A Behavioral Theory For Crash Failures and Erlang-style Recoveries In Distributed Systems

Giovanni Fabbretti, Ivan Lanese, Jean-Bernard Stefani

A Behavioral Theory For Crash Failures and Erlang-style Recoveries In Distributed Systems

Giovanni Fabbretti, Ivan Lanese, Jean-Bernard Stefani

RESEARCH REPORT
N° 9511
May 2023
Project-Teams Spades and Focus

ISSN 0249-6399 ISRN INRIA/RR--9511--FR+ENG

A Behavioral Theory For Crash Failures and Erlang-style Recoveries In Distributed Systems

Giovanni Fabbretti *, Ivan Lanese†, Jean-Bernard Stefani*

Project-Teams Spades and Focus

Research Report n° 9511 — May 2023 — 50 pages

Abstract: Distributed systems can be subject to various kinds of partial failures, and building fault-tolerance or failure mitigation mechanisms for distributed systems remains an important domain of research. In this paper, we present a calculus to formally model distributed systems subject to crash failures, and in which one can encode recovery mechanisms by leveraging a small set of lightweight (in terms of implementation cost) primitives. To the best of our knowledge, our calculus is the first one with support for *all* the following characteristics: i) asynchronous communication; ii) unique location for receivers; iii) dynamic nodes and links; iv) crash failures with recovery; v) nodes with imperfect knowledge of their context. We define a contextual equivalence for our calculus in the classical form of a barbed congruence, and a notion of bisimilarity which we prove fully abstract with respect to our barbed congruence. In addition, we show by means of examples that our calculus can support Erlang-style fault management and recovery, and that our behavioral theory agrees on key instances without recovery with previous work by Francalanza and Hennessy. This paper can be understood as a complete reworking and an extension to tackle recovery of Francalanza and Hennessy's work.

Key-words: formal calculi, distributed systems, concurrency, failures, recoveries

* Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, France
† Focus Team, Univ. of Bologna/INRIA, Italy

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Une théorie comportementale des défaillances et des reprise en style Erlang dans les systèmes distribués.

Résumé : Les systèmes distribués peuvent être soumis à différents types de défaillances partielles, et l'élaboration de mécanismes de tolérance aux pannes ou d'atténuation des défaillances pour les systèmes distribués reste un domaine de recherche important. Dans cet article, nous présentons un calcul permettant de modéliser formellement les systèmes distribués soumis à des défaillances accidentelles, et dans lequel il est possible d'encoder des mécanismes de récupération en tirant parti d'un petit ensemble de primitives légères (en termes de coût d'implémentation). À notre connaissance, notre calcul est le premier à prendre en charge toutes les caractéristiques suivantes : i) communication asynchrone ; ii) emplacement unique pour les récepteurs ; iii) nœuds et liens dynamiques ; iv) pannes avec récupération ; v) nœuds ayant une connaissance imparfaite de leur contexte. Nous définissons une équivalence contextuelle pour notre calcul sous la forme classique d'une congruence barrée, ainsi qu'une notion de bisimilarité que nous prouvons totalement abstraite par rapport à notre congruence barrée. En outre, nous montrons à l'aide d'exemples que notre calcul peut prendre en charge la gestion et la récupération des fautes à la manière d'Erlang, et que notre théorie comportementale est en accord avec les travaux antérieurs de Francalanza et Hennessy en ce qui concerne les instances clés sans récupération. Cet article peut être considéré comme un remaniement complet et une extension du travail de Francalanza et Hennessy pour aborder la récupération.

Mots-clés : calcul formel, systèmes distribués, concurrence, défaillances, reprises

Contents

1 Introduction

A key characteristic of distributed computer systems is the occurrence of partial failures, which can affect part of a system, e.g. failures of the system nodes (computers and the processes they support) or failures of the connections between them. The model of crash failures with recovery, where a node can fail by ceasing to operate entirely, and later on recover its operation (typically, after some administrator intervention) is an important model of failure to consider because of its relevance in practice. However the correct design of systems in this model is far from trivial, as the recent study on bugs affecting crash recovery in distributed systems demonstrates [17].

Developing a process calculus analysis of the behavior of distributed systems with crash failures and recovery can help in this respect for it can reveal subtle phenomena in the behavior of such systems, and it can be leveraged for the development of analysis tools, such as verifiers and debuggers. Unfortunately, to the best of our knowledge, the literature contains precious few examples of process calculi accounting for crash failures with recovery, notably [15, 1, 3, 4], and none account simultaneously for the following features which we deem essential to faithfully model actual distributed systems:

1. *Asynchronous communication*: interaction between processes proceeds not by rendez-vous but by an asynchronous exchange of messages, which can possibly be lost or reordered while transiting to their destination.

2. *Unique location for receivers*: each communication targets a single location, either local or remote. This would not be the case, e.g., if communication were via channels, and receivers for a same channel could reside on different locations. This feature ensures that the complexity of message exchange is commensurate with that of simple asynchronous communication used in the Internet, and that no hidden cost, due for instance to leader election or routing protocols, is implied for the implementation of a simple message exchange (see e.g. [14] for a discussion).

3. *Dynamic nodes and links*: the number of nodes in a system, and of communication links between them, is not fixed and may vary. During execution, new nodes and links can be established, existing nodes and links can be removed, either because of failures or by design. This feature is necessary to account for actual distributed systems whose configurations may vary at run-time, notably because of failure management and of performance management (e.g. scaling decisions in cloud systems). The explicit presence of links is important because partial connections often affect large distributed systems and because introducing link failures leads to a different behavioral theory than dealing with node failures only, as noted in [16].

4. *Crash failures*: when a node or a link fails, it does so silently, by ceasing execution. This is a simplifying assumption on the failure model of actual distributed systems, but one which, barring malicious faults, can be relatively well approximated in practice, as with the *let it fail* policy in Erlang.

5. *Recovery*: when a node or link has failed, it can be revived to resume its execution or function. This may imply external intervention (e.g. by a human administrator) and may not be under the control of the running system. Note that we place no strong requirement on the resume operation, just that a recovered node be able to perform some computation: e.g. it need not guarantee that a recovered node operates exactly as prior to its crash. This weak recovery model is consistent e.g. with node recovery in Erlang, or the node failure model adopted in the Verdi framework for distributed systems verification [29].

6. *Imperfect knowledge*: in general, in a distributed system, a node has only a partial and imperfect knowledge of the overall state of the system. We call *local view* the belief a node entertains of the status of its neighbors (nodes it assumes it is connected to). Local views are explicitly maintained by distributed programming language runtimes such as that of Erlang [2] and Elixir [21]. They play an essential part of key distributed algorithms, such as failure detection [19], membership management [7], checkpoint-rollback recovery schemes [9], or gossip-based peer sampling [20].

The two works dealing with crash failures and recovery that come closest to meeting these modeling requirements are [3] and the recent [4]. However, they only consider a fixed number of nodes, and to model recovery they rely on timers and checkpointing constructs (their save primitives). Checkpointing primitives are powerful mechanisms that do not match our weak requirements to account for recovery. For instance, recovery in Erlang systems can take place without relying on checkpointing, just by restarting failed nodes and processes in a predefined state, possibly relying on persistence mechanisms for preserving data across node failures.

The goal of this paper, then, is to introduce a process calculus exhibiting all the above features, with no recourse to timers, perfect failure detection or a built-in checkpointing primitive. We have two additional requirements for this calculus. On the one hand, it has to stay close to the behavior of Erlang; on the other hand it has to stay as close as possible to the work by Francalanza and Hennessy [16]. The Erlang programming language and its environment are representative of modern distributed programming facilities. It is a functional, concurrent and distributed language based on the actor model, and it is used in several large distributed projects [12]. It is well known for its "let it fail" policy, whereby a service that is not working as expected is killed as soon as the faulty behavior is detected, if not dead already, and restarted by its supervisor. Staying close to Erlang ensures our modeling remains faithful to actual distributed systems. The work by Francalanza and Hennessy [16] constitutes a good benchmark for comparison. While not tackling recovery and relying on perfect failure detection, it contains several of the features mentioned above, and develops a behavioral theory with node and link failures. Staying close to it allows us to compare our behavioral theory with an established one, when restricting our attention to systems without recovery.

The rest of this paper is organized as follows. Section 2 introduces a motivating example, which serves also as an introduction to our calculus. Section 3 presents the calculus, its reduction semantics, and discusses key design choices. Section 4 equips our calculus with a notion of barbed congruence and characterizes it by a notion of bisimilarity to obtain a proof technique for checking system equivalence. Section 5 shows our calculus can encode constructs similar to Erlang primitives to build systems with hierarchical failure management. Moreover, we show that our behavioral theory agrees with the one in [16] on key examples withour recovery. Finally, Section 6 discusses related work and concludes. Details of proofs and complementary material are available in the Appendix.

2 Crash and Recovery: Motivating Example

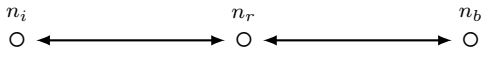
Before presenting our calculus, we discuss a motivating example which serves as an informal introduction (for the time being, we omit some details). An Erlang implementation of the example is discussed in Appendix B. Consider the following system:

$$\mathbf{servD} = \ \nu\, n_r, n_b, r_1, r_2, b.\ \Delta \triangleright \big(\ [\![I]\!]^{n_i} \parallel [\![R]\!]^{n_r} \parallel [\![B]\!]^{n_b}\ \big)$$

where:

$$I = req(y,z).\mathsf{spawn}\ n_r.\overline{r_1}\langle y,z\rangle \qquad R = (r_1(y,z).\mathsf{spawn}\ n_b.\overline{b}\langle y,z\rangle) \mid (r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle)$$
$$B = b(y,z).\mathsf{spawn}\ n_r.\overline{r_2}\langle z, w_y\rangle$$

System **servD** depicts a distributed server running on a network Δ. The network, Δ, whose formal definition we omit for the moment, can be graphically represented as follows,



where ○ represents an alive location and the arrow ◂⟷▸ represents a live bidirectional communication link between locations.

A location in our calculus represents a locus of computation. It can represent a hardware node in a distributed system, or a virtual machine or a container running on a hardware node. A location constitutes also a unit of crash failure: when a location fails, all processes inside the location cease to function. Located processes take the form $\lfloor P \rfloor^n$, where P denotes a process and where n is the location name. There can be several processes located at the same location: for instance, $\lfloor P \rfloor^n \parallel \lfloor Q \rfloor^n$ denotes two processes P and Q running in parallel inside the same location n.

The (admittedly simplistic) system **servD** behaves as follows. The *interface* process I, running on n_i, awaits a single request from the environment on channel req. Once received, the elements of the request (a parameter y, and a return channel z) are routed to location n_b, which runs the *backend* process B, through location n_r, which hosts the *router* process R, as there is no direct link between n_i and n_b. The router awaits for the elements of the request on a private channel r_1 and forwards them by spawning a message $\overline{b}\langle y,z\rangle$ on n_b, where b is a private channel. Message sending in our calculus can only occur locally. For two remote locations to communicate, like for n_i and n_r, it is necessary for one to asynchronously spawn the message on the other one. The backend handles the information y and returns the answer w_y to the interface, again by routing it through n_r. Finally, the interface emits the answer on z for the client to consume it.

The following is a possible client for **servD**

$$\lfloor \mathsf{spawn}\ n_i.(\overline{req}\langle h,z\rangle \mid z(w).Q) \rfloor^c$$

It sends a request to the interface n_i by spawning it on n_i. It also spawns a process on n_i to handle the response, which will take the form of a message on channel z located on n_i.

Now, consider the following system:

$$\mathbf{servDF} = \nu\, n_r, n_b, n_c, r_1, r_2, b, retry.\ \Delta' \triangleright (\quad \lfloor J \rfloor^{n_i} \parallel \lfloor R \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor !C \rfloor^{n_c} \quad)$$

where:

$$J = req(y,z).((\mathsf{spawn}\ n_r.\overline{r_1}\langle y,z\rangle) \mid retry.\mathsf{spawn}\ n_r.\overline{r_1}\langle y,z\rangle)$$
$$C = \mathsf{create}\ n_r.(R \mid \mathsf{spawn}\ n_i.\overline{retry})$$

Here, Δ' could be graphically represented like Δ only with an extra link between n_r and n_c. System **servDF** represents a distributed server where location n_r may be subject to one failure, modeled by the primitive kill. The system has a recovery mechanism in place to deal with that potential failure: the *controller* $\lfloor !C \rfloor^{n_c}$ is a location that keeps trying, through the apposite primitive create, to recreate n_r (the ! operator is akin to the π-calculus operator for replication), together with a message to restart the handling of the request. The interface is more sophisticated as it can now send a second request when asked to retry if something goes wrong with the first one. If n_r fails, the controller can create another router with a message $retry$ for the interface which will start the second attempt.

At first glance, **servDF** seems to correctly handle the failure scenario, but that is not the case. Indeed, consider an execution where the failure happens after the request has already been handled. In such case, the request would be processed again and another response sent back by the interface, a behavior that cannot be exhibited by **servD**. If one considers **servD** as the specification to meet, **servDF** does not satisfy it.

The following system correctly handles recovery, in that it meets the **servD** specification:

$$\mathbf{servDFR} = \nu\, n_r, n_b, n_c, r_1, r_2, b, c, \mathit{retry}.\ \Delta' \triangleright \big(\quad [\![K]\!]^{n_i} \parallel [\![R]\!]^{n_r} \parallel [\![\mathsf{kill}]\!]^{n_r} \parallel [\![!B]\!]^{n_b} \parallel [\![!C]\!]^{n_c} \quad \big)$$

where:

$$K = \mathit{req}(y, z).((\mathsf{spawn}\ n_r.\overline{r_1}\langle y, c\rangle) \mid c(w).\overline{z}\langle w\rangle \mid \mathit{retry}.\mathsf{spawn}\ n_r.\overline{r_1}\langle y, c\rangle)$$

In system **servDFR** the response from the backend is not directly sent to the client, but goes through a private channel c on which the interface listens only once. This mechanism prevents emitting the answer to the request twice. System **servDFR** can be understood as a masking 1-fault tolerant system, equivalent to the ideal one **servD**. We develop in Section 4 a behavioral theory able to tell apart **servD** from **servDF**, and able to prove equivalent **servD** and **servDFR**.

