

# 1 Using DDMon to monitor a gen\_server-based application

DDMon can monitor applications consisting of processes (written in Erlang or Elixir) based on the generic server (`gen_server`) behaviour. Intuitively, DDMon acts as a drop-in replacement for the `gen_server` module of the OTP standard library. At this stage, DDMon supports only the most commonly used features of generic servers, i.e. the `call` and `cast` callbacks. Timeouts, deferred responses (`no_reply`) and pooled calls through `reqids` are not covered by the prototype yet.

To instrument an Erlang or Elixir program with DDMon monitors, you'll need to follow these instructions, which depend on the language used to write each `gen_server` instance:

- In the case of `gen_servers` implemented in Elixir, it suffices to add the following line at the top of the file, immediately after `use GenServer`:

```
alias :ddmon, as: GenServer
```

- In the case of `gen_servers` written in Erlang, you will need to find-and-replace all references to the `gen_server` module with `ddmon`. (This is necessary because Erlang lacks the `alias` directive provided by Elixir.)

## 1.1 Example: monitoring a microchip factory application

We now present an example showing how DDMon can monitor a distributed application. We encourage the users to try this simple example, as it serves as a starting point for further experiments.

The example is an Elixir application available the `example-system` directory (see its README). The application implements a simple distributed system where Producers are requested to construct and return objects representing microchips; before responding to such requests, each Producer asks an Inspector to validate the microchip. Notably, Producers may call other Producers in order to construct and return a requested microchip; moreover, when a Producer asks an Inspector to validate a microchip, the Inspector invokes a Producer to perform the validation.

We implement two variations of this example: a small version (with just 2 producers and inspectors) and a larger version (with a much larger number of producers).

### 1.1.1 Small version

This case includes two Producers (with ids 1 and 2) and two Inspectors (with ids 1 and 2 as well). Inspectors 1 and 2 validate values produced by Producers 1 and 2 respectively by comparing them to metadata provided by Producers 2 and 1 respectively (note that the numerical ids are flipped).

There are two scenarios which may nondeterministically occur when this small example application runs, depending on how the calls and responses between `gen_servers` are scheduled: the run may complete successfully, or it may deadlock.

- **Execution without deadlock:**

1. Producer 1 receives a call
2. Producer 1 computes a result asks Inspector 1 for audit
3. Inspector 1 asks Producer 2 for metadata
4. Producer 2 replies to Inspector 1
5. Inspector 1 replies to Producer 1 with its audit
6. Producer 1 replies to the caller
7. Producer 2 receives a call
8. ... (Same as above but flip 1 and 2)

After the steps above, both calls emitted by the Producers are successfully completed, as they both receive a response. When this happens, the application will end with a “**Success**” message.

- **Execution with a deadlock:**

1. Producers 1 and 2 receive calls
2. Producer 1 asks Inspector 1 for audit
3. Producer 2 asks Inspector 2 for audit
4. Inspector 1 asks Producer 2 for metadata
5. Inspector 2 asks Producer 1 for metadata

Now, Producer 1 cannot reply to Inspector 2, because Producer 1 is waiting for a reply from Inspector 1. Similarly, Producer 2 cannot reply to Inspector 1, because Producer 2 is waiting for a reply from Inspector 2. Therefore, a deadlock has occurred. When this happens, the application will end with a “**Timeout**” message.

To execute the small example application, run the following command in the `ddmon` root directory:

```
docker run --rm ddmon bash -c 'cd example-system; mix run -e "MicrochipFactory.start_two"'
```

If both Producers' calls receive a response, you should see a green **Success** message. If a deadlock occurs, you should see a yellow **Timeout** message. You should **repeat the command above multiple times** to observe both possible outcomes.

**NOTE:** at this stage the application is *not* monitored yet, and therefore, the deadlock is not detected: the timeout is the only hint towards diagnosing the problem.

### 1.1.2 Larger version

This case illustrates a larger setup with 93 Producers and 3 Inspectors. Here, deadlocks may involve different services across different executions.

