1 DDMon Implementation overview

DDMon is implemented mainly in Erlang and is built using Mix. The overall structure is rather standard for Mix projects:

- `mix.exs` — project configuration
- `src/` — Erlang code
- `lib/` — Elixir code

1.1 Algorithm for distributed deadlock detection monitoring

Files of concern:

- `src/ddmon.erl` — implementation of the monitoring algorithm described in Section 5 of the companion paper
- `src/ddmon.hrl` — common macros
- `src/gen_monitored.erl` — auxiliary layer between our deadlock monitors and `gen_server` services

1.1.1 Startup

To be launched as a monitored service, a `gen_server` written in Erlang or Elixir must satisfy two requirements:

- It must call `ddmon:start` instead of `gen_server:start` (or `start_link`, respectively); and
- It must use the `ddmon:call/2,3` functions to perform calls to other monitored services.

This is necessary because the monitor needs to "wrap" the `gen_server` and intercept its calls, as required by the semantics of monitored services in *Section 4* of the companion paper.

(In the case of `gen_server`s written in Elixir, both requirements are satisfied by defining a simple alias, as discussed in the file EXAMPLE.md).

The following steps happen when a `gen_server` is started via `ddmon:start`:

- First, a `ddmon` monitor is started. If a process name has been provided, the monitor will use it.
- During `ddmon` process initialisation, the original `gen_server` is started with the provided parameters (except the name). The `gen_server` process is linked to the monitor, and its PID is never revealed outside the monitor. As a result, the original `gen_server` process is "wrapped" by the monitor.
- The PID of the monitor (not the original `gen_server`'s) is returned to the caller of `ddmon:start`.

1.1.2 Operation of a monitored service

The wrapped generic server is started through the `gen_monitored` module which operates exactly like `gen_server`, except it sets a couple variables in the process dictionary. This step not strictly needed, but is useful for integration with unmonitored components of the system and makes examples simpler.

1.1.3 Operation of the monitor

The monitor is implemented as a generic state machine (`gen_statem` behaviour) with three possible states:

- `unlocked` — when the monitor believes that the overseen service is not locked on any other service;
- `locked` — when the monitor believes that the overseen service is awaiting a response from another service;
- `deadlocked` — when the monitor detects and reports a deadlock.

The state machine reacts to incoming calls mimicking the `gen_server` interface of the overseen service. Such calls are forwarded to the monitored service as non-blocking `gen_server` calls (via `gen_server:send_request`). The monitor then waits for either a response or external call requests from the service, while handling other incoming calls and probes. Probes are communicated via `cast` messages. All unrecognised communications are forwarded directly.

1.1.3.1 Implementation to the semantics of monitored services This section connects our monitor implementation to the theory of monitor instrumentation in *Section 4* of the companion paper — specifically, the semantic rules in *Figure 9*. The following outline refers to the state callbacks (`unlocked`, `locked`, `deadlocked`) implemented in `src/ddmon.erl`.

- Rules `MON-I`, `MON-TO` and `MON-MI` are part of the Erlang runtime system as receiving messages (`call` and probe `cast` respectively).
- Rule `MON-TI` specifies how the monitor reacts to incoming messages:

  - If the message is an incoming query, then it is handled as `{call, From}` where `From` points to a foreign service (i.e. not the one supervised by the monitor).
  - If the message is a response, then it appears as `info` ("miscellaneous message"). If the monitor is locked, it compares it to the current request id to distinguish it from other messages.

- Rule `MON-O` triggers when an outgoing message is forwarded. In `ddmon`, this is handled in two possible ways:

  - If the message is a response, then it is handled as an `info` callback. The monitor keeps track of currently processed requests and is able to recognise outgoing responses using `gen_statem:check_response`.
  - Outgoing queries are pre-processed by `ddmon:call` to include information about the intended recipient (which is needed since the actual recipient is the monitor). The first parameter of the handler callback is of shape `{call, {Worker, PTag}}`, and the payload matches `{Msg, Server}`, where:
    - `Worker` is the PID of the monitored service
    - `PTag` is the unique reference of the call used to recognise a matching response. It is also used as the active probe until a response is received.
    - `Msg` is the original message to be forwarded
    - `Server` is the intended recipient of the call

- Rule `MON-TMI` describes how a probe is handled. In `ddmon`, probes are implemented as `cast` messages of the form `{?PROBE, Probe, Chain}`, where;

  - `?PROBE` is a constant Erlang atom defined in `src/ddmon.hrl`.
  - `Probe` is a unique `gen_server` call reference created when the initiator of the probe sent its request.
  - `Chain` is the list of nodes visited by the probe during the propagation. When a deadlock is reported, this list shows a minimal deadlocked set. This component is not part of the theory, but is mentioned in the paper in *Section 7.1.*

- In rules `MON_TI`, `MON-O` and `MON-TMI` a sequence of messages yielded by the algorithm is prepended to the monitor queue; in our algorithm (See *Section 5*), this sequence of messages can only contain outgoing probes which are scheduled to be sent before any other message is handled by the monitor. Therefore, in the implementation, `ddmon` simply sends such probes sequentially.

