Distributed System Theory Project
Consensus Problem
Implementation in Erlang

Giovanni Simoni
Student Id 142955

January 10, 2012

Contents

1 Introduction

The *Consensus problem* is a well-known abstract problem concerning *distributed systems*: many important results relay on it, and many algorithms solving it are available.

A correct run of a *Consensus algorithms* requires the *processes* of an asynchronous distributed environment to achieve the *quorum* on a certain decision and to select it consistently. Such decision can be generalized by the choice of a value for a truth assignment.

The YUNA project implements a well designed and configurable environment which follows the typical assumptions we can find in literature. The program is written in ERLANG (`http://erlang.org`), a functional programming language developed by ERICSSON, which is oriented to concurrent software. It is particularly well-suited for *highly reliable systems*.

On top of YUNA I implemented one of the many available algorithms for solving the *Consensus problem*, namely the one proposed by Mostefaoui and Raynal [1]. As this algorithm requires to be equipped with a *Failure Detector*, I also implemented the basic version of the Failure Detector Protocol, as proposed by Robbert van Renesse, Yaron Minsky, and Mark Hayden [2]. The implementation is based on a failure detector of class $\diamond S$ (*Eventually Correct*) [3].

1.1 Contents of this document

***Section* 2** Gives a quick overview on the ERLANG programming language;

***Section* 3** Explains the core of the program the communication system and the parameters needed by both;

***Section* 4** Explains how the selected *Consensus* algorithm has been implemented over the YUNA platform;

***Section* 5** Shows a little tutorial on how to launch the algorithm, also giving some statistics;

***Section* 6** Gives a reference on the *API* provided by the YUNA platform.

2 ERLANG and OTP

ERLANG is a functional, interpreted and weakly typed language, which is provided with a built-in process management *API* based on message passing. Some interesting features are:

- Easy way of spawning processes;
- Concurrency is implemented with real parallelism, and the language supports *SMP*;
- Processes can send and receive messages asynchronously;
- Dependency chains can be built among processes (so that a crashing process also tears down all the dependent ones);
- Processes can *monitor* other processes, getting notified in case of crash;
- Applications can be easily deployed over the network, with hot swapping of executable code.

This capabilities, plus the *side-effects free* design typical of a functional language, make ERLANG particularly well suited for distributed concurrent applications.

```
test_function (Value) ->
    case Value of
        3 -> io:format("I got the value 3\n");
        goat -> io:format("I got the 'goat' atom\n");
        [] -> io:format("I got an empty list\n");
        X when is_list(X) -> io:format("I got a list\n");
        x -> io:format("I got the atom 'x'");
        _ -> io:format("No idea...\n")
    end.
```

Listing 1: Example of ERLANG code

Apart from the language itself, ERLANG comes with a stable and tested standard library named OTP (OPEN TELECOM PLATFORM), which provides a generic framework for services development and a standard way of structuring applications.

This section defines some keywords and gives a very brief overview on the language and on how an ERLANG application is structured. The purpose is just giving a better understanding of the project: for a more detailed description, please refer to the official ERLANG documentation[4].

2.1 About the syntax

The language syntax is simple and straightforward. Naming conventions determine the semantics of the identifiers. The main data types are the following:

atoms are system-wide constant literals defined by `small_caps_identifiers` or `'single quote strings'`. Function names, module names and boolean values are atoms.

numbers are numerical constants;

tuples are defined as items of any type, comma-separated and `{surrounded, by, braces}`;

lists are sequences of items of any (possibly mixed) type. For instance `[atoms, plus, 1, number]`; also strings are lists;

pids are process identifiers, usually generated trough the language *API* when spawning processes.

More data types allow to work with low-level data streams and native platform low-level objects.

As any *small caps* identifier is a constant literal, *variables* must follow a `CamelCase` naming convention. They are placeholders for items of any type, which can be used in matching. The example shown in *Listing 1* may be clarifying.

The conventional signature of this function is `test_function/1`, where the value *1* declares the *arity* of the function

2.2 About OTP

The language provides a concept named **behaviors**. It resembles the semantics of *class* in HASKELL, or the semantics of *interface* in JAVA: a module declaring a certain **behavior** is forced to provide a set of the callback functions, which can be called by external modules.

This syntactic tool is used by OTP to provide many general-purpose services (henceforth **abstract components**), which use callback functions of user modules as stubs. The callbacks, in turn, are supposed to implement the actual business logic (henceforth **specific components**). OTP uses wisely the language capabilities and provide both synchronous and asynchornous services.

Many generic services are available. Among those:

- `gen_server` implements a *client-server* logic;
- `gen_fsm` implements a *finite state machine* logic;
- `gen_event` implements the logic of an *event handler*.

Moreover, OTP provides a standard way of organizing hierarchically the application: the generic `supervisor` module. The associated **behavior** requires the **specific component** to export an initialization function, which is supposed to return a list of *child specifications*; each specification is a tuple containing all the parameters needed to spawn the *supervised* processes. A **specific component** implementing this logic is called **supervisor**.

Trough child specifications, a supervised processes can be declare as **permanent** (namely it gets always restarted in case of termination), **temporary** (never restarted) and **transient** (restarted only if it terminates abnormally).

Finally, as design pattern, many generic services are started as process trough a specific function call which is part of the **abstract component**. This is valid for `supervisor` as well, thus a **supervisor** can manage **supervisors** in a *supervision hierarchy* (henceforth **application**).

2.2.1 Coding conventions

- The callbacks for **behaviors** of OTP library are usually required to return a tuple of the form `{ok, Status}` or `{error, Reason}`, where `ok` and `error` are *atoms*, while `Status` is the private data of the **specific component** (parameter for the next callback) and `Reason` is the reason which caused the error.
- Except for `init`, the last formal parameter of all **specific component** callbacks is conventionally bound to the afore mentioned private data.

Those conventions are followed by my program as well. See *Section* 6 for further details on the *API*.

3 The YUNA Environment

3.1 Overview

YUNA is a OTP-compliant **application**. It defines the hierarchy shown in Figure 1.

The involved **supervisors** are:

- `main` (file src/infrastructure/main.erl) which is placed at the top of the hierarchy, it just starts the business logic;
- `services` (file src/infrastructure/services.erl) which is in charge of managing the permanent services:
- `peers` (file src/infrastructure/peers.erl) which is a *pool* for the processes which will be executing the required distributed logic.