3 The Calculus

3.1 Names and notations

We assume given mutually disjoint infinite denumerable sets C, N and I. C is the set of *channel name*s, N is the set of *location names*, and I is the set of *incarnation variables*. We use the set of integers $\mathbb{Z}$ as the set of *incarnation numbers*. An incarnation number is paired with a location name for recovery purposes, to distinguish the current instance of a location from its past failed instances. We denote by $\mathsf{N}^{\odot}$ the set $\mathsf{N} \cup \{\odot\}$, where $\odot \notin \mathsf{N}$. As in the π-calculus, channel names can be free or bound in terms. The same holds for location names. Incarnation variables can be bound, but not incarnation numbers. We denote by $\tilde{u}$ a finite (possibly empty) tuple of elements. We write $T\{\tilde{v}/\tilde{u}\}$ for the usual capture-avoiding substitution of elements of $\tilde{u}$ by elements of $\tilde{v}$ in term T, assuming tuples $\tilde{u}$ and $\tilde{v}$ have the same arity. We write $u, \tilde{v}$ or $\tilde{v}, u$ for the tuple $\tilde{v}$ extended with element u as first or last element. Abusing notation, we sometimes identify a tuple $\tilde{u}$ with the set of its elements. We denote by $\mathbb{N}^+$ the set of strictly positive integers (by definition $0 \notin \mathbb{N}^+$), and by $\mathbb{N}$ the set of positive integers ($0 \in \mathbb{N}$). We denote by $\hat{\mathbf{0}}$ the function $\hat{\mathbf{0}} : \mathsf{N}^{\odot} \to \mathbb{Z}$ that maps any $n \in \mathsf{N}$ to 0 and $\odot$ to 1.

3.2 Syntax

Systems in our calculus are defined through three levels of syntax, one for *processes*, one for *configurations*, one for *systems*.

The syntax of *processes* is defined as follows:

$$\begin{array}{rcl} P, Q & ::= & \mathbf{0} \mid \overline{x}\langle\tilde{u}\rangle.P \mid x(\tilde{v}).P \mid\ !x(\tilde{v}).P \mid \nu\, w.P \mid \mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid P \mid Q \\ & & \mathsf{node}(n, \lambda).P \mid \mathsf{remove}\ n.P \mid \mathsf{spawn}\ n.P \mid \mathsf{kill} \mid \\ & & \mathsf{create}\ n.P \mid \mathsf{link}\ n.P \mid \mathsf{unlink}\ n.P \end{array}$$

where: $\tilde{u}, r, s \subset \mathsf{C} \cup \mathsf{N}^{\odot} \cup \mathsf{I} \cup \mathbb{Z}$ $\quad \tilde{v} \subset \mathsf{C} \cup \mathsf{N} \cup \mathsf{I}$ $\quad w \in \mathsf{C} \cup \mathsf{N}$ $\quad x \in \mathsf{C}$ $\quad n \in \mathsf{N}$ $\quad \lambda \in \mathsf{I}$

Terms of the form $x(\tilde{u}).P$, $\nu\, w.P$, and $\mathsf{node}(n, \lambda).P$ are binding constructs for their arguments $\tilde{u}$, w and n, λ, respectively.

The syntax of processes is that of the π-calculus with matching [26] and replicated receivers (first line of productions), extended with primitives for distributed computing inspired from the Erlang programming language (second line of productions), and three primitives to activate locations, establish and remove links (third line of productions).

$\mathbf{0}$ is the null process which can take no action. Processes can communicate by emitting a message $\overline{x}\langle\tilde{v}\rangle.P$ which can be received by some receiver process of the form $x(\tilde{u}).Q$ residing on the same location. We write $\overline{x}\langle\tilde{u}\rangle$ for $\overline{x}\langle\tilde{v}\rangle.\mathbf{0}$, and just $\overline{x}$ when $\tilde{u}$ is empty. We assume the calculus is well-sorted, so that the arity of receivers always matches that of received messages. The construct $!x(\tilde{u}).P$ is the replicated input construct, which replicates itself when receiving a message on channel x. Note that we use in our examples (as we did in the previous Section) the short-hand $!P$ for $\nu\, c.(\overline{c} \mid\ !c.(P \mid \overline{c}))$. The construct $\nu\, w.P$ is the standard restriction construct, which creates a fresh location or channel name. If $\tilde{u}$ is a (possibly empty) tuple of names, we write $\nu\,\tilde{u}.P$ for $\nu\, u_1 \ldots \nu\, u_n.P$ if $\tilde{u} = (u_1, \ldots, u_n)$. If $\tilde{u}$ is empty, $\nu\,\tilde{u}.P$ is just P. The construct $\mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q$ tests the equality of names r and s and continues as P if the names match and as Q otherwise. The construct $P \mid Q$ is the standard parallel composition for processes. Primitive $\mathsf{node}(n, \lambda).P$ substitutes n with the name of the current location, λ with the current incarnation number, and continues as P. Primitive $\mathsf{remove}\, n.P$ removes the location with name n from the local view of the current location and continues as P. Primitive $\mathsf{spawn}\, n.P$ launches process P at the location named n, if the latter is accessible. Primitive kill stops the current location in its current incarnation: no process can execute on a killed location; kill also models the crash of a location. Primitive $\mathsf{create}\, n.P$ creates a new location n, or reactivates a killed location with a new incarnation number, and launches process P on it. Primitive $\mathsf{link}\, n.P$ creates a connection between the current location and n and continues as P, while, $\mathsf{unlink}\, n.P$ breaks the link between the current location and n and continues as P; unlink also models the failure of a link.

The syntax of *configurations* is defined as follows:

$$L, M, N ::= \mathbf{0} \mid [\![P]\!]^n_\lambda \mid N \parallel M \qquad \text{where:} \qquad n \in \mathsf{N}^{\odot} \qquad \lambda \in \mathbb{Z}$$

A configuration can be the empty configuration $\mathbf{0}$, a located process $[\![P]\!]^n_\lambda$ or a parallel composition of configurations $N \parallel M$. A located process $[\![P]\!]^n_\lambda$ is a process P running on location (n, λ), where n is the name of the location and λ is an *incarnation number* (used for recovery). In examples, we may drop incarnation numbers of located processes if they are not relevant. Note that the special name $\odot$ identifies a well-known location which we will assume to be un-killable. Location $\odot$ is used merely for technical purposes to ensure we can simply build appropriate contexts for running systems. Apart from being unkillable, location $\odot$ behaves just as any other locations. We denote by $\mathbb{L}$ the set of configurations.

The syntax of *systems* is defined as follows:

$$S, R ::= \Delta \triangleright N \mid \nu\, w.S \qquad \text{where:} \qquad w \in \mathsf{N} \cup \mathsf{C}$$

A system is the composition of a network Δ with a configuration N, or a system under a name (channel or location) restriction. We denote by $\mathbb{S}$ the set of systems. A *network* Δ, is a tuple $\langle \mathcal{A}, \mathcal{L}, \mathcal{V} \rangle$ where

- $\mathcal{A}$ is a function $\mathcal{A} : \mathsf{N}^{\odot} \to \mathbb{Z}$ such that $\mathcal{A}(\odot) = 1$ and such that the set $\mathsf{supp}(\mathcal{A}) \stackrel{\text{def}}{=} \{n \in \mathsf{N} \mid \mathcal{A}(n) \neq 0\}$ is finite. Function $\mathcal{A}$ may record three types of information on locations. If $\mathcal{A}(n) = \lambda \in \mathbb{N}^+$, then location n is alive and its current incarnation number is λ. If $\mathcal{A}(n) = -\lambda, \lambda \in \mathbb{N}^+$ then location n has been killed and its last incarnation number while alive was λ. If $\mathcal{A}(n) = 0$, then there is no location n in the network, alive or not. Because

of the finiteness condition above, a network can only host a finite number of locations, alive or dead.

- $\mathcal{L} \subseteq \mathsf{N}^{\odot} \times \mathsf{N}^{\odot}$ is the set of links between locations. $\mathcal{L}$ is a finite symmetric binary relation over location names such that $\mathsf{dom}(\mathcal{L}) \stackrel{\text{def}}{=} \{n \in \mathsf{N}^{\odot} \mid \exists m, (n,m) \in \mathcal{L}\}$ is finite.

- $\mathcal{V} : \mathsf{N}^{\odot} \to (\mathsf{N}^{\odot} \to \mathbb{N})$ is a function that maps location names to their *local view*, which is such that the set $\mathsf{supp}(\mathcal{V}) \stackrel{\text{def}}{=} \{n \in \mathsf{N} \mid \mathcal{V}(n) \neq \hat{\mathbf{0}}\}$ is finite and $\mathsf{supp}(\mathcal{V}) \subseteq \mathsf{supp}(\mathcal{A})$. The local view of a location n is a function $\mathcal{V}(n) : \mathsf{N}^{\odot} \to \mathbb{N}$ such that the set $\{m \in \mathsf{N} \mid \mathcal{V}(n)(m) \neq 0\}$ is finite. If $\mathcal{V}(n)(m) = \kappa \in \mathbb{N}^{+}$, then location m in its incarnation κ is believed by n to be alive. If $\mathcal{V}(n)(m) = 0$, then location n holds no belief on the status of location m.

For convenience we use $\Delta_{\mathcal{A}}$, $\Delta_{\mathcal{L}}$, and $\Delta_{\mathcal{V}}$ to denote the individual components of a network representation Δ, and we use the following notations for extracting information from Δ:

- $\Delta \vdash n_{\lambda} : \mathsf{alive}$ if $\Delta_{\mathcal{A}}(n) = \lambda$ and $\lambda \in \mathbb{N}^{+}$.
- $\Delta \vdash n : \mathsf{dead}$ if $\Delta_{\mathcal{A}}(n) \notin \mathbb{N}^{+}$
- $\Delta \vdash n \leftrightarrow m$ if $(n,m) \in \Delta_{\mathcal{L}}$
- $\Delta \vdash n_{\lambda} \leftrightsquigarrow m_{\kappa}$ if $(n,m) \in \Delta_{\mathcal{L}}$, $\Delta \vdash n_{\lambda} : \mathsf{alive}$ and $\Delta \vdash m_{\kappa} : \mathsf{alive}$

We now define update operations over a network Δ.

Definition 1 (Network updates)**.** *Network update operations are defined as follows:*

- $\Delta \oplus n \leftrightarrow m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}} \cup \{(n,m),(m,n)\}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \ominus n \leftrightarrow m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}} \setminus \{(n,m),(m,n)\}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \oplus (n,\lambda) = \langle \Delta_{\mathcal{A}}[n \mapsto \lambda], \Delta_{\mathcal{L}} \cup \{(n,n)\}, \Delta_{\mathcal{V}}[n \mapsto \hat{\mathbf{0}}] \rangle$
- $\Delta \ominus (n,\lambda) = \langle \Delta_{\mathcal{A}}[n \mapsto -\lambda], \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \oplus n \succ (m,\lambda) = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}}[n \mapsto \Delta_{\mathcal{V}}(n)[m \mapsto \lambda]] \rangle$, *if* $n \neq m$
- $\Delta \ominus n \succ m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}}[n \mapsto \Delta_{\mathcal{V}}(n)[m \mapsto 0]] \rangle$, *if* $n \neq m$
- $\Delta \ominus n \succ n = \Delta \oplus n \succ (n,\lambda) = \Delta$

$\Delta \oplus n \leftrightarrow m$ and $\Delta \ominus n \leftrightarrow m$ add and remove a link, respectively, between n and m. $\Delta \oplus (n,\lambda)$ activates location n with incarnation number λ, and sets its view to the empty one. $\Delta \ominus (n,\lambda)$ kills a location in its incarnation λ. $\Delta \oplus n \succ (m,\lambda)$ adds (m,λ) to the view of n, and $\Delta \ominus n \succ m$ removes any belief on location m from the view of n.

We use below the notion of closed systems. Closed systems are systems which do not have free incarnation variables. The definition of free variables in systems and that of free names in configurations is completely standard. The notion of free names in a system is slightly unconventional because of the presence of a network, but it can be defined straightforwardly (see Appendix A for details).

$$[\text{S.Par.C}]\ N \parallel M \equiv M \parallel N \qquad [\text{S.Par.A}]\ (L \parallel M) \parallel N \equiv L \parallel (M \parallel N) \qquad [\text{S.Par.N}]\ (N \parallel \mathbf{0}) \equiv N$$

$$[\text{S.Res.C}]\ \nu\, u.\nu\, v.S \equiv \nu\, v.\nu\, u.S \qquad [\text{S.Res.Nil}]\ \frac{}{\nu\, u.S \equiv S}\ u \notin \mathsf{fn}(S) \qquad [\text{S.}\alpha]\ \frac{S =_\alpha R}{S \equiv R}$$

$$[\text{S.Ctx}]\ \frac{N \equiv M}{\mathbb{C}[N] \equiv \mathbb{C}[M]}$$

Figure 1: Structural Congruence Rules

Assuming $\Delta \vdash n_\lambda : \mathsf{alive}$

MSG
$$\overline{\Delta \triangleright [\![\, \overline{x}\langle \tilde{v} \rangle.Q \,]\!]^n_\lambda \parallel [\![\, x(\tilde{u}).P \,]\!]^n_\lambda \longrightarrow \Delta \triangleright [\![\, Q \,]\!]^n_\lambda \parallel [\![\, P\{\tilde{v}/\tilde{u}\} \,]\!]^n_\lambda}$$

BANG
$$\overline{\Delta \triangleright [\![\, !x(\tilde{u}).P \,]\!]^n_\lambda \longrightarrow \Delta \triangleright [\![\, x(\tilde{u}).(P \mid\ !x(\tilde{u}).P) \,]\!]^n_\lambda}$$

NEW
$$\overline{\Delta \triangleright [\![\, \nu\, u.P \,]\!]^n_\lambda \longrightarrow \nu\, u.\Delta \triangleright [\![\, P \,]\!]^n_\lambda}\ u \notin \mathsf{fn}(\Delta) \cup \{n, \lambda\}$$

FORK
$$\overline{\Delta \triangleright [\![\, P \mid Q \,]\!]^n_\lambda \longrightarrow \Delta \triangleright [\![\, P \,]\!]^n_\lambda \parallel [\![\, Q \,]\!]^n_\lambda}$$

IF-NEQ
$$\overline{\Delta \triangleright [\![\, \mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q \,]\!]^n_\lambda \longrightarrow \Delta \triangleright [\![\, Q \,]\!]^n_\lambda}\ r \neq s$$

NODE
$$\overline{\Delta \triangleright [\![\, \mathsf{node}(m, \kappa).P \,]\!]^n_\lambda \longrightarrow \Delta \triangleright [\![\, P\{n, \lambda/m, \kappa\} \,]\!]^n_\lambda}$$

IF-EQ
$$\overline{\Delta \triangleright [\![\, \mathsf{if}\ r = r\ \mathsf{then}\ P\ \mathsf{else}\ Q \,]\!]^n_\lambda \longrightarrow \Delta \triangleright [\![\, P \,]\!]^n_\lambda}$$

REMOVE
$$\overline{\Delta \triangleright [\![\, \mathsf{remove}\ m.P \,]\!]^n_\lambda \longrightarrow \Delta \ominus n \succ m \triangleright [\![\, P \,]\!]^n_\lambda}$$

Figure 2: Local Rules

3.3 Reduction Semantics

The operational semantics of our calculus is defined via a reduction semantics given by a binary relation $\longrightarrow \subseteq \mathbb{S} \times \mathbb{S}$ between closed systems, and a structural congruence relation $\equiv \subseteq\ \mathbb{S}^2 \cup \mathbb{L}^2$, that is a binary equivalence relation between systems and between configurations. Evaluation contexts are "systems with a hole $\cdot$" defined by the following grammar:

$$\mathbb{C}\ ::=\ \nu\, \tilde{w}.\Delta \triangleright \mathbb{E} \qquad \mathbb{E}\ ::=\ \cdot \ \big|\ (N \parallel \mathbb{E}) \qquad \text{where:} \qquad \tilde{w} \subset \mathsf{N} \cup \mathsf{C}$$

Relation $\equiv$ is the smallest equivalence relation defined by the rules in Fig. 1, where $=_\alpha$ stands for equality up to alpha-conversion, $M, N, L \in \mathbb{L}$, and $S, R \in \mathbb{S}$. Most rules are mundane. Rule S.Ctx turns $\equiv$ into a congruence for the parallel and restriction operators. Alpha-conversion on systems, which appears in Rule S.α, is slightly unusual but can be defined straightforwardly (see Appendix A for details).

