Formalising Monitors for Distributed Deadlock Detection

Radosław Jan Rowicki[1], Adrian Francalanza[2], and Alceste Scalas[1]

[1] Danmarks Tekniske Universitet, Kongens Lyngby, Denmark
rjro@dtu.dk alcsc@dtu.dk
[2] University of Malta, Msida, Malta
adrian.francalanza@um.edu.mt

Introduction

Modern software applications are often implemented as networks of (micro)services that communicate via message-passing. Many common protocols rely on forms of *remote procedure calls (RPC)*, where services (acting as RPC clients) perform external calls by sending requests to other services (acting as RPC servers) and then awaiting a response. For example, the *Open Telecom Platform* (OTP) and languages like Erlang and Elixir provide widely-used *behaviours* (like `gen_server` and `gen_statem`) to the development of RPC-based services [6]. Such behaviours offer what we dub *single-threaded RPC (SRPC)*, where each service handles one request at a time and remains idle while waiting for a response to a request that it sent; similar SRPC patterns can be found in popular actor frameworks like Akka/Pekko [7].

Such systems can run into *deadlocks* if a group of services end up waiting on each other in a circular dependency. Correctly identifying and fixing such deadlocks can be hard: in large distributed systems with high traffic, observers may see that part of the system is unresponsive and some requests time out, but these symptoms can be mistakenly attributed to performance issues. Deadlocks can be potentially prevented via static analysis [11, 15, 12, 13] — although this requires access to the source code of the whole system (which may not be available) and can produce false positives (which may be undesirable). In these cases, runtime verification via *monitors* [2, 8] can be a more practical method for identifying deadlocks as they occur.

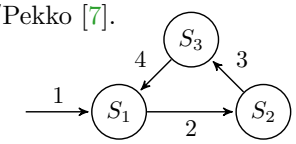

Figure 1: Deadlock example. The numbers show the order in which requests are sent.

A monitor is a process that oversees a service in order to identify faulty states. We are interested in developing *distributed* deadlock detection monitors that do not introduce centralised bottlenecks; moreover, we require our monitors to be *black-box* and *outline*, i.e., they can only observe the incoming/outgoing messages of each service without access to its internals

In this work, we present the Coq mechanisation of a theory that formalises networks of SRPC services with and without monitors, and a distributed black-box deadlock detection monitoring algorithm (inspired by [5, 14]) which we prove *sound* and *complete*. We also provide a framework (with an API inpired by Erlang and Elixir `gen_servers`) to conveniently model SRPC-based distributed applications for which we automatically construct correct deadlock monitors. To the best of our knowledge, we provide the first mechanised correctness proof of a distributed deadlock detection algorithm.

Mechanisation and Results

We define networks as functions from a set of *names* to states of services. In our model, communication is asynchronous, meaning that services never block

```
CoInductive Proc := (* Service implementation *)
| Recv (select : Name → option (Val → Proc))
| Send (to : Name) (msg : Val) (P : Proc)
| Tau (P : Proc).

Record Serv := (* service *)
  { i_que : list Msg; proc : Proc; o_que : list Msg}.
Record MServ := (* monitored service *)
  { m_que : list Msg; state : MProc; serv : Serv}.
```

Figure 2: Service DSL as defined in Coq.

when sending a message. Services are programmed directly in Gallina[10] via a coinductive DSL providing primitives for sending and receiving messages, inspired by process calculi: see fig. 2, where the `option` continuation in `Recv` allows the selective reception of messages based on senders. This design makes our model remarkably expressive while keeping our formalisation focused on communication semantics. Moreover, Coq entirely covers bindings and substitutions, which can notoriously complicate proofs if otherwise embedded [3, 1, 4]. On top of this DSL, we specify `SRPC` as a coinductive property that classifies service states as either: `Ready` while the service awaits requests, `Work`-ing when it actively processes a request, and `Lock`-ed if it awaits a response from another service.

We implement monitoring by *instrumenting* each service with a monitor process M and a monitor message buffer $\hat{q}$. Our monitors are *intercepting*, i.e., they act as proxies between their services and the network. Consequently, all incoming and outgoing messages pass through the monitor and its buffer, allowing the monitor M to observe the service's communication, update its state, and exchange messages with other monitors. Figure 3 illustrates how a monitored service interacts with the rest of the network.

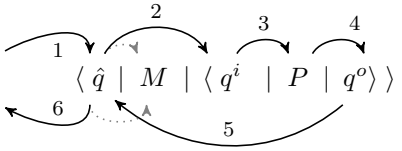

Figure 3: Journey of messages in a monitored service: (1) a message from the network reaches the monitor buffer $\hat{q}$; (2) the message is forwarded to the service's input queue q^i; (3) the service process P receives the message; (4) the service sends some message via its output queue q^o; (5) the output message reaches the monitor queue $\hat{q}$; (6) the message is forwarded to the network. The dotted arrows mean that the monitor observes messages (2) and (6), but it cannot observe the other message exchanges.