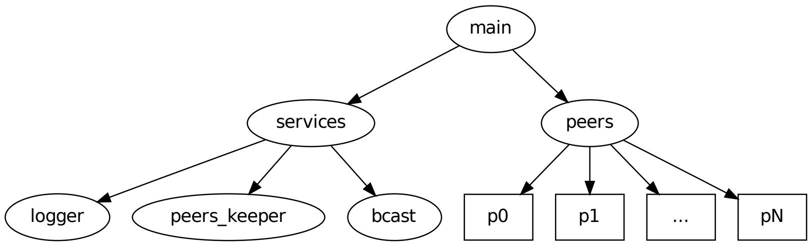

Figure 1: *the* YUNA *hierarchy*

The **services** under the `services` **supervisor** are:

- `logger` (file src/infrastructure/log_serv.erl) which provides a simple *API* for logging events and collecting statistics:
  - It implements an OTP `gen_server`;
  - As it corresponds to a dedicated process, logging messages are enqueued, thus the output text from the processes is never overlapped;
  - In order to reduce the logging overhead, calls to the logger are asynchronous
  - It supports many different calls for collecting statistics and signaling events, producing some output files which can be elaborated by the *gnuplot* software. The files are:
    * *prefix*_node_count (trace the number of nodes in time);
    * *prefix*_est_node_count (trace the approximated number of nodes seen by a subset of the running failure detectors in time);
    * *prefix*_decision_count (trace the number of nodes which decided some value in time);
    * *prefix*_events (events to be shown in the time diagram).

    where *prefix* is a string depending on the configuration file (see *Paragraph 6.3.2* for further details).

- `bcast` (file src/infrastructure/bcast.erl) which reads messages from the input queue and forwards them to all the processes which subscribed the service:
  - Like `logger`, it implements an OTP `gen_server`;
  - The broadcasting also implements a timer which is used to serve subscribers with a periodic beacon (this models the internal clock of distributed nodes, however they are not supposed to consider it as a *global clock*).

- `peers_keeper` is a generic service, namely it's constituted by an **abstract component** (file src/infrastructure/behavs/gen_keeper.erl) which can be used to monitor the working distributed algorithm. It must be associated with a **specific component** which implements the supervision part of the distributed algorithm.

The processes under the `peers` **supervisor** implement the logic associated with the actual distributed algorithm. As for the `peers_keeper`, the running peers are supposed to be **specific components** implementing an **abstract component** (file src/infrastructure/behavs/gen_peer.erl).

Both `gen_keeper` and `gen_peer` export a **behavior** which forces the implementation of some events, constituting an *event-driven* environment. They are also provided with a dedicated *API*, which abstracts away an internal communication protocol.

3.2 Startup dynamics

At startup the application reads the configuration file (ebin/yuna.app) which is required to contain the name of a module implementing the `gen_keeper` **behavior**. Let's call it **specific keeper**. When `gen_keeper` is activated by the `services` **supervisor**, it's parametrized with the name of the **specific keeper** (see *Paragraph* 6.3.2 for further details).

The *API* can now be used by the **specific keeper** to spawn and control the peers, being notified on what's going on as events are risen.

From the *peer* point of view, the concept is pretty much the same: the library function used by **specific keeper** to spawn nodes must be parametrized with the number N of required peers, plus the name of a module implementing the **behavior** of the `gen_peer` **abstract component**. So far N **specific peers** are spawned.

The spawning operation consists, under the hood, in subscribing dynamically child specifications to the `peers` OTP **supervisor**.

3.3 The communication system

Many theoretical problem concerning distributed systems are based on some assumptions about the environment in which the nodes execute the business logic. In this case we assume that:

- The nodes may be *faulty* (i.e. they can possibly *crash* during the execution of the algorithm);
- The communication channel is reliable (no messages are lost), but the communication may require a possibly unbounded time.

Both assumptions are reflected in the program, with modelizes them by implementing a communication system which has the following characteristics:

- With a certain probability p_{crash} the node may be killed just before the transmission;
- With probability $1 - p_{\text{crash}}$ the transmission will be achieved correctly, although delayed by a random time interval, according to a certain probability distribution.

The idea of embedding the *syntetic crash* into the transmission is justified by the facts that:

- Transmission is basically mandatory for any reasonable non-trivial distributed algorithm;
- During transmission a process can influence the outcome of an algorithm.

Of course this functionality is active only for **peers** sending data to other peers or in broadcast: there's no point in making the **keeper** or one of the internal service crash. Note also that, in case of broadcasting, the failure probability is p_{crash}, as for unicast messages.

3.4 Parameters

Aside from the parameters of the *Consensus Algorithm* (see *Susbection* 4.2), and the strictly technical ones (see *Susbection* 6.3), the YUNA environment has some parameters which are bound to the distributed system model:

Failure probability : as explained each node fails with a certain probability p_{crash} during a message transmission. Initially this was designed as an event concerning only a fixed set of processes, then I decided to make all nodes equally vulnerable to random crashes (this is more realistic).

This choice may be changed in future, but for the moment $p_{\text{crash}} = p_{\text{faulty}} \cdot p_{\text{fc}}$. Of course $p_{\text{faulty}}, p_{\text{fc}} \in [0, 1]$;

Random delivery delay : as mentioned, the communication can be delayed randomly. The parameters required for this functionality are the following:

- The minimum delay δ_{min}, expressed in milliseconds;
- The maximum delay δ_{max}, expressed in milliseconds;
- The probability distribution of the delay D_{delay}, which must be a distribution taking values in the interval $[0, 1]$. For instance, if we take the *uniform distribution* between 0 and 1, we obtain a delay δ uniformly distributed between δ_{min} and δ_{max}.

Number of nodes : to be mentioned here for completeness, despite it's trivial, $n = |\Pi|$.

Beaconing : the period of time between beacons is T_{beacon}, expressed in milliseconds.

As for any other parameter, those can be provided to the application by means of the *configuration file*. See *Susbection 6.3* for further details.

4 Consensus over YUNA

4.1 Organization of the Algorithm

The src/protocols directory is thought to contain the actual distributed protocols written by using the YUNA facility. The project comes with a very simple testing application (src/protocols/test directory) and, of course, with an implementation of the *consensus algorithm* (src/protocols/consensus-gossip-fd directory).