The reduction relation $\longrightarrow$ is defined by the rules in Fig. 2, 3 and 4. Fig. 2 depicts the *local* reduction rules, i.e., those rules that involve only a single location and that essentially do not modify the network. Rule msg defines the receipt of a message by an input process. Rule bang defines the expansion of a replicated input process. Rule new performs the scope extrusion of a name from a process to a system. We introduced this rule as a computational step instead of a structural congruence rule for it simplifies our proofs. Rule fork turns a parallel composition into parallel threads in the same location. Rules if-eq, if-neq define the semantics of the branching construct. Rule node gets hold of the current location name and its incarnation number for further processing. Finally, rule remove deletes from the local view of the current location the belief it may hold about a given location m.

Fig. 3 depicts the *distributed* rules, i.e., rules that involve several locations or modify the network. Rule spawn-s defines a successful spawn, conditional upon the fact that a link exists

Assuming $\Delta \vdash n_\lambda : \mathsf{alive}$

$$\textsc{spawn-s}\quad \frac{}{\Delta \triangleright [\,\mathsf{spawn}\ m.P\,]^n_\lambda \longrightarrow \Delta \oplus m \succ (n,\lambda) \triangleright [\,P\,]^m_\kappa} \begin{array}{l}\Delta_n(m) = \kappa \\ \Delta \vdash n_\lambda \rightsquigarrow m_\kappa\end{array}$$

$$\textsc{spawn-f}\quad \frac{}{\Delta \triangleright [\,\mathsf{spawn}\ m.P\,]^n_\lambda \longrightarrow \Delta \ominus n \succ m \triangleright \mathbf{0}} \begin{array}{l}\Delta_n(m) = \kappa \\ \Delta \nvdash n_\lambda \rightsquigarrow m_\kappa\end{array}$$

$$\textsc{create-s}\quad \frac{}{\Delta \triangleright [\,\mathsf{create}\ m.P\,]^n_\lambda \longrightarrow \Delta \oplus (m,\kappa+1) \triangleright [\,P\,]^m_{\kappa+1}} \begin{array}{l}\Delta \vdash m : \mathsf{dead} \\ \Delta_{\mathcal{A}}(m) = -\kappa\end{array}$$

$$\textsc{create-f}\quad \frac{}{\Delta \triangleright [\,\mathsf{create}\ m.P\,]^n_\lambda \longrightarrow \Delta \triangleright \mathbf{0}}\ \Delta \nvdash m : \mathsf{dead}$$

$$\textsc{kill}\quad \frac{}{\Delta \triangleright [\,\mathsf{kill}\,]^n_\lambda \longrightarrow \Delta \ominus (n,\lambda) \triangleright \mathbf{0}}\ n \neq \odot$$

$$\textsc{link}\quad \frac{}{\Delta \triangleright [\,\mathsf{link}\ m.P\,]^n_\lambda \longrightarrow \Delta \oplus n \leftrightarrow m \triangleright [\,P\,]^n_\lambda}\ \Delta \nvdash n \leftrightarrow m$$

$$\textsc{unlink}\quad \frac{}{\Delta \triangleright [\,\mathsf{unlink}\ m.P\,]^n_\lambda \longrightarrow \Delta \ominus n \leftrightarrow m \triangleright [\,P\,]^n_\lambda}\ \Delta \vdash n \leftrightarrow m$$

Figure 3: Distributed Rules

between the spawning and target locations, and that the spawning location rightly believes the target location, with the incarnation recorded in its view, to be alive, or has no belief on the target location in its local view. This last constraint is captured by the side condition $\Delta_n(m) = \kappa$, which we formally define as follow:

$$\Delta_n(m) = \begin{cases} \kappa & \text{if } n = m \text{ and } \Delta_{\mathcal{A}}(n) = \kappa \\ \kappa & \text{if } n \neq m \text{ and } \Delta_{\mathcal{V}}(n)(m) = \kappa \\ \kappa & \text{if } n \neq m \text{ and } \Delta_{\mathcal{V}}(n)(m) = 0 \text{ and } \Delta_{\mathcal{A}}(m) = \kappa \\ 0 & \text{otherwise} \end{cases}$$

Note that a successful spawn updates the target local view with the belief that the spawning location is alive with the incarnation number it had when it initiated the spawn. Rule SPAWN-F defines a failed spawn, which may fail due to a wrong view, to a missing link between the two locations, or because the remote location is not alive. The view of the sender is updated by removing the belief on the target. This behavior is inspired by Erlang whose runtime, in a manner transparent to the user, updates the view of the spawning location if it receives no acknowledgement from the target as part of its distributed protocol.

Rules LINK and UNLINK define respectively the establishment of a link and the removal of a link. Rule CREATE-S defines the successful creation or reactivation of a location, provided it did not already exist in the network or was crashed. The newly activated location has an incarnation number that is the successor of the previous one (0 by convention if the location was not present in the network). Note that we do not require the existence of an alive link between the current location and the one to be activated since we want this operation to also model the possibility of interventions external to the system, such as those performed by human administrators. Note also that a location cannot be created anew each time, otherwise it would be impossible to resume the execution of a service under a well-known name. The use of incarnation numbers provides support for recovery schemes (see the discussion below in Section 3.4). Rule CREATE-F defines

$$\text{PAR}\ \frac{\Delta \triangleright N \longrightarrow \nu\,\tilde{u}.\Delta' \triangleright N' \quad \tilde{u} \cap \mathsf{fn}(M) = \emptyset}{\Delta \triangleright N \parallel M \longrightarrow \nu\,\tilde{u}.\Delta' \triangleright N' \parallel M} \qquad \text{RES}\ \frac{S \longrightarrow S'}{\nu\,u.S \longrightarrow \nu\,u.S'} \qquad \text{STR}\ \frac{S \equiv S' \quad S' \longrightarrow R' \quad R' \equiv R}{S \longrightarrow R}$$

Figure 4: Contextual rules

the failure of a create operation, which can fail because the location to activate may already be alive. Rule KILL defines the killing of a location.

Fig. 4 shows the contextual rules of our calculus. Rules PAR, and RES are the rules respectively for parallel execution, and execution under restriction. Rule PAR is slightly unconventional in its use of restriction. When we consider the case where $\tilde{u}$ is empty in rule PAR, we obtain a more standard-looking contextual rule for the parallel operator. However, we also have to consider cases when the active branch in the composition promotes a restriction at system level (via an application of rule NEW). In this case we have to avoid name capture by the idle branch in the composition. This way of proceeeding, coupled with the systematic presence of the distinguished location $\odot$ in any network, spares us the need to introduce parallel composition between systems. The intuition is that we can always extend a system with a process located on $\odot$ (to ensure it is alive) performing the desired changes on the public part of the network. This intuition is formalized in Appendix C.

3.4 Discussion

Communication. In our calculus, communication is local. Remote communication is obtained using the spawn operation (very similar to the go operation in [16]) to send to a target node a process performing an output of a message. As a result remote communication is asynchronous (since the spawn is so), and there is a single location where the receiver can reside (since such location is specified in the spawn). This avoids the need of using a type system to ensure that possible receivers are located on a same node, differently from calculi based on channel-based remote communication such as [1].

Incarnation numbers. Incarnation numbers are called creation numbers in Erlang [11]. We introduce them in our calculus for two main reasons. On the one hand, this ensures we are faithful to Erlang and to the behavior Erlang systems exhibit in presence of failures (more details are provided in Appendix B). On the other hand, this ensures we have basic support in place for encoding different recovery schemes. Incarnation numbers ensure that a message issued by a previous incarnation of a location can safely be dropped, avoiding message duplication across different incarnations of the same location. They are present, under various names such as incarnation or epoch numbers, in several rollback-recovery schemes surveyed in [9], such as optimistic recovery [28, 6] or causal logging schemes [10]. They are also used for scalable distributed failure detection schemes [19] and in the SWIM protocol combining failure detection and membership management [7].

Imperfect knowledge. In distributed systems, the only way for locations to know something about the context that surrounds them is to communicate. If a location n receives a message from a remote one then n learns something on the context, namely that at some point in time the remote location was alive and working since it sent a message to n. Nonetheless, n cannot infer anything on the current status of the remote location or the status of the connection. Indeed, it could have stopped right after sending the message or the link could have broken right after the message was received or both. Erlang systems, like many others, have an optimistic approach: after a first two-way interaction two locations establish a mutual knowledge of their respective

incarnations, typically by means of a shared socket connection. From that point on, they keep using that shared connection until their view changes, rather than setting up a new connection for each message exchange. Reflecting this in our calculus plays a role in the semantics of our spawn primitive whenever the view of the locality is not in sync with the real state of the system. This in turn plays a role in our behavioral theory, as the following example illustrates.

Consider a variant **servDFV** of our running example where n_c is linked to n_i instead of n_r, the network is such that the router n_r is in its incarnation κ and the local view of the interface location n_i contains $n_r \mapsto \kappa$. The controller process running on n_c is defined as follows:

$$C = \mathsf{create}\ n_r.R \mid \mathsf{spawn}\ n_i.\overline{retry}$$

Now, **servDFV** is not equivalent to **servD** due to the fact that, in case of failure of n_r, it is the controller informing the interface to restart the request and not the router n_r, thus failing to update n_i's local view with the knowledge of n_r's new incarnation. A message from the interface at this point would fail as its local view contains the previous incarnation of n_r. If not for the imperfect knowledge of the context, **servDFV** would have been equivalent to **servD**. In Appendix B we discuss an implementation of this system and we show that this behavior arises in reality too.

4 Behavioral Theory

4.1 Weak Barbed Congruence

We define a standard notion of contextual equivalence called *weak barbed congruence*, originally proposed in [24]. We denote by $\Longrightarrow$ the reflexive and transitive closure of the reduction relation $\longrightarrow$. We rely on a notion of observables on systems, called *barbs*, formally defined as follows:

Definition 2 (Barb)**.** *A system S exhibits a barb on channel x at location n in its incarnation λ, in symbols $S{\downarrow_{x@n_\lambda}}$, iff $S \equiv \nu\, \tilde{u}.\Delta \triangleright \lfloor \overline{x}\langle \tilde{v} \rangle.P \rfloor^n_\lambda \parallel N$, for some $x, n, \lambda, \tilde{u}, \tilde{v}, P, N$, where $x, n \notin \tilde{u}$, and $\Delta \vdash n_\lambda : \mathsf{alive}$. Also, $S{\Downarrow_{x@n_\lambda}}$ iff $S \Rightarrow S'$ and $S'{\downarrow_{x@n_\lambda}}$.*

We now define standard properties expected for a contextual equivalence.

Definition 3 (System congruence)**.** *An equivalence relation $\mathcal{R}$ over closed systems is a* system congruence *iff, whenever $\nu\, \tilde{u}.\Delta_1 \triangleright N \ \ \mathcal{R} \ \ \nu\, \tilde{v}.\Delta_2 \triangleright M$, for any names $\tilde{w}$ and for any configuration L such that $fn(L) \cap \tilde{u} = fn(L) \cap \tilde{v} = \emptyset$, we have:*

$$\nu\, \tilde{w}.\nu\, \tilde{u}.\Delta_1 \triangleright N \parallel L \quad \mathcal{R} \quad \nu\, \tilde{w}.\nu\, \tilde{v}.\Delta_2 \triangleright M \parallel L$$

Definition 4 (Weak Barb-preserving relation)**.** *A relation $\mathcal{R}$ over closed systems is* weak barb-preserving *iff whenever $S\ \mathcal{R}\ R$ and $S{\downarrow_{x@n}}$ then $R{\Downarrow_{x@n}}$.*

Definition 5 (Weak reduction-closed relation)**.** *A relation $\mathcal{R}$ over closed systems is* weak reduction-closed *iff whenever $S\ \mathcal{R}\ R$ and $S \longrightarrow S'$ then $R \Longrightarrow R'$ for some R' such that $S'\ \mathcal{R}\ R'$.*

Definition 6 (Weak barbed congruence)**.** Weak barbed congruence*, noted $\dot{\approx}$, is the largest weak barb-preserving, reduction-closed, system congruence.*

As a first result, we can check that structural congruence is included in weak barbed congruence:

Assuming $\Delta \vdash n_\lambda : \mathsf{alive}$

$$\text{L-IN}\quad \frac{}{\Delta \triangleright [\, x(\tilde{v}).P \,]^n_\lambda \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright [\, P\{\tilde{u}/\tilde{v}\} \,]^n_\lambda} \qquad \text{L-OUT}\quad \frac{}{\Delta \triangleright [\, \overline{x}\langle \tilde{v} \rangle.P \,]^n_\lambda \xrightarrow{\overline{x}\langle \tilde{v} \rangle@n_\lambda} \Delta \triangleright [\, P \,]^n_\lambda}$$

Figure 5: Concurrent and distributed Rules

Proposition 1. *Structural congruence $\equiv$ is a weak barb-preserving reduction-closed system congruence.*

Proof. Structural congruence is a barb-preserving system-congruence by definition. The fact that structural congruence is weak reduction-closed follows from propositions 5 and 6 in Appendix E. □

Much as in [25], a simpler kind of barbs gives rise to the same barbed congruence. The interested reader can find a discussion in Appendix D. We keep the more detailed observables $\downarrow_{x@n_\lambda}$ for convenience, to simplify certain arguments in our proofs.

4.2 A Labeled Transition Semantics

In this section we present a labeled transition semantics for our calculus in order to have a co-inductive characterization of weak barbed congruence. Labels α in our LTS semantics take the following forms:

$$\alpha ::= \quad \tau \mid \nu\, \tilde{w}.\overline{x}\langle \tilde{u} \rangle@n_\lambda \mid x(\tilde{u})@n_\lambda \mid \mathsf{kill}(n,\lambda) \mid \mathsf{create}(n,\lambda) \mid \oplus n_\lambda \mapsto m \mid \ominus n_\lambda \mapsto m \mid n_\lambda \succ m$$

The first three labels are classical: silent action, output action (possibly with restricted names), and input action. Output and input action mention the name and incarnation of the location performing the action. Labels $\mathsf{kill}(n,\lambda)$ and $\mathsf{create}(n,\lambda)$ indicate respectively the killing and activation of location n in incarnation λ. Labels $\oplus n_\lambda \mapsto m$ and $\ominus n_\lambda \mapsto m$ signal respectively the creation and destruction of a link between n and m, initiated by n at incarnation λ. Finally, $n_\lambda \succ m$ signals that location n at incarnation λ holds the correct belief about location m.

