIA to a backup */
      ⟨ send MIA to backup subid 19 ⟩
      tom_dump(tom + message.subid);
      tom_renew(tom, mia + message.subid);
      break;
```

Figure 10: Excerpt from the CWEB source of the DIR net.