As mentioned in *Section 1*, the implemented algorithm is split into two parts:

- The *Failure Detector*;
- The actual *Consensus Algorithm*.

YUNA provides an **abstract component** for the nodes of a distributed system. Both parts are implemented as **specific component**, since both of them can be modelled with the same event interface; so far each node is an instance of the following structure:

- A proxy object (gfd_peer.erl) is directly interfaced with the **abstract component** logic, and forwards events to the *Failure Detector* (faildet.erl) and to the *Consensus Algorithm* (consensus.erl);
- Messages delivered through the `handle_message/3` callback are required to match the following pattern:
  - `{faildet, Msg}` where `faildet` is an *atom* and `Msg` can be any object;
  - `{cons, Msg}` where `cons` is an *atom* and `Msg` can be any object;

  Message of the first kind are unwrapped and the `Msg` item is forwarded, respectively, to the `faildet:handle_message/3` and the `consensus:handle_message/3` function;

  A snippet of code showing this technique is shown in *Listing 2*.

```
handle_message (From, {faildet, Msg}, Status = #status{ fd=FD }) ->
    case faildet:handle_message(From, Msg, FD) of
        {ok, NewFD} ->
            % We had no errors, update the status with the updated Failure
            % detector;
            NewStatus = Status#status {
                fd=NewFD
            },
            {ok, NewStatus};
        Error ->
            % If we had some error, we propagate it outside the function
            Error
    end;
```

Listing 2: Proxying for the failure detector

- Introductions (`handle_introduction/3`) are always redirected to the *Failure Detector*, since it's in charge of managing the list of known hosts;
- The beacons (`handle_beacon/1`) are also redirected to the *Failure Detector*, since it triggers the deletion of non-responding nodes;
- Each change in the *Failure Detector*, either caused by an introduction, a beacon or by the distributed logic of the *Failure Detector* itself, is reflected to the *Consensus Algorithm* as a virtual message:
    - When new nodes are discovered, the *Consensus Algorithm* is notified through a message (by using the `consensus:handle_message/3` callback), and provided with the list of discovered nodes plus the updated number of alive nodes;
    - When known nodes are *suspected*, the *Consensus Algorithm* is notified through the `consensus:handle_message/3` callback, and provided with the list of suspected nodes plus the updated number of alive nodes (note that this is very important, as the *Consensus Algorithm* must be aware of the current status of the *coordinator*);
    - In both cases the *atom* `faildet` is set as sender of the message.

  A snippet of code showing this technique is shown in *Listing 3*.

4.2 Algorithm Parameters

The *Consensus* algorithm itself would not require parameters, but it uses a *Failure Detector* system, which in the original paper[2] requires three parameters: T_{fail}, T_{cleanup} and T_{gossip}.

All three values are length of time periods, so far in the application they are expressed in milliseconds.

All parameters can be provided to the application by means of the *configuration file*. See *Susbection 6.3* for further details.

4.3 Implementation

As the program interface is *event driven*, the algorithm as presented in the paper has been split and distributed among the callbacks, however it keeps its basic behavior. The algorithm life cycle is the following:

```
handle_beacon (Status = #status{ fd=FD, cons=Cons }) ->
    try
        % The beacon may either update the Failure Detector (if some
        % node turns into suspected) or make it go in a wrong state.
        NewFD =
            case faildet:handle_beacon(FD) of
                {ok, FD0} -> FD0;
                E0 -> throw(E0)             % Terminate immediately.
            end,
        NewCons =
            case faildet:get_last_dead(NewFD) of
                [] ->
                    % In case we don't have dead nodes the consensus
                    % algorithm keeps the current state.
                    Cons;
                Dead ->
                    % Else the algorithm is notified about which nodes are
                    % suspected.
                    NAlive = element(1, faildet:get_neighbors(NewFD)),
                    Msg = {dead, Dead, NAlive},
                    case consensus:handle_message(faildet, Msg, Cons) of
                        {ok, Cons1} -> Cons1;
                        E1 -> throw(E1)
                    end
            end,
        % Updated versions of the internal status are returned
        Status#status {
            fd=NewFD,
            cons=NewCons
        }
    of
        S -> {ok, S}
    catch
        % Any throwed error will be in the form {error, WhatHappened}
        thorw:E -> E
    end.
```

Listing 3: Events from the *Failure Detector*

```
init (IdAssignment) ->
    Tree = lists:foldl(fun ({I,P}, T) -> gb_trees:insert(I, P, T) end,
                       gb_trees:empty(), IdAssignment),
    {ok, #cons{ id_assign = Tree }}.
```

Listing 4: initialization phase

4.3.1 Initialization phase

When the algorithm is initialized (*Listing* 4), it's provided with the *ID* assignment for the next rounds. This value is provided as list of pairs `{Id, Pid}`, where `Id` is a numeric value and `Pid` is a process identifier.

Note that the assignment tells nothing about which process is alive and which is dead. This information comes from the *Failure Detector*.

After the initialization phase, each **node** waits for the startup signal from the **keeper**. This triggers the *startup phase* (described in *Paragraph* 4.3.2).

```
handle_message (keeper, start, Cons = #cons{ phase=0 }) ->
    % Happening only when we are in phase 0.
    run_round(Cons);

...
...

run_round (Cons = #cons{ est=Est }) ->
    % If a node recognizes itself as the coordinator, the estimated decision value
        is broadcasted. If the value has not been estimated yet, it's created
        randomly.
    Self = self(),
    NewEst =
        case get_coordinator(Cons) of
            {_, Self} ->
                % I am the coordinator.
                Est_c =
                    % If we don't have an estimation, take one randomly, else
                        propagate the one we have.
                    case Est of
                        '?' -> random_estimate();
                        _ -> Est
                    end,
                log_serv:log("Coordinator started with est=~p",
                             [Est_c]),
                % Broadcast to all nodes, talk directly with their consensus
                    algorithm
                gfd_api:cons_bcast({est_c, Est_c}),
                % Update value for estimation
                Est_c;
            _ ->
                % Estimation doesn't change.
                Est
        end,
    NewCons = Cons#cons {
        est=NewEst,      % Updated estimation
        phase=1          % We go to phase 1
    },
    {ok, NewCons}.
```

Listing 5: Startup phase

4.3.2 Startup phase

The startup phase behavior (*Listing* 5) depends on the role of the node. If the node is a coordinator, then the value to be decided is chosen randomly, otherwise we directly skip to the *estimation phase* (described in *Paragraph* 4.3.3).