The transition relation labeled by α is denoted $\xrightarrow{\alpha}$. We denote by $\stackrel{\tau}{\Longrightarrow}$ the reflexive and transitive closure of $\xrightarrow{\tau}$. For $\alpha \neq \tau$, we denote by $\stackrel{\alpha}{\Longrightarrow}$ the relation $\stackrel{\tau}{\Longrightarrow} \xrightarrow{\alpha} \stackrel{\tau}{\Longrightarrow}$.

Transitions relations $\xrightarrow{\alpha}$ in our LTS semantics are defined inductively by several sets of inference rules. The first set of rules, not shown here, contains the equivalent of all the local rules of the reduction relation in Figure 2, except for rule MSG. Fig. 5 depicts part of the the rules for concurrent and distributed primitives. Rules L-IN and L-OUT are as in the standard early instantation-style LTS for the π-calculus [26]. Rules which are not shown contain the equivalent of local rules in Fig. 3. Rules which are not shown are derived from Figures 2 and 3 by replacing $\longrightarrow$ by $\xrightarrow{\tau}$ (see Appendix A for the full list of rules).

Fig. 6 depicts the rules modeling the possible interactions of a context on the public part of the network. In particular the creation of a new location (L-CREATE-EXT), the killing of a location (L-KILL-EXT), the linking of two locations (L-LINK-EXT) or the unlinking of two locations (L-UNLINK-EXT). Finally, rule L-VIEW imposes equality of views of locations for two equivalent systems.

Fig. 7 depicts composition rules for the labeled transition semantics. Rules L-PAR_L and L-SYNC_L have symmetric rules L-PAR_R and L-SYNC_R, which are not shown. Most rules are mundane, we only discuss the non standard ones. Rules L-PAR_L is the standard rule for parallel composition allowing independent evolution of one branch of the composition. The side condition on the idle

$$\text{L-CREATE-EXT}\quad \frac{}{\Delta \triangleright N \xrightarrow{\mathsf{create}(n,\kappa+1)} \Delta \oplus (n,\kappa+1) \triangleright N} \quad \begin{array}{l}\Delta \vdash n : \mathsf{dead}\\ \Delta_{\mathcal{A}}(n) = -\kappa\end{array}$$

$$\text{L-KILL-EXT}\quad \frac{}{\Delta \triangleright N \xrightarrow{\mathsf{kill}(n,\lambda)} \Delta \ominus (n,\lambda) \triangleright N} \quad \Delta \vdash n_\lambda : \mathsf{alive}$$

$$\text{L-UNLINK-EXT}\quad \frac{}{\Delta \triangleright N \xrightarrow{\ominus n_\lambda \mapsto m} \Delta \ominus n \leftrightarrow m \triangleright N} \quad \begin{array}{l}\Delta \vdash n_\lambda : \mathsf{alive}\\ \Delta \vdash n \leftrightarrow m\end{array}$$

$$\text{L-LINK-EXT}\quad \frac{}{\Delta \triangleright N \xrightarrow{\oplus n_\lambda \mapsto m} \Delta \oplus n \leftrightarrow m \triangleright N} \quad \begin{array}{l}\Delta \vdash n_\lambda : \mathsf{alive}\\ \Delta \nvdash n \leftrightarrow m\end{array}$$

$$\text{L-VIEW}\quad \frac{}{\Delta \triangleright N \xrightarrow{n_\lambda \succ m} \Delta \triangleright N} \quad \begin{array}{l}\Delta \vdash n_\lambda : \mathsf{alive}\\ \mathsf{and}\ (\Delta_{\mathcal{A}}(m) = \Delta_n(m) \neq 0\\ \mathsf{or}\ \Delta_n(m) = 0)\end{array}$$

Figure 6: Net Rules

$$\text{L-PAR}_{\text{L}}\quad \frac{\Delta \triangleright N \xrightarrow{\alpha} \nu\,\tilde{u}.\Delta' \triangleright N' \quad \tilde{u} \cap \mathsf{fn}(M) = \emptyset}{\Delta \triangleright N \parallel M \xrightarrow{\alpha} \nu\,\tilde{u}.\Delta' \triangleright N' \parallel M}$$

$$\text{L-SYNC}_{\text{L}}\quad \frac{\Delta \triangleright N \xrightarrow{\overline{x}\langle\tilde{u}\rangle@n_\lambda} \Delta \triangleright N' \qquad \Delta \triangleright M \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright M'}{\Delta \triangleright N \parallel M \xrightarrow{\tau} \Delta \triangleright N' \parallel M'}$$

$$\text{L-RES}_{\text{O}}\quad \frac{S \xrightarrow{\nu\,\tilde{v}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} S' \quad w \in \tilde{u} \setminus \tilde{v}, x, n}{\nu\,w.S \xrightarrow{\nu\,w.\nu\,\tilde{v}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} S'}$$

$$\text{L-RES}\quad \frac{S \xrightarrow{\alpha} S' \quad u \notin \mathsf{fn}(\alpha)}{\nu\,u.S \xrightarrow{\alpha} \nu\,u.S'}$$

Figure 7: Composition Rules

branch is required to avoid name capture when the other branch introduces a restriction. Rule L-RES$_{\text{O}}$ is analogous to the classical OPEN rule in the π-calculus. What is unusual is that there is no corresponding CLOSE rule in our LTS semantics, because rule L-RES$_{\text{O}}$ operates at the system level, and we have no operation for composing systems. Rule L-RES$_{\text{O}}$ is a way to signal that a system is ready to send a message at a given address, possibly bearing private names in its payload.

4.3 Full Abstraction

Before presenting the main result of the paper we introduce the definition of *weak bisimilarity.*

Definition 7 (Weak Bisimilarity)**.** *A binary relation over closed systems $\mathcal{S} \subseteq \mathbb{S}^2$ is a* weak simulation *iff whenever $(S,R) \in \mathcal{S}$ and $S \xrightarrow{\alpha} S'$, then $R \stackrel{\alpha}{\Longrightarrow} R'$ for some R' with $(S',R') \in \mathcal{S}$. A binary relation $\mathcal{S}$ over closed systems is a* weak bisimulation *if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are weak simulations.* Weak bisimilarity*, denoted by $\approx$, is the largest weak bisimulation.*

Our main result states that weak bisimilarity fully characterizes weak barbed congruence.

Theorem 1 (Full Abstraction)**.** *$S \dot{\approx} R$ iff $S \approx R$.*

Proof. Details of the proof can be found in Appendix E. □

A few comments are in order. A consequence of our result is that the public part of the network (i.e. those nodes and links whose names are not restricted) of two weak barbed congruent systems must coincide. This is visible directly from the rules in Fig.6. In particular, incarnation numbers of public nodes must coincide. This may seem to be overly discriminative but in fact it is warranted: recovery protocols or failure handling protocols such as SWIM [7], which rely on incarnation numbers, would operate differently in dissimilar systems.

5 Examples

5.1 Deriving Erlang's Like Constructs

This sub-section shows how to implement several Erlang primitives using the ones provided by our calculus. Let us begin with the ping definition, which is used to test for accessibility of a remote location (m) by the current one (n). If the ping succeeds the view of n is updated with the knowledge of m and vice-versa, moreover the ping evaluates to an atom pong to signal success. If it fails n removes any belief it may had about the m and evaluates to pang.

Since in our calculus we do not have atoms we encode a slight variation of the ping that branches according to the result of the test. The successful one mimics the evaluation to pong and the negative to pang.

$$\mathsf{ping}\ m.P\ \mathsf{else}\ Q \stackrel{\text{def}}{=} \nu\, x,t,f.\mathsf{node}(u,\lambda).\left(\begin{array}{l}\overline{x}\langle f\rangle \mid \mathsf{spawn}\ m.(\mathsf{spawn}\ u.\overline{x}\langle t\rangle) \mid \\ x(y).\mathsf{if}\ y=t\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{remove}\ m.Q\end{array}\right)$$

Primitive monitor repeatedly tests for a remote location accessibility and continues as P when the accessibility test fails. Since the ping can nondeterministically fail at any point the monitor too can nondeterministically fail. We encode it as follows.

$$\mathsf{monitor}(m).P \stackrel{\text{def}}{=} \nu\, x.(Q \mid !x.Q) \qquad \text{where} \qquad Q = \mathsf{ping}\ m.\overline{x}\ \mathsf{else}\ P$$

Primitive start creates a new location l, provided that it does not exist already. Location l is said a *slave* node and the creator location is said the *master* node. The slave dies when its master dies, hence we need a monitoring process to implement this behavior. We encode it as follows.

$$\mathsf{start}(l) \stackrel{\text{def}}{=} \mathsf{node}(u,\lambda).\mathsf{create}\ l.(\mathsf{monitor}(u).\mathsf{kill})$$

Primitive stop halts the execution of node m. We encode it as follows.

$$\mathsf{stop}(m) \stackrel{\text{def}}{=} \mathsf{spawn}\ m.\mathsf{kill}$$

Primitive disconnect removes m from the view of n and vice-versa. We encode it as follows.

$$\mathsf{disconnect}(m).P \stackrel{\text{def}}{=} \mathsf{node}(u,\lambda).\mathsf{remove}\ m.(\mathsf{spawn}\ m.(\mathsf{remove}\ u))$$

5.2 Behavioral Theory In Action

We begin this section by applying our behavioral theory to the motivating example in Section 2.

Example 1 (**servD** and **servDFR** are bisimilar.)**.** *To prove* $\boldsymbol{servD} \approx \boldsymbol{servDFR}$ *it suffices to show a candidate bisimulation relation and then play the bisimulation game to its elements. Consider relation* $\mathcal{R} = \{(\boldsymbol{servD}, \boldsymbol{servDFR})\} \cup \mathcal{S}_0 \cup \mathcal{S}_1 \cup \mathcal{S}_2$ *where*

$$\mathcal{S}_0 = \{(\boldsymbol{servD}, R_0) \mid \boldsymbol{servDFR} \stackrel{\tau}{\Longrightarrow} R_0\}$$
$$\mathcal{S}_1 = \{(S_1, R_1) \mid (S_0, R_0) \in \mathcal{S}_0, S_0 \stackrel{req(x,y)@n_i}{\Longrightarrow} S_1 \wedge R_0 \stackrel{req(x,y)@n_i}{\Longrightarrow} R_1\}$$
$$\mathcal{S}_2 = \{(S_2, R_2) \mid (S_1, R_1) \in \mathcal{S}_1, S_1 \stackrel{\overline{z}\langle w_\lambda\rangle@n_i}{\Longrightarrow} S_1 \wedge R_1 \stackrel{\overline{z}\langle w_\lambda\rangle@n_i}{\Longrightarrow} R_2\}$$

Intuitively, from an external perspective, $\boldsymbol{servD}$ *only inputs the request and exhibits the answer; hence, we need to prove that* $\boldsymbol{servDFR}$ *is able to match those two actions. The proof is available in Appendix F.* ◊

We now present two examples inspired from [16] to show that our behavioral theory agrees with the one they present in absence of recovery.

Example 2 (Synchronous moves)**.** *This example declines [16, Example 10] into a version that can nondeterministically fail w.r.t. the original one, due to the fact that we do not have a perfect failure detector. Consider the construct* $\mathsf{move}\ m.P\ \mathsf{else}\ Q$ *which attempts to migrate* P *to* m *from the current location and if it fails, launches* Q *locally.*

Assuming $\Delta \vdash (n, \lambda) : \mathsf{alive}$*, the behavior could be defined as*

$$\text{MOVE}\ \frac{}{\Delta \triangleright [\![\ \mathsf{move}\ m.P\ \mathsf{else}\ Q\ ]\!]^n_\lambda \to \Delta \triangleright [\![\ P\ ]\!]^m_\lambda}\ \frac{\Delta_n(m) = \lambda'}{\Delta \vdash n_\lambda \leftrightsquigarrow m'_\lambda} \qquad \text{NMOVE}\ \frac{}{\Delta \triangleright [\![\ \mathsf{move}\ m.P\ \mathsf{else}\ Q\ ]\!]^n_\lambda \to \Delta \triangleright [\![\ Q\ ]\!]^n_\lambda}$$

Now one could try to implement the move *primitive through the following macro*

$$\mathsf{mv}\ m.P\ \mathsf{else}\ Q \stackrel{def}{=} \nu\, a, b.\mathsf{node}(u, \lambda).(\mathsf{spawn}\ m.(b.P \mid \mathsf{spawn}\ u.a.(\mathsf{spawn}\ m.\overline{b})) \mid \overline{a} \mid \mathsf{monitor}_a m.Q)$$

The above code sends $b.P$ *to the target location and then goes back to signal the successful arrive of* $b.P$*, by synchronizing on the private channel* a *and eventually goes back to release* P*. The implementation uses mutual exclusion on* a *together with the macro* $\mathsf{monitor}_a m.Q$*, which is a slight variation of the one in Section 5.1. This macro acquires the resource* a *before testing and releases it after the success of the test whereas it releases* Q *if* n' *is not accessible. It can be encoded as*

$$\mathsf{monitor}_a m.Q \stackrel{def}{=} \nu\, test.(\overline{test} \mid !test.a.\mathsf{ping}\ m.(\overline{test} \mid \overline{a})\ \mathsf{else}\ Q)$$

We now show that in our setting, as in [16], the primitive move *is not observationally equivalent to* mv*. To prove this we take advantage of the contextuality property of* $\dot{\approx}$ *and show that*

$$\nu\, \tilde{u}.\Delta \triangleright [\![\ \mathsf{move}\ m.P\ \mathsf{else}\ Q\ ]\!]^n_\lambda \parallel [\![\ \mathsf{unlink}\ n\ ]\!]^m_\kappa \not\dot{\approx} \nu\, \tilde{u}.\Delta \triangleright [\![\ \mathsf{mv}\ m.P\ \mathsf{else}\ Q\ ]\!]^n_\lambda \parallel [\![\ \mathsf{unlink}\ n\ ]\!]^m_\kappa$$

where $\Delta = \langle \{n \mapsto \lambda, m \mapsto \kappa\}, \{n \leftrightarrow m)\}, \{n \mapsto \hat{\mathbf{0}}, m \mapsto \hat{\mathbf{0}}\}\rangle$

The intuition is that the right-hand side system can reach a point where $b.P$ *reached successfully* m *while* $\mathsf{spawn}\ u.a.(\mathsf{spawn}\ m.\overline{b}))$ *manages to go back to* n *and also synchronizes on* a*. Now, if at this point the* unlink *reduces we get to a point where* n *and* m *cannot synchronize on* b *hence the two locations are alive but both* P *and* Q *are blocked. Due to the atomicity of the* move *the same state cannot be achieved on the left-hand side.*

◊

Example 3 (Distributed Server)**.** *Here, we rephrase [16, Example 11], where Francalanza and Hennessy show that the behavioral theory they present is able to distinguish a distributed server only able to reach its backend by a direct connection and one that, in addition to the direct connection, has also an indirect connection that goes through a third locality.*

The two systems are the following.

$$\boldsymbol{servFHD} \Leftarrow \nu\, data, w_y.\Delta \rhd \left(\begin{array}{c} [\![\, req(x,y).\mathsf{spawn}\ n'.\overline{data}\langle x,y\rangle \,]\!]^{n} \parallel \\ [\![\, data(x,y).\mathsf{spawn}\ n.\overline{x}\langle w_y\rangle \,]\!]^{n'} \end{array} \right)$$

$$\boldsymbol{sFHD2Rt} \Leftarrow \nu\, data, w_y.\Delta \rhd \left(\begin{array}{l} \left[req(x,y).\nu\, sync. \left(\begin{array}{l} \mathsf{spawn}\ n'.\overline{data}\langle sync, x\rangle \mid \\ \mathsf{spawn}\ n''.\mathsf{spawn}\ n'.\overline{data}\langle sync, x\rangle \mid \\ sync(x).\overline{y}\langle x\rangle \end{array} \right) \right]^{n} \parallel \\ \left[data(x,y). \left(\begin{array}{l} \mathsf{spawn}\ n'.\overline{x}\langle f(y)\rangle \mid \\ \mathsf{spawn}\ n''.\mathsf{spawn}\ n.\overline{x}\langle w_y\rangle \end{array} \right) \right]^{n'} \end{array} \right)$$

Now, ***servD*** *and* ***sFHD2Rt****, like in [16], can be distinguished by the following context*

$$C \equiv \cdot \parallel [\![\, \mathsf{unlink}\ n \,]\!]^{n'} \parallel [\![\, \overline{req}\langle z,h\rangle \,]\!]^{n}$$

as ***servD*** *would stop working after the* break *reduces, while* ***servD2Rt*** *would keep working correctly since it could route the request through* m*.*

$\Diamond$

6 Related work and conclusion

We have presented in this paper a distributed π-calculus with location and link crash failures and recoveries. The calculus basic constructs, including incarnation numbers to distinguish different versions of the same location across recoveries and local imperfect knowledge, were inspired mainly by the Erlang programing language and environment. To the best of our knowledge, this is the first work that combines these different features, and the first to deal with recovery without relying on some form of checkpointing (as in Erlang-style systems).