Instrumentations (`instr`) are functions that transform unmonitored SRPC networks (of type `Net`) into monitored ones (of type `MNet`) by equipping each service with a monitor process and buffer. *Vice versa*, `deinstr` "strips" monitors from a monitored network. We prove that instrumentation is *transparent*, i.e., it does not introduce nor suppress behaviours w.r.t. the original network. We formalise transparency as an *operational correspondence* [9] (proving soundness and completeness) between monitored and unmonitored networks: see Coq theorems above, where a path is a sequence of reduction steps.

```
Variable apply_instr (i : instr) : Net → MNet.
Coercion apply_instr : instr >-> Funclass.

Theorem transp_sound :
  ∀ (N₀ : Net) (i₀ : instr) path' (MN₁ : MNet),
    (i₀ N₀ =[ path' ⊨⇒ MN₁) →
    ∃ path, (N₀ =[ path ⊨⇒ deinstr MN₁).

Theorem transp_complete :
  ∀ (N₀ N₁ : Net) path (i₀ : instr),
    (N₀ =[ path ⊨⇒ N₁) →
    ∃ path' (i₁ : instr), (i₀ N₀ =[ path' ⊨⇒ i₁ N₁).
```

Concretely, each monitor M implements a deadlock detection algorithm that, based on the incoming/outgoing messages it observes, estimates the current state of the service it oversees. In particular, if it infers that the service is awaiting a response, it sends *probes* (in the style of [5, 14]) to the monitors of all services from which it receives a request. Monitors communicate by forwarding such probes; if a monitor receives back a non-outdated probe that it has previously emitted, then it concludes that there is a dependency cycle and reports a deadlock.

Following runtime verification standards, we define *correctness* of monitors as *soundness* (every reported deadlock is real) and *completeness* (all deadlocks are eventually reported, assuming fairness of components [16]). We finally prove that any distributed application modelled with our Erlang/Elixir `gen_server`-inspired framework (`gen_net dapp`) can be correctly instrumented (`gen_instr`) and monitored; we achieve this by finding and proving invariants that characterise correct instrumentations. All proofs are successfully mechanised in Coq.

```
Definition detect_sound (N₀ : Net) (i₀ : instr) :=
  ∀ path' MN₁, (i₀ N₀ =[ path' ⊨⇒ MN₁) ∧ reports_deadlock MN₁ →
  ∃ path, (N₀ =[ path ⊨⇒ deinstr MN₁) ∧ has_deadlock (deinstr MN₁).

Definition detect_complete (N₀ : Net) (i₀ : instr) :=
  ∀ path N₁, (N₀ =[ path ⊨⇒ N₁) ∧ has_deadlock N₁ →
  ∃ path' (i₁ : instr),
    (i₀ N₀ =[ path' ⊨⇒ i₁ N₁) ∧ reports_deadlock (i₁ N₁).

Theorem gen_net_instr_correct : ∀ dapp,
    detect_sound (gen_instr (gen_net dapp))
  ∧ detect_complete (gen_instr (gen_net dapp)).
```

References

[1] Beniamino Accattoli, Horace Blanc, and Claudio Sacerdoti Coen. Formalizing Functions as Processes. In *ITP 2023 - 14th International Conference on Interactive Theorem Proving*, Bialystok, Poland, July 2023. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.ITP.2023.5.

[2] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018. doi:10.1007/978-3-319-75632-5_1.

[3] Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science*, Volume 5, Issue 2, June 2009. doi:10.2168/lmcs-5(2:16)2009.

[4] Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Legesse Tirore, Martin Vassor, Nobuko Yoshida, and Daniel Zackon. The concurrent calculi formalisation benchmark. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *Lecture Notes in Computer Science*, pages 149–158. Springer, 2024. doi:10.1007/978-3-031-62697-5_9.

[5] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, May 1983. doi:10.1145/357360.357365.

[6] Ericsson AB. *Erlang/OTP System Documentation 14.2.5.8*, chapter 10, pages 313–344. 2025. Accessed: 2025-03-10. URL: https://www.erlang.org/docs/26/pdf/otp-system-documentation.pdf.

[7] Erlang/OTP Team. *Apache Pekko gRPC*. Apache Software Foundation, 2025. Accessed: 2025-03-10. URL: https://pekko.apache.org/docs/pekko-grpc/current/index.html.

[8] Adrian Francalanza. A theory of monitors. *Information and Computation*, 281:104704, 2021. doi:10.1016/j.ic.2021.104704.

[9] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010. doi:10.1016/J.IC.2010.05.002.

[10] Gérard Huet. The Gallina specification language: A case study. In Rudrapatna Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 229–240, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. doi:10.1007/3-540-56287-7_108.

[11] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998. doi:10.1145/276393.278524.

[12] Naoki Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006. doi:10.1007/11817949_16.

[13] Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017. doi:10.1016/j.ic.2016.03.004.

[14] Don P. Mitchell and Michael J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, page 282–284, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800222.806755.

[15] Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/

2603088.2603116.

[16] Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.