4.3.3 Estimation phase

The estimation phase (*Listing* 6) consists in listening for events. Two kind of events are significative: the suspicion (from the *Failure Detector*) of *coordinator death*, or the obtained message indicating an estimation. This message may be propagated by the coordinator itself or by any neighbor which obtained it transitively.

At this point the node may have an estimation of the value or it may not know any estimated value v. In both cases the node broadcast a message in the form `{phase2, E}`, where `E` may be the value v or the *atom* `'?'`. After this, the algorithms moves to the *agreement phase* (described in *Paragraph* 4.3.4).

```
handle_message (_From, {est_c, Est_c}, Cons = #cons{ phase=1 }) ->
    % The massage from current coordinator (either direct or propagated by other
        nodes) moves us to phase 2
    log_serv:log("Got estimation: ~p", [Est_c]),
    run_phase2(Cons#cons{ est_from_c=Est_c });

handle_message (faildet, {dead, DeadList, NAlive}, Cons = #cons{}) ->
    % When the Failure-Detector signals some dead node, the number of alive peers
        must be updated
    NewCons = Cons#cons{ nalive=NAlive },
    case {NewCons#cons.phase, is_coordinator_dead(NewCons, DeadList)} of
        {1, true} ->
            % We are in phase 1 and the coordinator is dead. Go to phase 2 and set
                the estimate to '?'.
            log_serv:log("Coordinator is dead"),
            run_phase2(NewCons#cons{ est_from_c='?' });
        _ ->
            {ok, NewCons}
    end;

...
...

run_phase2 (Cons = #cons{ est_from_c=E }) ->
    gfd_api:cons_bcast({phase2, E}),
    NewCons = Cons#cons {
        phase=2,    % Entering phase 2
        rec=ordsets:new(),
        prop=gb_sets:new()
    },
    {ok, NewCons}.
```

Listing 6: Estimation phase

4.3.4 Agreement phase

This phase (shown in *Listing* 7), requires at least $\lfloor n/2 \rfloor + 1$ different nodes to be producing, as last part of the *estimation phase*, the same proposed value. Such a value is read from a message in the form `{phase2, E}` where either `E` = v or `E` = `'?'`.

If all the received message have the same value, the *consensus* is achieved correctly.

```
handle_message (From, {phase2, E},
                Cons = #cons{ phase=Phase, rec=Rec, prop=Prop }) ->
    case Phase of
        2 ->
            NewCons0 = Cons#cons {
                rec=ordsets:add_element(E, Rec),
                prop=gb_sets:add_element(From, Prop)
            },
            agreement(NewCons0);
        _ ->
            {ok, Cons}
    end;

...
...

agreement (Cons=#cons{ phase=2, rec=Rec, prop=Prop, id_assign=Ids }) ->
    % The agreement phase: check whether we reached the target number of
    % proposal, decide if this is the case and all the nodes required
    % selected the value.
    N = gb_trees:size(Ids),
    Target = trunc(N/2) + 1,
    case gb_sets:size(Prop) of
        M when M < Target ->
            {ok, Cons};
        M ->
            log_serv:log("Heard enough peers: ~p/~p; deciding", [M, N]),
            case Rec of
                ['?'] ->
                    log_serv:log("Next round"),
                    run_next_round(Cons);
                [V] when V =/= '?' ->
                    % Decide
                    gfd_api:cons_decide(V),
                    {ok, Cons#cons{ decided=true }};
                ['?', V] ->
                    log_serv:log("Next round with value"),
                    run_next_round(Cons#cons{ est=V });
                _ ->
                    log_serv:log("Y U No Agree? Rec=~p", [Rec]),
                    {error, yuna}
            end
    end.
```

Listing 7: Agreement Phase

5 Experiments and Stats

5.1 Tutorial: launching the application

Conventionally, an ERLANG **application** has a target directory in which binary object (.beam) are pushed during compilation. This directory is conventionally named ebin, and the interpreter must be aware of this position. So far, in order to launch the application, use the following command line:

```
$ erl -pa ebin
Erlang R14B03 (erts-5.8.4) [source] [64-bit] [smp:2:2] [rq:2]
[async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.8.4  (abort with ^G)
1>
```

Before launching the application it may be necessary to re-compile the modules:

```
1> make:all([load]).
...
up_to_date
```

Then the application can be launched effectively:

```
2> application:start(yuna).
Loading configuration...
Starting YUNA...
Starting main supervisor...
Starting peers pool...
Starting services...
ok
2012/1/4 3:30:59 <0.98.0>  Nodes started: N=500
2012/1/4 3:30:59 <0.98.0>  Preparing protocol...
2012/1/4 3:30:59 <0.98.0>  Nodes started. Letting them know each other...
...
```

After some time (depending on the execution) the process will stop:

```
...
2012/1/4 3:32:4 <0.98.0>  Consensus (should) has been reached:
2012/1/4 3:32:4 <0.98.0>    Value false has been selected by 496 nodes
2012/1/4 3:32:4 <0.98.0>  Killing remaining processes...
2012/1/4 3:32:4 <0.98.0>  Terminating.
2012/1/4 3:32:4 <0.98.0>  Keeper finished.
```

This will produce the files containing the statistics. The name of the file will depend, as explained in *Paragraph* 6.3.2, by a prefix stored in the configuration variable. In this case (the prefix is *statfile* we have the following files:

```
$ ls statfile_*
statfile_decision_count.log  statfile_est_node_count.log statfile_events.log
    statfile_node_count.log
```

From this files we can produce easily the graph of the execution by feeding the *gnuplot* software with the prefix and the configuration I've put in the priv directory:

```
$ gnuplot -e 'prefix="statfile"' priv/plot_stats.gnuplot
```

This will produce a *pdf* file named after the prefix, in this case statfile.pdf. *Figure* 4 shows an example of graph.

5.2 Experiment: ambush to the coordinator

For testing purposes I patched the program in order to allow an external process (ideally the ERLANG shell) to tamper the system before the protocol runs.

I added an external boolean parameter, `start_direct`, which can be setted to **false** obtaining the system to wait for user interaction *before* starting the protocol.