As mentioned in the Introduction, there are only a few works in the literature proposing a process calculus analysis of distributed systems with crash failures and recoveries. We discussed the work by Berger and Honda [3] and by Bocchi et al. [4] in the Introduction. The work by Fournet et al. on the join calculus [15] shows how to extend the join calculus with primitives for crash failures and recoveries but does not take into account links and link failures and does not present a behavioral theory for these extensions of the join calculus. The work by Amadio [1] on the π_{1l}-calculus presents an asynchronous π-calculus with unique receivers, located processes, and location failures. It develops a behavioral theory for this calculus by translation of the π_{1l}-calculus into the π_1 calculus (an asynchronous π-calculus with unique receivers), but it relies on perfect failure detectors, does not support links and link failures, and recovery just consists in restarting a stopped process in its exact state at the moment of failure. In its discussion of weaker failure detectors, it does indicate how an extension could support a local view (of failed locations) but it does not elaborate the corresponding calculus and behavioral theory.

Our main inspiration for this paper was the work by Francalanza and Hennessy [16] for their handling of node and link failures, and their behavioral theory. Apart from dealing with recovery, which they do not consider, our development is markedly different from theirs. We have opted for a simpler handling of scope extrusion, with simpler labels in our LTS semantics (just names instead of complex information about the network, including links and liveness of locations),

and no need to make explicit the partial view of a network available to an observer. In effect, we have opted for the alternative design choice they discussed when contrasting their work to that of De Nicola et al. [25]: keep simple labels in the LTS and opt for a classical handling of scope extrusion, possibly at the expense of larger bisimulation relations. We believe the gain in simplicity of exposition and understanding, coupled with the simple handling of recovery our approach allows, is well worth it. Also, we Introduce explicit local views, which correspond to the belief that a location has of its neighbours and their current incarnation. Our local views are handled similarly as in Erlang – in particular they may not reflect the current state of the network –, and they have no equivalent in [16]. The notion of partial views for observers in [16] is different: it is a way to filter out information contained in their complex labels so as to obtain full abstraction, and it plays no role in the operational semantics of their model.

Formal models for distributed systems with failures can also be found in recent verification tools for distributed algorithms and distributed systems such as Disel [27], Gobra [30], Perennial [5], Psync [8], TLC [18], Verdi [29]. They can either rely on a specific language (such as Gobra, for Go programs), or domain specific languages for formally specifying algorithms (such as PSync or Disel), or be more general purpose, relying on a mixture of logic and more operational models (such as Perennial, Verdi or TLC). Verdi in particular is interesting for it supports a variety of failure models, including a model of crash failures and recoveries which is quite close to ours, and makes use of simulation relations in its proof techniques. However, to the best of our knowledge, they (Verdi included) do not provide as we do a compositional theory of system equivalence in presence of crash failures and recoveries.

The calculus presented in this paper provides us with a basis for further studies. It would certainly be interesting to further expand it to cater for other failure models, including the kinds of grey failures tackled by Bocchi et al. [4], probably making it parametric in failure models along the lines of Verdi [29]. In its current form, we think it is close enough to the behavior of Erlang systems, and its primitives sufficient to account for a significant subset of Erlang failure handling constructs, to allow us to study reversible debugging for distributed Erlang systems with failures and recoveries, building on the substantial amount of work developed in the past ten years around reversibility and reversible debugging in Erlang [22, 23].

References

[1] Roberto M. Amadio. An asynchronous model of locality, failure and process mobility. In David Garlan and Daniel Le Métayer, editors, *Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings*, volume 1282 of *Lecture Notes in Computer Science*, pages 374–391. Springer, 1997.

[2] Joe Armstrong. *Programming Erlang – Software for a Concurrent World.* The Pragmatic Bookshelf, 2007.

[3] Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended pi-calculus. In Luca Aceto and Björn Victor, editors, *7th International Workshop on Expressiveness in Concurrency, EXPRESS 2000, Satellite Workshop of CONCUR 2000, State College, PA, USA, August 21, 2000*, volume 39 of *Electronic Notes in Theoretical Computer Science*, pages 21–46. Elsevier, 2000.

[4] Laura Bocchi, Julien Lange, Simon Thompson, and A. Laura Voinea. A model of actors and grey failures. In Maurice H. ter Beek and Marjan Sirjani, editors, *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13271 of *Lecture Notes in Computer Science*, pages 140–158. Springer, 2022.

[5] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 243–258. ACM, 2019.

[6] Om P. Damani, Ashis Tarafdar, and Vijay K. Garg. Optimistic recovery in multi-threaded distributed systems. In *The Eighteenth Symposium on Reliable Distributed Systems, SRDS 1999, Lausanne, Switzerland, October 19-22, 1999, Proceedings*, pages 234–243. IEEE Computer Society, 1999.

[7] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 303–312. IEEE Computer Society, 2002.

[8] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 400–415. ACM, 2016.

[9] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[10] E. N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Computers*, 41(5):526–531, 1992.

[11] URL: `https://www.erlang.org/doc/apps/erts/erl_ext_dist.html#NEW_PID_EXT`.

[12] URL: `erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html`, 2019.

[13] URL: `https://github.com/gfabbretti8/erlang-experiments`, 2023.

[14] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 372–385. ACM Press, 1996.

[15] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 1996.

[16] Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failure. *Inf. Comput.*, 206(6):711–759, 2008.

[17] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. An empirical study on crash recovery bugs in large-scale distributed systems. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 539–550. ACM, 2018.

[18] Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. TLC: temporal logic of distributed components. *Proc. ACM Program. Lang.*, 4(ICFP):123:1–123:30, 2020.

[19] Indranil Gupta, Tushar Deepak Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. In Ajay D. Kshemkalyani and Nir Shavit, editors, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA, August 26-29, 2001*, pages 170–179. ACM, 2001.

[20] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.

[21] Sasa Juric. *Elixir in Action*. Manning, 2015.

[22] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71 – 97, 2018.

[23] Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay reversible semantics for message passing concurrent programs. *Fundam. Informaticae*, 178(3):229–266, 2021.

[24] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.

[25] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Basic observables for a calculus for global computing. *Inf. Comput.*, 205(10):1491–1525, 2007.

[26] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001.

[27] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018.

[28] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[29] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015.

[30] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2021.

A Notations and rules

For ease of reference, we gather in this section all the notations and inference rules used in the paper.

A.1 Notations

We assume given mutually disjoint infinite denumerable sets C, N and I. C is the set of *channel name*s, N is the set of *location names*, and I is the set of incarnation number variables. We use the set of integers $\mathbb{Z}$ as the set of *incarnation numbers*. An incarnation number is paired with a location name for recovery purposes, to distinguish the current instance of a location from its past failed instances. We denote by $\mathsf{N}^{\odot}$ the set $\mathsf{N} \cup \{\odot\}$, where $\odot \notin \mathsf{N}$. As in the π-calculus, channel names can be free or bound in terms. The same holds for location names. Incarnation variables can be bound, but not incarnation numbers. We denote by $\tilde{u}$ a finite (possibly empty) tuple of elements. We write $T\{\tilde{v}/\tilde{u}\}$ for the usual capture-avoiding substitution of elements of $\tilde{u}$ by elements of $\tilde{v}$ in term T, assuming tuples $\tilde{u}$ and $\tilde{v}$ have the same arity. We write $u, \tilde{v}$ or $\tilde{v}, u$ for the tuple $\tilde{v}$ extended with element u as first or last element. Abusing notation, we sometimes identify a tuple $\tilde{u}$ with the set of its elements. We denote by $\mathbb{N}^+$ the set of strictly positive integers (by definition $0 \notin \mathbb{N}^+$), and by $\mathbb{N}$ the set of positive integers ($0 \in \mathbb{N}$). We denote by $\hat{\mathbf{0}}$ the function $\hat{\mathbf{0}} : \mathsf{N}^{\odot} \to \mathbb{Z}$ that maps any $n \in \mathsf{N}$ to 0 and $\odot$ to 1.

A.2 Calculus syntax and alpha-conversion on systems

$$
\begin{array}{rl}
P, Q ::= & \mathbf{0} \mid \overline{x}\langle\tilde{u}\rangle.P \mid x(\tilde{v}).P \mid !x(\tilde{v}).P \mid \nu\, w.P \mid \mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid P \mid Q \\
& \mathsf{node}(n, \lambda).P \mid \mathsf{remove}\, n.P \mid \mathsf{spawn}\, n.P \mid \mathsf{kill} \mid \\
& \mathsf{create}\, n.P \mid \mathsf{link}\, n.P \mid \mathsf{unlink}\, n.P \\
L, M, N ::= & \mathbf{0} \mid [\![\, P \,]\!]^n_\lambda \mid N \parallel M \\
S, R ::= & \Delta \triangleright N \mid \nu\, w.S
\end{array}
$$

where: $\tilde{u}, r, s \subset \mathsf{C} \cup \mathsf{N}^{\odot} \cup \mathsf{I} \cup \mathbb{Z}$ $\quad \tilde{v} \subset \mathsf{C} \cup \mathsf{N} \cup \mathsf{I}$ $\quad w \in \mathsf{C} \cup \mathsf{N}$ $\quad x \in \mathsf{C}$ $\quad n \in \mathsf{N}$ $\quad \lambda \in \mathsf{I}$

Free names in processes and configurations are defined inductively as follows:

$$
\begin{array}{ll}
\mathsf{fn}(\mathbf{0}) = \emptyset & \\
\mathsf{fn}(\overline{x}\langle\tilde{u}\rangle.P) = \tilde{u} \cup \{x\} \cup \mathsf{fn}(P) & \\
\mathsf{fn}(x(\tilde{v}).P) = \{x\} \cup \mathsf{fn}(P) \setminus \tilde{v} & \\
\mathsf{fn}(!x(\tilde{v}).P) = \{x\} \cup \mathsf{fn}(P) \setminus \tilde{v} & \\
\mathsf{fn}(\nu\, w.P) = \mathsf{fn}(P) \setminus \{w\} & \\
\mathsf{fn}(\mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q) \cup \{r, s\} & \\
\mathsf{fn}(P \mid Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q) & \\
\mathsf{fn}(\mathsf{node}(n, \lambda).P) = \mathsf{fn}(P) \setminus \{n\} & \\
\mathsf{fn}(\mathsf{remove}\, n.P) = \mathsf{fn}(P) \cup \{n\} & \\
\mathsf{fn}(\mathsf{spawn}\, n.P) = \mathsf{fn}(P) \cup \{n\} & \mathsf{fn}(\mathsf{kill}) = \emptyset \\
\mathsf{fn}(\mathsf{create}\, n.P) = \mathsf{fn}(P) \cup \{n\} & \mathsf{fn}(\mathsf{link}\, n) = \mathsf{fn}(\mathsf{unlink}\, n) = \{n\} \\
\mathsf{fn}([\![\, P \,]\!]^n_\lambda) = \mathsf{fn}(P) \cup \{n\} & \\
\mathsf{fn}(N \parallel M) = \mathsf{fn}(N) \cup \mathsf{fn}(M) &
\end{array}
$$

$\mathbb{L}$ denotes the set of configurations. $\mathbb{S}$ denotes the set of systems. A *network* Δ is a tuple $\langle \mathcal{A}, \mathcal{L}, \mathcal{V} \rangle$ where:

- $\mathcal{A}$ is a function $\mathcal{A} : \mathsf{N}^{\odot} \to \mathbb{Z}$ such that $\mathcal{A}(\odot) = 1$ and such that the set $\mathsf{supp}(\mathcal{A}) \stackrel{\text{def}}{=} \{n \in \mathsf{N} \mid \mathcal{A}(n) \neq 0\}$ is finite. Function $\mathcal{A}$ records three types of information on locations. If $\mathcal{A}(n) = \lambda \in \mathbb{N}^+$, then location n is alive and its current incarnation number is λ. If $\mathcal{A}(n) = -\lambda, \lambda \in \mathbb{N}^+$ then location n has been killed and its last incarnation number while alive was λ. If $\mathcal{A}(n) = 0$, then there is no location n in the network, alive or not.
- $\mathcal{L} \subseteq \mathsf{N}^{\odot} \times \mathsf{N}^{\odot}$ is the set of links between locations. $\mathcal{L}$ is a finite symmetric binary relation over location names such that $\mathsf{dom}(\mathcal{L}) \stackrel{\text{def}}{=} \{n \in \mathsf{N}^{\odot} \mid \exists m, (n, m) \in \mathcal{L}\}$ is finite.
- $\mathcal{V} : \mathsf{N}^{\odot} \to (\mathsf{N}^{\odot} \to \mathbb{N})$ is a function that maps location names to their *local view* which is such that the set $\mathsf{supp}(\mathcal{V}) \stackrel{\text{def}}{=} \{n \in \mathsf{N} \mid \mathcal{V}(n) \neq \hat{\mathbf{0}}\}$ is finite and $\mathsf{supp}(\mathcal{V}) \subseteq \mathsf{supp}(\mathcal{A})$. The local view of a location n is a function $\mathcal{V}(n) : \mathsf{N}^{\odot} \to \mathbb{N}$ such that the set $\{m \in \mathsf{N} \mid \mathcal{V}(n)(m) \neq 0\}$ is finite. If $\mathcal{V}(n)(m) = \kappa \in \mathbb{N}^+$, then location m in its incarnation κ is believed by n to be alive. If $\mathcal{V}(n)(m) = 0$, then location n holds no belief on the status of location m.