To execute the larger example application, run the following command in the `ddmon` root directory:

```
docker run --rm ddmon bash -c 'cd example-system; mix run -e "MicrochipFactory.start_many"'
```

You should **repeat the command above multiple times** to observe a variety of successful executions (terminating with "Success") and deadlocking executions (terminating with "Timeout"). In this case, the most likely outcome is that the application will deadlock, although each execution may deadlock at a different point, and the deadlock may involve a different set of Producers and Consumers.

## 1.2 Instrumenting the example `gen_servers` with DDMon monitors

To instrument the example application with DDMon, edit the following files:

- `example-system/lib/microchip_factory/producer.ex`
- `example-system/lib/microchip_factory/inspector.ex`

In each, uncomment the *line 3*. For example, `producer.ex` should begin as follows:

```
defmodule MicrochipFactory.Producer do
  use GenServer
  alias :ddmon, as: GenServer

  def start_link(microchip_metadata) do
    GenServer.start_link(__MODULE__, microchip_metadata, [])
  end

  ...
```

This replacement can also be performed using `sed`:

```
sed -i 's/ # alias :ddmon/ alias :ddmon/g' ./example-system/lib/microchip_factory/*.ex
```

After that, you will need to **rebuild the docker image**:

```
docker build -t ddmon .
```

Now you can try rerunning the application several times with the additional `:monitored` parameter, which enables deadlock reporting on the terminal. (For more details about this parameter, and how deadlock reports are propagated, see the section "Collecting deadlock reports" at the end of this file.)

- Small version:

```
docker run --rm ddmon bash -c 'cd example-system; mix run -e "MicrochipFactory.start_two :monitored"'
```

- Large version:

```
docker run --rm ddmon bash -c 'cd example-system; mix run -e "MicrochipFactory.start_many :monitored"'
```

If no deadlock occurs, you should get a **Success** output exactly as before. Instead, if the system deadlocks, you should see a red **Deadlock** message (instead of "Timeout"), followed by a list of PIDs: those are the PIDs of the processes that formed the deadlock cycle. The symbol `<=` marks the PID of the process whose monitor reported the deadlock. For instance, the output may look as follows (for the small version of the example):

**Deadlock:**

```
- :prod1 <=
- :insp2
- :prod2
- :insp1
```

- :prod1 <==

(For more details, and for interpreting the PIDs in the deadlock reports, see example-system/README.md.)

Observe that in the large version of the example only a subset of the application PIDs may be part of the actual deadlock cycle — although the whole application may be stuck, and other PIDs may be awaiting a response from the deadlocked PIDs.

### 1.2.1 Collecting deadlock reports

When DDMon is enabled, its deadlock reports are propagated among monitors. There are two ways for receiving deadlock reports from the “outside” of a system of monitored `gen_server` behaviours:

1. A process may subscribe to deadlock reports by calling `ddmon:subscribe_deadlocks(<Service>)`. Then, if the monitored `gen_server` instance identified by `<Service>` becomes involved in a deadlock cycle, it monitor will send a message of the form `{?DEADLOCK, DL}` to each subscriber — where `DL` is the list of services involved in the deadlock cycle.
2. A client may send a `gen_server` call of form `{?MONITORED_CALL, <Msg>}` to a *monitored gen\_server* instance — where `<Msg>` is the message that would be normally sent if the client wanted to simply call a regular (i.e., unmonitored) `gen_server` instance. Then, if the call causes a deadlock, the monitor will send back to the client a response of the form `{?DEADLOCK, DL}` where `DL` is the list of services involved in the deadlock cycle.

In both cases above, `?MONITORED_CALL` and `?DEADLOCK` are macros defined in the file `src/ddmon.hrl`.

This example application uses the first of the two methods above: the `:monitored` parameter instructs the scripts that run the application to subscribe to the deadlock reports generated by DDMon (via `ddmon:subscribe_deadlocks(...)`), and visualise them on the console. If DDMon is enabled but the `:monitored` parameter is omitted, then deadlocks are still detected by DDMon, but the corresponding deadlock reports are not displayed on the console. For more details, see the source code of `example-system/lib/microchip_factory.ex` (in particular, the function `do_calls`).

Importantly, the `:monitored` parameter must *only* be used if the system is monitored by DDMon (i.e., if the `alias` directives described above are present).