I also added a few calls to the `gfd_keeper` component (which is the **specific keeper** for the process):

- `persist/1`
  If called with parameter **false** (disable *persistence*), the **keeper** is required to abort the protocol the number of failing nodes f overcomes $n/2$;
- `launch_consensus/0`
  Launch the protocol if it's not running yet (not running if `start_direct` = **false**);
- `schedule_killing/1`
  Allows the external project to declare a list of identifiers to be killed. For instance `schedule_killing([1])` will kill the coordinator just before starting the protocol.

I wrote a small file which calls those functions. The file is directly passed (as script) to an instance of the interpreter and run in batch mode, so that the following sequence gets executed:

1. Start the application;
2. Tamper the execution by removing the *persistence* of the **keeper**;
3. Start the protocol;
4. Put an ERLANG *monitor* on the **keeper** process (in order to get informed when the keeper terminates);
5. Send a synchronous message to the **logging server** (in order to build a *causality relation* and ensure all event messages are served before retrieving data).

At this point I wrote a little *perl* script (src/priv/probtest.pl) which runs the application with different (incremental) values for the p_{crash} parameter. For each probability value the script runs a certain number of experiments. Then the same set of experiments is ran with a small modification in the batch file, scheduling the crash of the *coordinator* before starting the protocol.

The script retrieves the events (for instance the protocol startup and stop time) at the end of each experiment. The collected data is so far crunched and displayed on a picture by *gnuplot*.

Figure 2 shows graphically the outcome of the described experiment: I ran it starting from a probability $p_{\text{crash}} = 0$ (perfect run, no crashes), incrementing p_{crash} of 0.005 at each experiment. I also defined the stop condition the conjunction of $p_{\text{crash}} < 1$ and *less than 5 consecutive fails*, so that the failure probability remains in a reasonable interval.

Apart from probability, which is changing, the remaining configuration for this experiment has been the following:

```
beaconwait 120
deliver_dist {random, uniform, []}
deliver_maxdel 1500
deliver_mindel 500
file_prefix statfile_default
keeper gfd_keeper
```

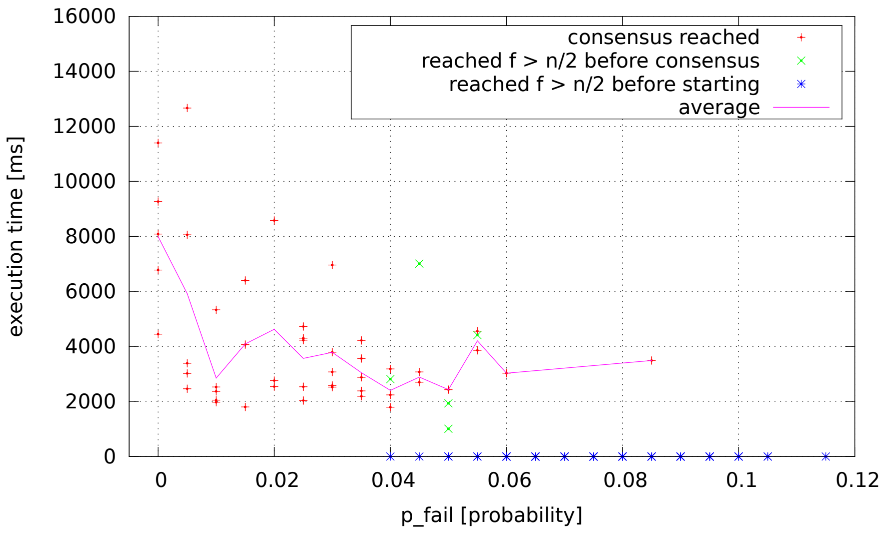

Figure 2: *Termination time for experiments. Experiments are run with incremental per-transmission failure probability (x axis). As probability grows, the number of faulty nodes f reaches $n/2$, thus the protocol is aborted. The pink line shows the average success time for each probability value. See* Paragraph 5.2.1 *for further details.*

```
npeers 500
statpeers 0.01
tbeacon 500
tcleanup 20
tfail 10
tgossip 4
```

5.2.1 Considerations

In a predictable manner, the experiments highlight a very strong susceptibility to failure, even for small values of p_{crash}: each process, in fact, follows a *geometric distribution*

$$\mathcal{P}\left[\text{Crash at } k\text{-th transmission}\right] = p_{\text{crash}} \cdot (1 - p_{\text{crash}})^{k-1}$$

$$p_{\text{k}} = \mathcal{P}\left[\text{Being crashed after } k \text{ transmissions}\right] = \sum_{i=1}^{k} \left(p_{\text{crash}} \cdot (1 - p_{\text{crash}})^{i-1} \right)$$

and this distribution grows pretty quickly, which justifies the fact that executions stop being meaningful for $p_{\text{crash}} > 0.8$. In order to determine the expected number of alive processes in time, a more in-depth analysis should be achieved.

Figure 3 shows the graph deriving from the described experiment; *Figure* 6 reports some statistics about the executions; *Figure* 4 and *Figure* 5 show respectively a normal and a tampered execution.