$\Delta_{\mathcal{A}}$, $\Delta_{\mathcal{L}}$, and $\Delta_{\mathcal{V}}$ denote the components of a network representation Δ. Notations for extracting information from Δ:

- $\Delta \vdash n_{\lambda} : \mathsf{alive}$ if $\Delta_{\mathcal{A}}(n) = \lambda$ and $\lambda \in \mathbb{N}^+$.
- $\Delta \vdash n : \mathsf{dead}$ if $\Delta_{\mathcal{A}}(n) \notin \mathbb{N}^+$
- $\Delta \vdash n \leftrightarrow m$ if $(n, m) \in \Delta_{\mathcal{L}}$
- $\Delta \vdash n_{\lambda} \rightsquigarrow m_{\kappa}$ if $(n, m) \in \Delta_{\mathcal{L}}$, $\Delta \vdash n_{\lambda} : \mathsf{alive}$ and $\Delta \vdash m_{\kappa} : \mathsf{alive}$

Update operations over a network Δ:

- $\Delta \oplus n \leftrightarrow m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}} \cup \{(n, m), (m, n)\}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \ominus n \leftrightarrow m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}} \setminus \{(n, m), (m, n)\}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \oplus (n, \lambda) = \langle \Delta_{\mathcal{A}}[n \mapsto \lambda], \Delta_{\mathcal{L}} \cup \{(n, n)\}, \Delta_{\mathcal{V}}[n \mapsto \hat{\mathbf{0}}] \rangle$
- $\Delta \ominus (n, \lambda) = \langle \Delta_{\mathcal{A}}[n \mapsto -\lambda], \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \oplus n \succ (m, \lambda) = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}}[n \mapsto \Delta_{\mathcal{V}}(n)[m \mapsto \lambda]] \rangle$, if $n \neq m$
- $\Delta \ominus n \succ m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}}[n \mapsto \Delta_{\mathcal{V}}(n)[m \mapsto 0]] \rangle$, if $n \neq m$
- $\Delta \ominus n \succ n = \Delta \oplus n \succ (n, \lambda) = \Delta$

Free names of networks and systems are defined inductively as follows:

$$\mathsf{fn}(\Delta) = \mathsf{supp}(\Delta_{\mathcal{A}}) \cup \mathsf{dom}(\Delta_{\mathcal{L}})$$
$$\mathsf{fn}(\Delta \triangleright N) = \mathsf{fn}(\Delta) \cup \mathsf{fn}(N)$$
$$\mathsf{fn}(\nu\, \tilde{u}.S) = \mathsf{fn}(S) \setminus \tilde{u}$$

To define alpha-conversion on systems, we define capture-avoiding substitution on networks. A capture avoiding substitution $\{v/u\}$ on network Δ is defined as follows: if $\Delta = (\mathcal{A}, \mathcal{L}, \mathcal{V})$, then $\Delta\{v/u\} = (\mathcal{A}\{v/u\}, \mathcal{L}\{v/u\}, \mathcal{V}\{v/u\})$ where:

$$\mathcal{A}\{v/u\} = \begin{cases} \mathcal{A}[v \mapsto \mathcal{A}(u)][u \mapsto 0] & \text{if} \quad u, v \in \mathsf{N} \text{ and } v \notin \mathsf{supp}(\mathcal{A}) \\ \mathcal{A} & \text{otherwise} \end{cases}$$

$$\mathcal{L}\{v/u\} = \begin{cases} (\mathcal{L} \setminus \mathcal{L}(u)) \cup \mathcal{L}(u)\{v/u\} & \text{if} \quad u, v \in \mathsf{N} \text{ and } v \notin \mathsf{supp}(\mathcal{A}) \\ \mathcal{L} & \text{otherwise} \end{cases}$$

$$\mathcal{V}\{v/u\} = \begin{cases} \mathcal{V}[v \mapsto \mathcal{V}(u)][u \mapsto \hat{\mathbf{0}}] & \text{if} \quad u, v \in \mathsf{N} \text{ and } v \notin \mathsf{supp}(\mathcal{A}) \\ \mathcal{V} & \text{otherwise} \end{cases}$$

with:

$$\mathcal{L}(n) = \{(a, b) \in \mathcal{L} \mid a = n \text{ or } b = n\}$$
$$\mathcal{L}(n)\{m/n\} = \{(a\{m/n\}, b\{m/n\}) \mid (a, b) \in \mathcal{L}(n)\}$$
$$a\{m/n\} = \begin{cases} m & \text{if} \quad a = n \\ a & \text{otherwise} \end{cases}$$

The effect of a capture avoiding substitution $\{v/u\}$ on a system $\Delta \triangleright N$ is defined inductively as follows. We define the set occ of name occurrences of a system as follows:

$$(\Delta \triangleright N)\{v/u\} = \Delta\{v/u\} \triangleright N\{v/u\}$$
$$(\nu\, w.S)\{v/u\} = \begin{cases} \nu\, w.S\{v/u\} & \text{if } u, v \neq w \\ \bot & \text{otherwise} \end{cases}$$

Equality modulo alpha-conversion $=_\alpha$ on systems is now defined as the smallest equivalence relation on systems defined by the following rules:

$$\frac{S\{v/u\} \neq \bot}{\nu\, u.S =_\alpha \nu\, v.S\{v/u\}} \qquad \frac{S =_\alpha T}{\nu\, u.S =_\alpha \nu\, u.T}$$

A.3 Reduction semantics

The operational semantics of our calculus is defined via a reduction semantics given by a binary relation $\longrightarrow \subseteq \mathbb{S} \times \mathbb{S}$ between closed systems, and a structural congruence relation $\equiv \subseteq \mathbb{S}^2 \cup \mathbb{L}^2$, that is a binary equivalence relation between systems and between configurations. Evaluation contexts are "systems with a hole $\cdot$" defined by the following grammar:

$$\mathbb{C} \;::=\; \nu\, \tilde{w}.\Delta \triangleright \mathbb{E} \qquad \mathbb{E} \;::=\; \cdot \;\big|\; (N \parallel \mathbb{E}) \qquad \text{where:} \qquad \tilde{w} \subset \mathsf{N} \cup \mathsf{C}$$

Relation $\equiv$ is the smallest equivalence relation defined by the rules in Fig. 8, where $=_\alpha$ stands for equality up to alpha-conversion, $M, N, L \in \mathbb{L}$, and $S, R \in \mathbb{S}$.

Reduction relation $\longrightarrow$ is defined by the rules in Fig. 9, where:

$$\Delta_n(m) = \begin{cases} \kappa & \text{if } n = m \text{ and } \Delta_{\mathcal{A}}(n) = \kappa \\ \kappa & \text{if } n \neq m \text{ and } \Delta_{\mathcal{V}}(n)(m) = \kappa \\ \kappa & \text{if } n \neq m \text{ and } \Delta_{\mathcal{V}}(n)(m) = 0 \text{ and } \Delta_{\mathcal{A}}(m) = \kappa \\ 0 & \text{otherwise} \end{cases}$$

$$[\text{S.Par.C}]\ N \parallel M \equiv M \parallel N \qquad [\text{S.Par.A}]\ (L \parallel M) \parallel N \equiv L \parallel (M \parallel N) \qquad [\text{S.Par.N}]\ (N \parallel \mathbf{0}) \equiv N$$

$$[\text{S.Res.C}]\ \nu\, u.\nu\, v.S \equiv \nu\, v.\nu\, u.S \qquad [\text{S.Res.Nil}]\ \frac{}{\nu\, u.S \equiv S}\ u \notin \mathsf{fn}(S) \qquad [\text{S.}\alpha]\ \frac{S =_\alpha R}{S \equiv R}$$

$$[\text{S.Ctx}]\ \frac{N \equiv M}{\mathbb{C}[N] \equiv \mathbb{C}[M]}$$

Figure 8: Structural Congruence Rules

A.4 LTS semantics

Labels α in our LTS semantics take the following forms:

$$\alpha ::= \quad \tau \mid \nu\, \tilde{w}.\overline{x}\langle \tilde{u} \rangle @ n_\lambda \mid x(\tilde{u})@n_\lambda \mid \mathsf{kill}(n,\lambda) \mid \mathsf{create}(n,\lambda) \mid \oplus\, n_\lambda \mapsto m \mid \ominus\, n_\lambda \mapsto m \mid n_\lambda \succ m$$

Free names in labels are defined as follows:

$$\mathsf{fn}(\tau) = \emptyset \qquad \mathsf{fn}(\nu\, \widetilde{w}.\overline{x}\langle \tilde{u} \rangle @ n_\lambda) = (\tilde{u} \cup \{x, n\}) \setminus \widetilde{w} \qquad \mathsf{fn}(x(\tilde{u})@n_\lambda) = \tilde{u} \cup \{x, n\}$$
$$\mathsf{fn}(\mathsf{kill}(n,\lambda)) = \{n\} \qquad \mathsf{fn}(\mathsf{create}(n,\lambda)) = \{n\}$$
$$\mathsf{fn}(\oplus n_\lambda \mapsto m) = \{n, m\} \qquad \mathsf{fn}(\ominus n_\lambda \mapsto m) = \{n, m\} \qquad \mathsf{fn}(n_\lambda \succ m) = \{n, m\}$$

Labelled transition relations $\xrightarrow{\alpha}$ of our LTS semantics are defined by the rules in Fig. 10 and 11.

Assuming $\Delta \vdash n_\lambda : \mathsf{alive}$

MSG
$$\Delta \triangleright [\,\overline{x}\langle\tilde{v}\rangle.Q\,]^n_\lambda \parallel [\,x(\tilde{u}).P\,]^n_\lambda \longrightarrow \Delta \triangleright [\,P\{\tilde{v}/\tilde{u}\}\,]^n_\lambda \parallel [\,Q\,]^n_\lambda$$

NEW
$$\Delta \triangleright [\,\nu\, u.P\,]^n_\lambda \longrightarrow \nu\, u.\Delta \triangleright [\,P\,]^n_\lambda \quad u \notin \mathsf{fn}(\Delta) \cup \{n, \lambda\}$$

IF-NEQ
$$\Delta \triangleright [\,\mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q\,]^n_\lambda \longrightarrow \Delta \triangleright [\,Q\,]^n_\lambda \quad r \neq s$$

IF-EQ
$$\Delta \triangleright [\,\mathsf{if}\ r = r\ \mathsf{then}\ P\ \mathsf{else}\ Q\,]^n_\lambda \longrightarrow \Delta \triangleright [\,P\,]^n_\lambda$$

SPAWN-S
$$\frac{\Delta_n(m) = \kappa \qquad \Delta \vdash n_\lambda \leftrightsquigarrow m_\kappa}{\Delta \triangleright [\,\mathsf{spawn}\ m.P\,]^n_\lambda \longrightarrow \Delta \oplus m \succ (n, \lambda) \triangleright [\,P\,]^m_\kappa}$$

SPAWN-F
$$\frac{\Delta_n(m) = \kappa \qquad \Delta \not\vdash n_\lambda \leftrightsquigarrow m_\kappa}{\Delta \triangleright [\,\mathsf{spawn}\ m.P\,]^n_\lambda \longrightarrow \Delta \ominus n \succ m \triangleright \mathbf{0}}$$

CREATE-S
$$\frac{\Delta \vdash m : \mathsf{dead} \qquad \Delta_{\mathcal{A}}(m) = -\kappa}{\Delta \triangleright [\,\mathsf{create}\ m.P\,]^n_\lambda \longrightarrow \Delta \oplus (m, \kappa + 1) \triangleright [\,P\,]^m_{\kappa+1}}$$

CREATE-F
$$\frac{\Delta \not\vdash m : \mathsf{dead}}{\Delta \triangleright [\,\mathsf{create}\ m.P\,]^n_\lambda \longrightarrow \Delta \triangleright \mathbf{0}}$$

RES
$$\frac{S \longrightarrow S'}{\nu\, u.S \longrightarrow \nu\, u.S'}$$

BANG
$$\Delta \triangleright [\,!x(\tilde{u}).P\,]^n_\lambda \longrightarrow \Delta \triangleright [\,x(\tilde{u}).(P \mid\ !x(\tilde{u}).P)\,]^n_\lambda$$

FORK
$$\Delta \triangleright [\,P \mid Q\,]^n_\lambda \longrightarrow \Delta \triangleright [\,P\,]^n_\lambda \parallel [\,Q\,]^n_\lambda$$

NODE
$$\Delta \triangleright [\,\mathsf{node}(m, \kappa).P\,]^n_\lambda \longrightarrow \Delta \triangleright [\,P\{m, \kappa/n, \lambda\}\,]^n_\lambda$$

REMOVE
$$\Delta \triangleright [\,\mathsf{remove}\ m.P\,]^n_\lambda \longrightarrow \Delta \ominus n \succ m \triangleright [\,P\,]^n_\lambda$$

LINK
$$\frac{\Delta \not\vdash n \leftrightarrow m}{\Delta \triangleright [\,\mathsf{link}\ m.P\,]^n_\lambda \longrightarrow \Delta \oplus n \leftrightarrow m \triangleright [\,P\,]^n_\lambda}$$

UNLINK
$$\frac{\Delta \vdash n \leftrightarrow m}{\Delta \triangleright [\,\mathsf{unlink}\ m.P\,]^n_\lambda \longrightarrow \Delta \ominus n_\lambda \leftrightarrow m \triangleright [\,P\,]^n_\lambda}$$

KILL
$$\frac{n \neq \odot}{\Delta \triangleright [\,\mathsf{kill}\,]^n_\lambda \longrightarrow \Delta \ominus (n, \lambda) \triangleright \mathbf{0}}$$

PAR
$$\frac{\Delta \triangleright N \longrightarrow \nu\, \tilde{u}.\Delta' \triangleright N' \quad \tilde{u} \cap \mathsf{fn}(M) = \emptyset}{\Delta \triangleright N \parallel M \longrightarrow \nu\, \tilde{u}.\Delta' \triangleright N' \parallel M}$$

STR
$$\frac{S \equiv S' \quad S' \longrightarrow R' \quad R' \equiv R}{S \longrightarrow R}$$

Figure 9: Reduction Rules

Assuming $\Delta \vdash n_\lambda : \mathsf{alive}$

L-BANG

$$\Delta \triangleright [\, !x(\tilde{u}).P \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright [\, x(\tilde{u}).(P \mid !x(\tilde{u}).P) \,]^n_\lambda$$

L-FORK

$$\Delta \triangleright [\, P \mid Q \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright [\, P \,]^n_\lambda \parallel [\, Q \,]^n_\lambda$$

L-NEW

$$\frac{u \notin \mathsf{fn}(\Delta) \cup \{n, \lambda\}}{\Delta \triangleright [\, \nu\, u.P \,]^n_\lambda \xrightarrow{\tau} \nu\, u.\Delta \triangleright [\, P \,]^n_\lambda}$$