A practical difference w.r.t. the theory in the paper is that the DDMon monitor implementation distinguishes `gen_server` calls forwarded by other monitors from calls coming from unmonitored processes: this is achieved by wrapping each message `Msg` sent by a monitored service with `{?MONITORED_CALL, Msg}`, where `?MONITORED_CALL` is a constant Erlang atom defined in `src/ddmon.hrl`. This way, a monitored system can be deployed to serve `gen_server` calls coming from "external" (and unmonitored) clients; the "external" clients will never receive any monitoring probe — but if one of their calls leads to a deadlock, then the deadlock will be detected by DDMon.

1.1.3.2 Implementation of the deadlock monitoring algorithm

In the companion paper (*Section 5*) we formalise the monitor state as a record with three fields. These fields correspond to the `ddmon` implementation as follows. Note that the monitor implementation's current state arises from a combination of the state tracked by the `gen_statem` behaviour (`unlocked`, `locked`, `deadlocked`), plus the contents of a `state` record (referred to as "data" in the `gen_statem` documentation).

Paper	DDMon implementation
`probe` defined	Monitor state machine is in the `locked` state + field `req_tag` in `state` record
`probe` undefined	Monitor state machine is in the `unlocked` state
`waiting`	Field `waitees` in `state` record
`alarm`	Whether state machine is in `deadlocked` state

We use `gen_server:reply_tag()` to implement probes. These tags are uniquely generated by `gen_server` internals for each call and thus address our requirements (i.e., probes must be unique). Therefore, the `freshProbe` function is not explicitly defined, as the necessary unique probe is obtained directly from the outgoing query.

To implement the `waiting` list, we use the collection of request ids provided by `gen_server` (`gen_server:request_id_collection()`). This lets us conveniently manage the list of waiting services and provides us with a handle to properly forward responses.

The deadlock detection monitoring rules in *Figure 12* in the companion paper are implemented by the state callbacks of `ddmon`.

1.1.3.2.1 Deadlock reporting

Whenever a DDMon monitor detects a deadlock, it begins a *deadlock reporting* phase. In the companion paper, this phase is modelled as setting the `alarm` field of the monitor state to `true`. In DDMon, instead, monitors engage in additional communications to propagate the deadlock alarm. These additional communications are not formalised in the paper, as they are an implementation detail for deadlock reporting and they do not influence deadlock detection.

More precisely, when a DDMon monitor detects a deadlock, the monitor sends deadlock notifications in a similar manner to probes — except that such deadlock notifications are sent as `gen_server` responses of the form `{?DEADLOCK, DL}` where `?DEADLOCK` is a constant defined in `ddmon.hrl` and `DL` is the set of PIDs forming the deadlock cycle. Such deadlock notifications never reach the services overseen by the monitors: only monitors see and propagate such notifications.

This deadlock notification mechanism allows an unmonitored Erlang process to be optionally notified whenever one of its `gen_server` calls results in a deadlock. This feature is used by our testing and example scripts to print deadlock reports on the console. (See e.g. EXAMPLE.md and example-system/lib/microchip_factory.ex, where the optional `:monitored` parameter enables the reception and visualisation of deadlock notifications.)

1.2 Testing scenarios

The following modules are related to benchmarking and testing described in *Section 7* of the companion paper, and the test scenario DSL documented in SCENARIOS.md. The following files implement utilities such as the scenario DSL, the visualisation of logs, supervision, and examples.

- `src/ansi_color.erl` — ANSI coloring and formatting.
- `src/tracer` — Erlang debugging process that inspects events in a simulated network.
- `src/scenario` — Entry point for benchmarks and tests: parses scenario files, applies preprocessing, runs experiments and handles results.
- `src/logging.erl` — Visualisation of logs produced by the tracer.
- `src/scenario_gen.erl` — Generator of large scenarios from predefined templates. Used in large benchmarks.
- `lib/test_server.ex` — Generic server evaluating the scenario DSL.
- `lib/main.ex` — Command line interface for running scenarios.