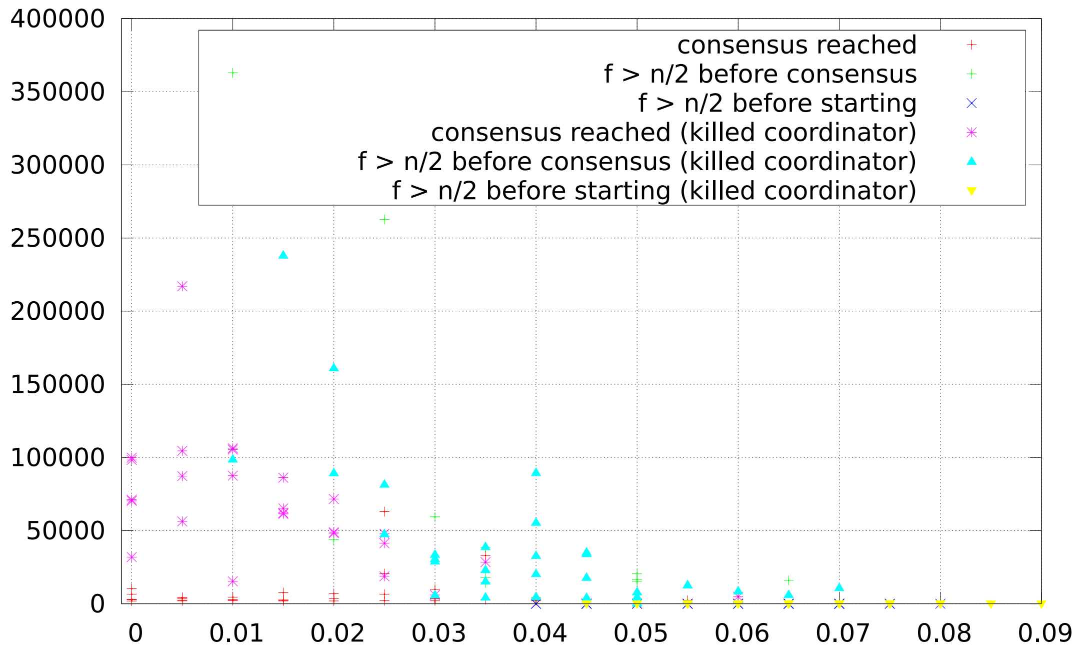

Figure 3: *Termination time for experiments. Different behaviors when the coordinator has been killed before value proposal. The points labelled* consensus reached *can be seen in detail in* Figure 2.

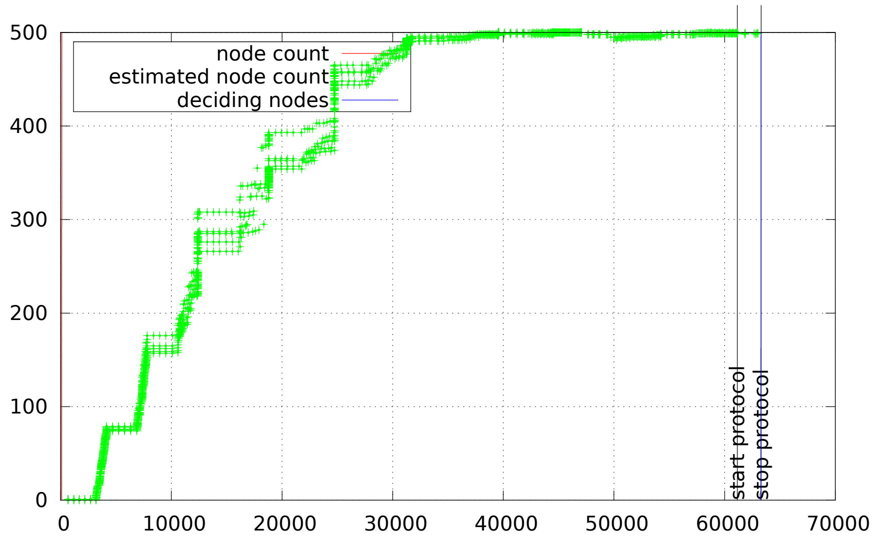

Figure 4: *A run of the* consensus *algorithm with the default settings. The green points (labelled* est_node_count*) show the estimated number of nodes as provided by the* Failure Detector *of the nodes enabled for statistical retrieval. The actual number of nodes is 500 (no nodes crashed during the experiment). The* start protocol *and the* end protocol *labels show respectively the startup instant of the* consensus *protocol and the instant in which the nodes achieved the agreement.*

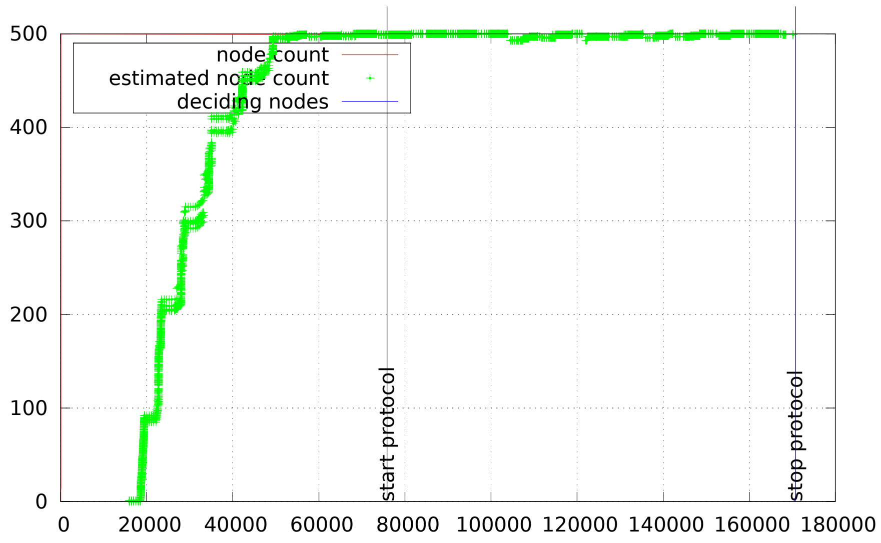

Figure 5: *A run of the* tampered consensus *algorithm with the default settings. The experiment follows the same characteristics of the one showed in* Figure 4*, except the coordinator is killed just before the protocol starts. We can appreciate the difference in termination time.*

Probability	Avg [ms]	Max [ms]
0	4918	10285
0.005	3261	4274
0.01	3032	4488
0.015	3394	7572
0.02	4112	6920
0.025	23031	62993
0.03	4723	9782
0.035	17863	32955
0.04	2327	2600
0.055	2385	2385

(a) Protocol termination time for normal consensus

Probability	Avg [ms]	Max [ms]
0	74305	99863
0.005	116259	216993
0.01	78656	106305
0.015	68723	86209
0.02	56247	71610
0.025	35976	47802
0.03	5978	5978

(b) Protocol termination time for tampered consensus

Figure 6: *Comparison between normal runs of the* Consensus *algorithm and runs in which the coordinator in charge has been killed before value proposal.*

6 YUNA ***API*** reference

Before reading this, please read *Section* 2. *Paragraph* 2.2.1 gives order to get an overview on coding conventions: the last parameter of all *events* is the *status* of the running logic.