L-NODE

$$\Delta \triangleright [\, \mathsf{node}(m, \kappa).P \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright [\, P\{m, \kappa/n, \lambda\} \,]^n_\lambda$$

L-IF-NEQ

$$\frac{r \neq s}{\Delta \triangleright [\, \mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright [\, Q \,]^n_\lambda}$$

L-REMOVE

$$\Delta \triangleright [\, \mathsf{remove}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \ominus n \succ m \triangleright [\, P \,]^n_\lambda$$

L-IF-EQ

$$\Delta \triangleright [\, \mathsf{if}\ r = r\ \mathsf{then}\ P\ \mathsf{else}\ Q \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright [\, P \,]^n_\lambda$$

KILL

$$\frac{n \neq \odot}{\Delta \triangleright [\, \mathsf{kill} \,]^n_\lambda \xrightarrow{\tau} \Delta \ominus (n, \lambda) \triangleright \mathbf{0}}$$

L-IN

$$\Delta \triangleright [\, x(\tilde{v}).P \,]^n_\lambda \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright [\, P\{\tilde{u}/\tilde{v}\} \,]^n_\lambda$$

L-OUT

$$\Delta \triangleright [\, \overline{x}\langle \tilde{v} \rangle.P \,]^n_\lambda \xrightarrow{\overline{x}\langle \tilde{v} \rangle@n_\lambda} \Delta \triangleright [\, P \,]^n_\lambda$$

L-SPAWN-S

$$\frac{\Delta_n(m) = \kappa \qquad \Delta \vdash n_\lambda \leftrightsquigarrow m_\kappa}{\Delta \triangleright [\, \mathsf{spawn}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \oplus m \succ (n, \lambda) \triangleright [\, P \,]^m_\kappa}$$

L-LINK

$$\frac{\Delta \nvdash n \leftrightarrow m}{\Delta \triangleright [\, \mathsf{link}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \oplus n \leftrightarrow m \triangleright [\, P \,]^n_\lambda}$$

L-SPAWN-F

$$\frac{\Delta_n(m) = \kappa \qquad \Delta \nvdash n_\lambda \leftrightsquigarrow m_\kappa}{\Delta \triangleright [\, \mathsf{spawn}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \ominus n \succ m \triangleright \mathbf{0}}$$

L-UNLINK

$$\frac{\Delta \vdash n \leftrightarrow m}{\Delta \triangleright [\, \mathsf{unlink}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \ominus n \leftrightarrow m \triangleright [\, P \,]^n_\lambda}$$

L-CREATE-S

$$\frac{\Delta \vdash m : \mathsf{dead} \qquad \Delta_{\mathcal{A}}(m) = -\kappa}{\Delta \triangleright [\, \mathsf{create}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \oplus (m, \kappa + 1) \triangleright [\, P \,]^m_{\kappa+1}}$$

L-CREATE-F

$$\frac{\Delta \nvdash m : \mathsf{dead}}{\Delta \triangleright [\, \mathsf{create}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright \mathbf{0}}$$

Figure 10: LTS Rules (1)

L-CREATE-EXT
$$\frac{\Delta \vdash n : \mathsf{dead} \qquad \Delta_{\mathcal{A}}(n) = -\kappa}{\Delta \triangleright N \xrightarrow{\mathsf{create}(n,\kappa+1)} \Delta \oplus (n, \kappa+1) \triangleright N}$$

L-KILL-EXT
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive}}{\Delta \triangleright N \xrightarrow{\mathsf{kill}(n,\lambda)} \Delta \ominus (n, \lambda) \triangleright N}$$

L-UNLINK-EXT
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \vdash n \leftrightarrow m}{\Delta \triangleright N \xrightarrow{\ominus n_\lambda \mapsto m} \Delta \ominus n \leftrightarrow m \triangleright N}$$

L-LINK-EXT
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \nvdash n \leftrightarrow m}{\Delta \triangleright N \xrightarrow{\oplus n_\lambda \mapsto m} \Delta \oplus n \leftrightarrow m \triangleright N}$$

L-VIEW
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \mathsf{and}\ (\Delta_{\mathcal{A}}(m) = \Delta_n(m) \neq 0 \qquad \mathsf{or}\ \Delta_n(m) = 0)}{\Delta \triangleright N \xrightarrow{n_\lambda \succ m} \Delta \triangleright N}$$

L-RES
$$\frac{S \xrightarrow{\alpha} S' \quad u \notin \mathsf{fn}(\alpha)}{\nu\, u.S \xrightarrow{\alpha} \nu\, u.S'}$$

L-PAR$_{\mathrm{L}}$
$$\frac{\Delta \triangleright N \xrightarrow{\alpha} \nu\, \tilde{u}.\Delta' \triangleright N' \quad \tilde{u} \cap \mathsf{fn}(M) = \emptyset}{\Delta \triangleright N \parallel M \xrightarrow{\alpha} \nu\, \tilde{u}.\Delta' \triangleright N' \parallel M}$$

L-PAR$_{\mathrm{R}}$
$$\frac{\Delta \triangleright N \xrightarrow{\alpha} \nu\, \tilde{u}.\Delta' \triangleright N' \quad \tilde{u} \cap \mathsf{fn}(M) = \emptyset}{\Delta \triangleright M \parallel N \xrightarrow{\alpha} \nu\, \tilde{u}.\Delta' \triangleright M \parallel N'}$$

L-SYNC$_{\mathrm{L}}$
$$\frac{\Delta \triangleright N \xrightarrow{\overline{x}\langle \tilde{u} \rangle @ n_\lambda} \Delta \triangleright N' \qquad \Delta \triangleright M \xrightarrow{x(\tilde{u}) @ n_\lambda} \Delta \triangleright M'}{\Delta \triangleright N \parallel M \xrightarrow{\tau} \Delta \triangleright N' \parallel M'}$$

L-RES$_{\mathrm{O}}$
$$\frac{S \xrightarrow{\nu\, \tilde{v}.\overline{x}\langle \tilde{u} \rangle @ n_\lambda} S' \quad w \in \tilde{u} \setminus \tilde{v}, x, n}{\nu\, w.S \xrightarrow{\nu\, w.\nu\, \tilde{v}.\overline{x}\langle \tilde{v} \rangle @ n_\lambda} \nu\, S.'}$$

L-SYNC$_{\mathrm{R}}$
$$\frac{\Delta \triangleright N \xrightarrow{\overline{x}\langle \tilde{u} \rangle @ n_\lambda} \Delta \triangleright N' \qquad \Delta \triangleright M \xrightarrow{x(\tilde{u}) @ n_\lambda} \Delta \triangleright M'}{\Delta \triangleright M \parallel N \xrightarrow{\tau} \Delta \triangleright M' \parallel N'}$$

L-α
$$\frac{S =_\alpha T \qquad T \xrightarrow{\alpha} T' \qquad T' =_\alpha S'}{S \xrightarrow{\alpha} S'}$$

Figure 11: LTS Rules (2)

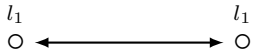
B Erlang

B.1 Experiments on Erlang's Semantics

In this section we will show a typical distributed Erlang behaviors. To simulate a distributed environment Erlang nodes will be running on dockers' container connected through a network. By using docker facilities we will simulate failure and changes in the network structure. The code of this examples can be found at [13].

Pinging a reincarnation of a previous known location

Description. In this scenario l_2 successfully contacts l_1, thus establishing a connection. Then, l_1 fails and recovers. Finally, l_2, before detecting that the incarnation of l_1 to which it was connected has failed, tests again its accessibility. Graphically, the network could be represented as follow.



The following commands set up the configuration and attach a remote shell to l_2.

```
gfabbret@ubuntu:~/$ docker-compose up -d
gfabbret@ubuntu:~/$ docker exec -it l2.com erl -name test@l2.com
                    -setcookie cookie -remsh app@l2.com -hidden
```

Then, to establish a connection between the two locations in the Erlang console we can ping the remote location.

```
(app@l2.com)2> net_adm:ping('app@l1.com').
pong
```

To induce the fault and make it seem like a genuine interruption of services, without third parties being notified, we detach the container from all the networks, restart it, and reconnect it to the networks it was connected to. The disconnection is required as otherwise the other containers would be notified of the restart. Detaching and re-attaching the container to the network takes few milliseconds, hence not enough for the Erlang system to detect it. To do so we get the container and network ids through the docker commands and we feed them to the restart script.

```
gfabbret@ubuntu:~/scenario$ docker container ls
CONTAINER ID   IMAGE          COMMAND     CREATED          STATUS         NAMES
ba5915332d87   erlang:25.0.3  "erl ..."   49 seconds ago   Up 49 seconds  l2.com
76c89a08761c   erlang:25.0.3  "erl ..."   49 seconds ago   Up 49 seconds  l1.com
gfabbret@ubuntu:~/scenario-4$ docker network ls
NETWORK ID     NAME                              DRIVER    SCOPE
fec145052743   bridge                            bridge    local
1b349490c51c   host                              host      local
373b2aa1c5e9   none                              null      local
cfc05f57eeee   scenario_net1                    bridge    local
gfabbret@ubuntu:~/scenario$ ./restart.sh cfc05f57eeee 76c89a08761c
ba5915332d87
done
```

Finally we attempt to ping the remote location after the restarting.

```
...
(app@l2.com)3> spawn('app@l1.com', fun() -> self() end).
<0.107.0>
(app@l2.com)4> =WARNING REPORT==== 2-May-2023::15:16:02.174308 ===
** Can not start erlang:apply,[#Fun<erl_eval.43.3316493>,[]] on 'app@l1.com' **
```

The test for spawn failed even if there is a live running instance of l_1. The reason why is that l_2 attempted to spawn on the instance of l_2 that it knew already (in terms of our calculus, location l_2 at incarnation λ), which was stored in its view, and not the one currently alive (location l_2 at

incarnation $\lambda + 1$). If we had given l_2 enough time it would have detected that the incarnation of l_2 it knew was dead since it *would not feel its heartbeat* and in that case the spawn would have succeeded since l_1 would have initiated a new connection.

An example of the corresponding system in our calculus is the following one:

$$\Delta \triangleright [\![\, \mathsf{ping}\ l_1 \,]\!]^{l_2}_{\kappa} \parallel [\![\, \mathsf{spawn}\ l_1.\bar{t} \,]\!]^{l_2}_{\kappa} \parallel [\![\, \mathsf{kill} \,]\!]^{l_1}_{\lambda} \parallel [\![\, \mathsf{create}\ l_1.\mathbf{0} \,]\!]^{\odot}$$

where

$$\Delta = \langle \{l_1 \mapsto \lambda, l_2 \mapsto \kappa)\}, \{l_1 \leftrightarrow n_2\}, \{l_2 \mapsto \hat{\mathbf{0}}\} \rangle$$

B.2 Running Example in Erlang

In this section we discuss how to reproduce the experiments showing the behaviors of the three servers presented in Section 2. Actually, for simplicity as bugged behavior we show **servDFV** from Section 3.4.

All the code of the examples can be found at [13] (including **servDF**).

We begin by **servD**.

The procedure to set up the three nodes is as above.

Then, we can connect to the interface.

```
gfabbret@ubuntu:~/$ docker exec -it interface.com erl -name test@interface.com
                    -setcookie cookie -remsh app@interface.com -hidden
```

Then, we can invoke the interface process which will send the request and observe the outcome.

```
(app@interface.com)1> servD:interface().
Initiating request
Response received
```

As expected, everything went well.

Now, let us discuss the more interesting **servDFV**, where the misbehavior is due to a spawn that fails because of a wrong view.

The procedure to set up the system is as before, nonetheless this time two remote consoles are required: one on the interface to start the process and one on the controller to restart the router. The commands to attach the remote consoles are as above, but for replacing the name of the container.

Now, we can initiate the request from the interface.

```
(app@interface.com)1> servDFV:interface().
Initiating request
```

The process gets stuck because the (bugged) router drops the message. Now, we can simulate a failure of the router container by means of

```
gfabbret@ubuntu:~/$ ./restart servdf_net1 router.com
```

Then, right after, before the interface node detects the absence of the router because of the lack of its *heartbeat* we can restore the service from the controller console.

```
(app@controller.com)1> servDFV:controller().
ok
```

and observe the following behavior on the interface

```
=WARNING REPORT==== 3-May-2023::09:06:33.441402 ===
** Can not start erlang:apply,[#Fun<servDF.1.93666681>,[]] on 'app@router.com'
**
```

where the spawn has failed due to wrong view.

Finally, let us discuss **servDFR**.

To set up the scenario and attach consoles to the interface and the controller we proceed as above.

Then we start the request

```
(app@interface.com)1> servDFR:interface().
Initiating request
```

The process gets stuck like it did in **servDFV**. As in **servDFV** we restart the router and we create a new router through the controller process. In this implementation though is the router to send a *retry* message to the interface, hence its view is correctly updated and the following expected behavior is shown.

```
Retrying request
Response received
ok
```

C Modifying networks

We formalize in this section the intuition that we can modify, in essentially arbitrary ways, the observable part of networks in our systems.

Proposition 2. *Consider a system $S = \nu\,\tilde{u}.\Delta \triangleright N$, where $\Delta = \langle \mathcal{A}, \mathcal{L}, \mathcal{V} \rangle$, a set of live locations $\mathcal{A}'$, such that $\forall n \in \mathsf{supp}(\mathcal{A}) \setminus \mathsf{supp}(\mathcal{A}')$, $n \notin \tilde{u}$ and a set of live links $\mathcal{L}'$ s.t. $\forall n, m, (n, m) \in \mathcal{L} \setminus \mathcal{L}' \implies n, m \notin \tilde{u}$. Let $\Delta' = \langle \mathcal{A}', \mathcal{L}', \mathcal{V} \rangle$, then, there exists a configuration L with $fn(L) \cap \tilde{u} = \emptyset$ such that*

$$\nu\,\tilde{u}.\Delta \triangleright N \parallel L \Longrightarrow \nu\,\tilde{u}.\Delta' \triangleright N$$

Proof. Assume to have $\nu\,\tilde{u}.\Delta \triangleright N$, where $\Delta = \langle \mathcal{A}, \mathcal{L}, \mathcal{V} \rangle$, $\mathcal{A}'$ and $\mathcal{L}'$.