6.1 The keeper *API*

6.1.1 Events and callbacks

- `init/1`
  This function is provided with an extern parameter, corresponding to the `keeper_args` voice in the configuration file (see *Susbection 6.3* for further details). It is supposed to do some initialization (for instance spawning peers using the `keeper_proto:add_peers/3`). It can return `{ok, Status}` in case of success or `{error, Reason}` in case of failure;

- `handle_spawn_notify/2`
  A freshly peer can call the `peer_ctrl:notify_spawn/0` function. This will result in this event to be issued. The first parameter is the **Pid** of the notifying process;

- `handle_result_notify/3`
  A peer can call the `peer_ctrl:notify_result/1` function in order to send a piece of elaborated information to the **keeper**. This will result in this event to b e issued. The first parameter is the *pid* of the peer, the second is the result;

- `handle_term_notify/4`
  When a peer dies (both for normal termination or crash) the **keeper** gets notified trough this event. The first parameter is the *pid* of the peer, the second may be the **undefined** atom or a ERLANG *reference* associated to the spawning (see ERLANG documentation for further details), the third is the reason of the termination;

- `handle_info/2`
  If the **abstract component** gets some raw message, not appropriately enveloped with the underlaying protocol, the **keeper** gets notified through this event. The first parameter is, of course, the raw packet.

6.1.2 Library functions

1. `keeper_inject`

   - `send/2`
     Using this function, the keeper can send a message to a specific node (first parameter), triggering its `handle_introduction/3` callback. The message (second parameter) is supposed to be understood by the **specific peer**, which will see the *atom* `keeper` as sender;

   - `send/3`
     This function is the same as `send/2`, except it has and additional parameter (the first one) which specifies a sender for the message. This can be used, for instance, to spoof a *pid*;

   - `introduce/2`
     This function triggers the `handle_introduction/3` callback of the peer which *pid* is specified as first parameter. The peer will be introduced to the peer specified as second parameter. The sender of the introduction message will be the *atom* `keeper`;

   - `introduce/3`
     This function is the same as `introduce/2`, except it has and additional parameter (the first one) which specifies a sender for the introduction message. This can be used, for instance, to spoof a *pid*;

   - `bcast/1`
     This function achieves broadcasting of a message. It's pretty similar to `send/2`, except of course it doesn't require a target *pid*;

- `bcast/2`
  This function achieves broadcasting of a message. It's pretty similar to `send/3`, except of course it doesn't require a target *pid*;

2. `keeper_proto`

   - `add_peers/3`
     Insert new nodes in the `peers` **supervisor**. The parameters are the number N of nodes to spawn, an *atom* giving the name of the **specific component** which implements the peer logic, and finally the parameter for the logic (which will be passed as first parameter for the `init/1` function of the specified module).
     Note that this function can be called multiple times, adding new nodes to the running protocol. The nodes can be added at any time, and may possibly correspond to different **specific components**.
     The function yields a list of tuples `{Pid, Ref}`, where `Pid` is the process identifier of the spawned process, and `Ref` is the reference of an ERLANG *monitor*, which can be used to match the second parameter of the `handle_term_nofity/4` event;
   - `add_peers_pidonly/3`
     Same as `add_peers/3`, just returns a list of process identifiers, discarding *references*;
   - `enable_beacon/1`
     Require the `bcast` component to send periodically a beacon to the subscribers. The parameter to be provided is the beacon period (in milliseconds);
   - `disable_beacon/0`
     Disable the beacon enabled with `enable_beacon/0`.

6.2 The peer *API*

6.2.1 Events and callbacks

- `init/1` This event is provided with an extern parameter, corresponding to the last parameter of the `keeper_proto:add_peers/3` function. It is supposed to do some initialization dependent on the **specific component** implementing the peer. It can return `{ok, Status}` in case of success or `{error, Reason}` in case of failure;

- `handle_message/3`
  This event is called when the **generic component** is messaged with a valid message. The first parameter corresponds to the process identifier of the sender or to the *atom* `keeper` if the message has been send by the **keeper** through the `keeper_inject:send/2`

- `handle_introduction/3`
  This event is triggered when the **peer** has been introduced to another **peer**. As this introduction can be done both by the **keeper** or by another **peer**, the first argument can be either the `keeper` *atom* or a *pid*.

- `handle_beacon/1`
  If the **keeper** enables the beacon (trough the `keeper_proto:enable_beacon/1` function), this event will be called periodically.

- `handle_info/2`
  If the **abstract component** gets some raw message, not appropriately enveloped with the underlaying protocol, the **peer** gets notified through this event. The first parameter is, of course, the raw packet.

6.2.2 Library functions

1. `peer_chan`

   - `send/2`
     Send a message through the internal protocol. The first parameter is the *pid* of the target process, the second is the message;
   - `greet/1`
     Greet a **peer** by raising its `handle_introduction/3` event. The first parameter is the *pid* of the target process;
   - `bcast_send/1`
     Broadcast a message through the internal protocol. The parameter is the message to broadcast;
   - `bcast_greet/0`
     Greet all **peers**, in broadcast, raising their `handle_introduction/3` event.

2. `peer_ctrl`

   - `notify_spawn/0`
     Notify the **keeper** about a successfull spawning (see the `handle_spawn_notify/2` event)
   - `notify_result/1`
     Notify the **keeper** about a result (see the `handle_result_notify/3`);
   - `notify_term/1`
     Notify the **keeper** about process termination. Note that, as this function is used internally already, there's no need to use it explicitly (see the `handle_term_notify/4`);
   - `notify_term/0`
     As `notify_term(normal)`.

6.3 The configuration file

Basing on the ERLANG coding standards, the configuration of the application is placed in a file named ebin/A.app, where A is the name of the application. In our case the file is ebin/yuna.app, shown in *Listing 8*.

The first chunk of the file describes the application details (giving the name of the application, the version number and so on), while the remaining part of the file contains the environment definition as *key-value* mapping. The accepted format matches the following pattern:

```
{env, [{K1, V1}, {K2, V2}... {Kn, Vn}]}
```

The configuration in the .app file can be overriden by another configuration in a .conf file by passing it as parameter to the ERLANG interpreter: suppose our file is called override.conf, we can do the following:

```
erl -conf override
```

All this details are better explained in the official ERLANG documentation[4]. Since however this is pretty boring, I've decided to write a little *perl* script which is capable of sintetizing a correct configuration file starting from a simple text file containing the parameters. See *Paragraph 6.3.5* for further details.

```
{application, yuna, [
    {description, "YUNA - Distributed Consensus implementation"},
    {vsn, "0.0.1"},
    {modules, [chan_filters, tweaked_chan, bcast, peers, keeper_proto,
               keeper_inject, yuna, gen_peer, gen_keeper, peer_chan,
               peer_ctrl, main, log_serv, randel, utils, services]}
    {registered, []},
    {application, [kernel, stdlib]},
    {mod, {yuna, []}},

    {env, [
        % Default configuration, can be overriden by configuration file.
        {faulty_prob, 0.01},         % 1% of nodes are faulty
        {faulty_fail_prob, 0.05},    % 5% crash probab. for faulty node

        {deliver_mindel, 500},       % Minimum deliver delay
        {deliver_maxdel, 1500},      % Maximum deliver delay
        {deliver_dist, {random, uniform, []}},  % Delay distribution

        {log_args, { standard_error,     % For visual logging
                     "statfile"          % For stats retrival
                   }
        },
        {keeper, gfd_keeper},
        {keeper_args, { {10, 20, 4},      % TFail, TCleanup, TGossip
                        500,             % NPeers
                        true,            % Don't wait for user modifications
                        0.01,            % StatPeers [ratio]
                        500,             % TBeacon, [ms]
                        120              % BeaconWait, [TBeacon]
                        % 120 beacons = 1 min
                      }
        }
    ]}
]}.
```

Listing 8: default configuration file

6.3.1 Parameters for the YUNA environment

(see *Susbection* 3.4 for further details)

- The probability of having a faulty node p_{faulty} corresponds to the `faulty_prob` key;
- The probability, for a faulty node, of fail during a transmission p_{fc} corresponds to the `faulty_fail_prob`.
- The minimum delivery delay δ_{min} corresponds to the `deliver_mindel` key;
- The maximum delivery delay δ_{max} corresponds to the `deliver_maxdel` key;
- The probability distribution of the delay D_{delay} is specified as an ERLANG *tuple* `{M, F, A}` providing the coordinates (*module, function name* and *arguments*) of the function. A uniform distribution can be obtained using `{random, uniform, []}`. This parameter corresponds to the `deliver_dist` key.
- Due to the fact that the **keeper** is in charge of spawning the **peers**, the number of nodes $n = |\Pi|$ is part of the **keeper** configuration, thus described in *Paragraph* 6.3.4;
- The same is valid for the *beacon*, thus T_{beacon} is part of the **keeper** configuration, described in *Paragraph* 6.3.4 as well.

6.3.2 Technical parameters

- The only one parameter for the logger is associated to the key `log_args`. It must be a tuple `{Stream, NamePrefix}`, where `Stream` is a file descriptor (like `standard_error`, which is the default), and `NamePrefix` is a string giving the prefix for the logger output file names (see *Susbection* 3.1 for further details);
- The **keeper** parametrization and selection goes trough two keys:
  - The `keeper` key must be associated with an *atom* declaring the name of the **specific componet** to be used as **keeper**;
  - The `keeper_args` key can be associated to any ERLANG item, which will be the actual parameter for the `init/1` function of the **specific keeper**.
- We need to tune the system and get the better trade-off between statistics significance and overhead. `StatPeers` is the ratio $r \in [0, 1]$ of processes which will report statistics. It's par of the **keeper** configuration, thus described in *Paragraph* 6.3.4.

6.3.3 Parameters for Consensus

All the parameters for the distributed system nodes (see *Susbection* 4.2 for further details) are managed by the **keeper** during the spawning phase. *Susbection* 4.2 describes the configurations for T_{fail}, T_{cleanup} and T_{gossip}.

6.3.4 Keeper configuration

The **specific keeper** for the implemented *Consensus* algorithm accepts as parameter a *tuple* of the form

```
{PeerParams, NPeers, StatPeers, TBeacon, BeaconWait}
```

where

- `PeerParams` contains the parameters for the **peers**
- `NPeers` is the number n of processes running the algorithm;
- `TBeacon` is the beacon time period T_{beacon}.

The `PeerParams` is required to be a tuple of the form

```
{TFail, TCleanup, TGossip}
```

respectively corresponding to T_{fail}, T_{cleanup} and T_{gossip}.

6.3.5 Automatic configuration building

Since building a configuration may be boring, I've written a little program that does it automatically. The usage is trivial:

1. Build a skeleton for the configuration (example in *Listing* 9);
2. Crunch and get the configuration for ERLANG (example in *Listing* 10).

```
$ priv/buildconf.pl -skel quickconf

$ more quickconf
faulty_prob
beaconwait
tfail
tcleanup
tgossip
npeers
deliver_dist
deliver_mindel
file_prefix
deliver_maxdel
faulty_fail_prob
keeper
tbeacon
statpeers
```

Listing 9: Generating a quick configuration skeleton

7 Bibliography

References

[1] Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach (*Mostefaoui and Raynal, 1999*)

[2] A Gossip-Style Failure Detection Service (*Robbert van Renesse, Yaron Minsky, and Mark Hayden*)

[3] On the quality of service of failure detectors (*W. Chen, S. Toueg, and M. Aguilera*)

[4] `http://www.erlang.org/doc/design_principles/applications.html`

```
$ more configs/override

beaconwait 120
deliver_dist {random, uniform, []}
deliver_maxdel 1000
deliver_mindel 100
faulty_fail_prob 0.05
faulty_prob 0.01
file_prefix statfile_default
keeper gfd_keeper
npeers 500
statpeers 0.01
tbeacon 500
tcleanup 20
tfail 10
tgossip 4

$ priv/buildconf.pl < configs/override > override.conf

$ more override.conf
[{ yuna, [
    {faulty_prob, 0.01},
    {faulty_fail_prob, 0.05},
    {deliver_mindel, 500},
    {deliver_maxdel, 1500},
    {deliver_dist, {random, uniform, []}},
    {log_args, { standard_error,
                 "statfile_default" }
    },
    {keeper, gfd_keeper},
    {keeper_args, {{10, 20, 4},
               500,
               0.01,
               500,
               120}
    }
]}].

$ erl -pa ebin -conf override
Erlang R14B03 (erts-5.8.4) [source] [64-bit] [smp:2:2] [rq:2]
[async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.8.4  (abort with ^G)
1> application:load(yuna).
ok
2> application:get_env(yuna, deliver_mindel).
{ok,100}
3> application:get_env(yuna, deliver_maxdel).
{ok,1000}
```

Listing 10: From quick configuration to ERLANG .conf file