To transform $\mathcal{A}$ into $\mathcal{A}'$ we need to remove all the elements that are in $\mathcal{A}$ but not in $\mathcal{A}'$ and we need to add all those elements that are in $\mathcal{A}'$ but not in $\mathcal{A}$. To remove the nodes it suffices to build the following context

$$L^-_{\mathcal{A}} = \prod_{n \in \mathsf{supp}(\mathcal{A}) \setminus \mathsf{supp}(\mathcal{A}')} [\,\mathsf{kill}\,]^n_\lambda$$

Then, to add the nodes it suffices to build the following context

$$L^+_{\mathcal{A}} = \prod_{n \in \mathsf{supp}(\mathcal{A}') \setminus \mathsf{supp}(\mathcal{A})} [\,\mathsf{create}\; n.\mathbf{0}\,]^{\odot}$$

To change $\mathcal{L}$ in to $\mathcal{L}'$ we need to perform the same operations as above. To remove the links it suffices to build the following context

$$L^-_{\mathcal{L}} = \prod_{(n,m) \in \mathcal{L} \setminus \mathcal{L}'} [\,\mathsf{unlink}\; m.\mathbf{0}\,]^n$$

Then, to add the links it suffices to build the following context

$$L^+_{\mathcal{L}} = \prod_{(n,m) \in \mathcal{L}' \setminus \mathcal{L}} [\,\mathsf{link}\; m.\mathbf{0}\,]^n$$

Finally, we obtain as $L = L^-_{\mathcal{A}} \parallel L^+_{\mathcal{A}} \parallel L^-_{\mathcal{L}} \parallel L^+_{\mathcal{L}}$. □

Note that we did not mention incarnation numbers for new nodes in $\mathsf{supp}(\mathcal{A}') \setminus \mathsf{supp}(\mathcal{A})$. If a particular incarnation number $\lambda > 1$ is required for location n, it suffices to repeatedly kill and (re-)create location n in $L^+_{\mathcal{A}}$ until the correct incarnation number is reached.

In the above proposition views updates are not mentioned because there are some limits in extending them. In particular consider the network Δ

$$\langle \{(\odot, 1), (n, \lambda), (n', \lambda')\}, \{n \leftrightarrow n'\}, \{(n', \{(n, \lambda_o)\})\} \rangle \tag{1}$$

where $n \neq n'$, $\lambda_o < \lambda$ and the following network Δ'

$$\langle \{(\odot, 1), (n, \lambda), (n', \lambda')\}, \{n \leftrightarrow n'\}, \{(n', \{(n, \lambda_o)\}), (n, \{(n', \lambda')\})\} \rangle \tag{2}$$

Then Δ cannot be modified by context into Δ'. The problem lies in the fact that any primitive that adds a location to another location's view requires for the remote location to be in its view. In the above example, n' view is not aligned with the reality, since n' believes to be connected to (n, λ_o) while (n, λ_o) is dead and now (n, λ), a newer incarnation, is up and running.

D Barb alternative

This section shows that it is possible to simplify our notion of barb, a result akin to that put forward by De Nicola et al. in [25]. The alternate definition of barb we consider in this section is one where a barb just displays the location of messages.

Definition 8 (Location Barb)**.** *We say that a system S exhibits a barb at location n, in symbols $S{\downarrow_n}$, iff $S \equiv \nu\,\tilde{u}.\Delta \triangleright \llbracket\, \overline{x}\langle\tilde{v}\rangle.P \,\rrbracket^n_\lambda \parallel N$, for some $x, n, \lambda, \tilde{u}, \tilde{v}, P, N$, where $x, n \notin \tilde{u}$, and $\Delta \vdash (n, \lambda) : \mathsf{alive}$. Also, $S{\Downarrow_n}$ if $S \Rightarrow S'$ and $S'{\downarrow_n}$. We denote by $\dot{\approx}_l$ the weak barbed congruence obtained by considering only barbs at locations.*

Now the two weak barbed congruences $\dot{\approx}$ and $\dot{\approx}_l$ coincide

Proposition 3. $\dot{\approx}_l = \dot{\approx}$

Proof. In what follows we use the following notation : if $U \equiv \nu\,\tilde{u}.\Delta \triangleright N$, and L is such that $\mathsf{fn}(L) \cap \tilde{u} = \emptyset$, then $U \parallel L$ denotes the system $\nu\,\tilde{u}.\Delta \triangleright N \parallel L$.

That $\dot{\approx} \subseteq \dot{\approx}_l$ is clear. We show the converse. Let closed systems S and T be such that $S \,\dot{\approx}_l\, T$ and $S{\downarrow_{x@n_\lambda}}$. By definition of full barbs, we have $S \equiv \nu\,\tilde{u}.\Delta \triangleright \llbracket\, \overline{x}\langle\tilde{v}\rangle.P \,\rrbracket^n_\lambda \parallel N$ for some $x, n, \lambda, \tilde{u}, \tilde{v}, P, N$. By definition of location barbs, we have $S{\downarrow_n}$. Since $S \,\dot{\approx}_l\, T$, we must have $T \Longrightarrow T_1{\downarrow_n}$ for some T_1. Assume for the sake of contradiction that $\neg(T{\Downarrow_{x@n_\lambda}})$, and consider the systems $S \parallel L$ and $T \parallel L$, where:

$$L = \llbracket\, \mathsf{create}\ k.\mathsf{link}\ n.\mathsf{spawn}\ n.(x(\tilde{y}).\mathsf{spawn}\ k.\overline{t}) \,\rrbracket^{\odot}_1 \qquad\qquad k, t \notin \mathsf{fn}(S) \cup \mathsf{fn}(T)$$

Since $\dot{\approx}_l$ is a system congruence, we must have $S \parallel L \;\dot{\approx}_l\; T \parallel L$. Now, by construction, we have

$$S \parallel L \Longrightarrow \nu\,\tilde{u}.\Delta \triangleright N \parallel \llbracket\, \overline{t} \,\rrbracket^k_1 \qquad\qquad \text{and} \quad S{\downarrow_k}$$

But since $\neg(T{\Downarrow_{x@n_\lambda}})$, and t, k are fresh for T, there can be no T' such that $T \parallel L \Longrightarrow T'$ and $T'{\Downarrow_k}$, contradicting the fact that $S \;\dot{\approx}_l\; T$. □

E Full Abstraction

E.1 Soundness

Definition 9 (Strong Simulation)**.** *A binary relation $\mathcal{S} \subseteq \mathbb{S} \times \mathbb{S}$ over closed systems is a strong simulation iff whenever $(P, Q) \in \mathcal{S}$,*

- $P \xrightarrow{\alpha} P'$ *implies* $Q \xrightarrow{\alpha} Q'$ *for some* Q' *with* $(P', Q') \in \mathcal{S}$

Definition 10 (Strong Bisimulation)**.** *A relation $\mathcal{S}$ is a strong bisimulation if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are strong simulations.*

Definition 11 (Strong Bisimilarity)**.** *Strong bisimilarity, denoted by $\sim$, is the largest srong bisimulation over systems.*

The fact $\sim$ is an equivalence relation is a standard result for any labelled transition system.

We start with a simple lemma on the preservation of free names by labelled transitions. If α is a label from our LTS semantics, we define $\mathsf{pi}(\alpha)$ and $\mathsf{po}(\alpha)$ as follows:

$$\mathsf{po}(\alpha) = \begin{cases} \tilde{u} \setminus \tilde{w} & \text{if} \quad \alpha = \nu\, \tilde{w}.\overline{x}\langle \tilde{u} \rangle @ n_\lambda \\ \emptyset & \text{otherwise} \end{cases} \qquad \mathsf{pi}(\alpha) = \begin{cases} \tilde{u} & \text{if} \quad \alpha = x(\tilde{u}) @ n_\lambda \\ \emptyset & \text{otherwise} \end{cases}$$

Lemma 1. *Let S, S' be closed systems such that $S \xrightarrow{\alpha} S'$. Then $\mathsf{fn}(S') \subseteq \mathsf{fn}(S) \cup \mathsf{pi}(\alpha)$.*

Proof. By induction on the derivation of $S \xrightarrow{\alpha} S'$. □

We now prove two lemmas relating labelled transitions in the calculus LTS semantics and the structure of systems. In the remainder, we extend $\equiv$ to output actions, and identify equivalent output actions, setting:

$$\nu\, v.\nu\, w.\overline{x}\langle \tilde{u} \rangle @ n_\lambda \;\equiv\; \nu\, w.\nu\, v.\overline{x}\langle \tilde{u} \rangle @ n_\lambda \qquad\qquad \nu\, w.\alpha \;\equiv\; \nu\, w.\omega \quad \text{if} \quad \alpha \;\equiv\; \omega$$

Lemma 2 (Input/output actions and systems)**.** *Let S, S' be closed systems such that $S \xrightarrow{\alpha} S'$. The following properties hold:*

1. *if $\alpha = \nu\, \tilde{v}.\overline{x}\langle \tilde{u} \rangle @ n_\lambda$ then*

   $$S \equiv \nu\, \tilde{w}.\Delta \triangleright \llbracket \overline{x}\langle \tilde{u} \rangle.P \rrbracket^n_\lambda \parallel N \quad \text{for some } \Delta, \tilde{w}, P, N \quad \text{with} \quad x, n \notin \tilde{w},\ \tilde{v} \subseteq \tilde{w},\ \Delta \vdash n_\lambda : \mathsf{alive}$$

2. *if $\alpha = x(\tilde{u}) @ n_\lambda$ then*

   $$S \equiv \nu\, \tilde{w}.\Delta \triangleright \llbracket x(\tilde{u}).P \rrbracket^n_\lambda \parallel N \quad \text{for some } \Delta, \tilde{v}, \tilde{w}, P, N \quad \text{with} \quad \tilde{u}, x, n \cap \tilde{w} = \emptyset,\ \Delta \vdash n_\lambda : \mathsf{alive}$$

Proof. We show the first assertion, the second one is handled similarly. We reason by induction on the derivation of $S \xrightarrow{\alpha} S'$, where $\alpha = \nu\, \tilde{v}.\overline{x}\langle \tilde{u} \rangle @ n_\lambda$, considering the last rule used in the proof tree:

- Rule L-OUT: in this case, we have $S = \Delta \triangleright \llbracket \overline{x}\langle \tilde{u} \rangle.P \rrbracket^n_\lambda$, $\Delta \vdash n_\lambda : \mathsf{alive}$, as required.
- Rule L-PARL: in this case, we have $S = \Delta \triangleright N \parallel M$, $\alpha = \overline{x}\langle \tilde{u} \rangle @ n_\lambda$, with $\Delta \triangleright N \xrightarrow{\alpha} \Delta' \triangleright N'$. By induction assumption, we have $N \equiv \llbracket \overline{x}\langle \tilde{u} \rangle.P \rrbracket^n_\lambda \parallel L$ for some L, with $\Delta \vdash n_\lambda : \mathsf{alive}$. Hence $S \equiv \Delta \triangleright \llbracket \overline{x}\langle \tilde{u} \rangle.P \rrbracket^n_\lambda \parallel L \parallel M$, $\Delta \vdash n_\lambda : \mathsf{alive}$, as required.

- Rule L-PAR$_{\mathrm{R}}$: same as L-PAR$_{\mathrm{L}}$.
- Rule L-RES: in this case we have $S = \nu\, z.S'$ with $S' \xrightarrow{\alpha} S''$, for some S', S'', and $v \notin \mathsf{fn}(\alpha)$. By induction assumption, we have $S' \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \overline{x}\langle \tilde{s}\rangle.P \,]\!]^n_\lambda \parallel N$, with $\Delta \vdash n_\lambda : \mathsf{alive}$, $x, n \notin \tilde{w}$, $\tilde{v} \subseteq \tilde{w}$. Thus $S \equiv \nu\, z, \tilde{w}.\Delta \triangleright [\![\, \overline{x}\langle \tilde{s}\rangle.P \,]\!]^n_\lambda \parallel N$, with $\Delta \vdash n_\lambda : \mathsf{alive}$, and $x, n \notin v, \tilde{w}$, $\tilde{v} \subseteq z, \tilde{w}$ as required.
- Rule L-RES$_{\mathrm{O}}$: in this case, we have $S = \nu\, z.T$ with $T \xrightarrow{\omega} T'$, $z \in \tilde{u} \setminus x, n$, $\alpha = \nu\, z.\omega$. By induction assumption, we have

  $T \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \overline{x}\langle \tilde{r}\rangle.P \,]\!]^n_\lambda \parallel N \quad$ for some $\Delta, \tilde{u}, \tilde{w}, P, N$ with $x, n \notin \tilde{w}$, $\tilde{v} \backslash \{z\} \subseteq \tilde{w}$, $\Delta \vdash n_\lambda : \mathsf{alive}$

  Hence, we have

  $S \equiv \nu\, z, \tilde{w}.\Delta \triangleright [\![\, \overline{x}\langle \tilde{r}\rangle.P \,]\!]^n_\lambda \parallel N \quad$ for some $\Delta, \tilde{u}, \tilde{w}, P, N$ with $x, n \notin z, \tilde{w}$, $\tilde{v} \subseteq z, \tilde{w}$, $\Delta \vdash n_\lambda : \mathsf{alive}$

  as required.

□

Lemma 3 (Silent actions and systems)**.** *Let S, S' be closed systems such that $S \xrightarrow{\tau} S'$. Then one of the following properties hold, for some $\tilde{u}, \tilde{w}, u, n, m, \lambda, P, Q, N$ with $\Delta \vdash (n, \lambda) : \mathsf{alive}$:*

$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, !x(\tilde{u}).P \,]\!]^n_\lambda \parallel N \tag{3}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \nu\, u.P \,]\!]^n_\lambda \parallel N \tag{4}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q \,]\!]^n_\lambda \parallel N \tag{5}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, P \mid Q \,]\!]^n_\lambda \parallel N \tag{6}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{node}(m, \kappa).P \,]\!]^n_\lambda \parallel N \tag{7}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{remove}\, n.P \,]\!]^n_\lambda \parallel N \tag{8}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{kill} \,]\!]^n_\lambda \parallel N \quad \text{and} \quad n \neq \odot \tag{9}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{spawn}\, m.P \,]\!]^n_\lambda \parallel N \tag{10}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{create}\, m.P \,]\!]^n_\lambda \parallel N \tag{11}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{link}\, m.P \,]\!]^n_\lambda \parallel N \tag{12}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \mathsf{unlink}\, m.P \,]\!]^n_\lambda \parallel N \tag{13}$$
$$S \equiv \nu\, \tilde{w}.\Delta \triangleright [\![\, \overline{x}\langle \tilde{u}\rangle.P \,]\!]^n_\lambda \parallel [\![\, x(\tilde{u}).Q \,]\!]^n_\lambda \parallel N \tag{14}$$

Proof. We reason by induction on the derivation of $S \xrightarrow{\tau} S'$, considering the last rule used in the derivation.

- Rule L-RES: In this case, we have $S = \nu\, u.T$ and $S' = \nu\, u.T'$, with $T \xrightarrow{\tau} T'$. Applying the induction assumption, we have $T \equiv \nu\, \tilde{w}.\Delta \triangleright L \parallel N$, where L is one of the located processes listed in the lemma assertions ($[\![\, !x(\tilde{u}).P \,]\!]^n_\lambda$, $[\![\, \nu\, u.P \,]\!]^n_\lambda$, etc.). Hence $S \equiv \nu\, u, \tilde{w}.\Delta \triangleright L \parallel N$, as required.
- Rule L-SYNC$_{\mathrm{L}}$: In this case, we have $S = \Delta \triangleright N \parallel M$ with $\Delta \triangleright N \xrightarrow{\overline{x}\langle \tilde{u}\rangle @ n_\lambda} \Delta \triangleright N'$ and $\Delta \triangleright M \xrightarrow{x(\tilde{u}) @ n_\lambda} \Delta \triangleright M'$. Applying Lemma2 we have $N \equiv [\![\, \overline{x}\langle \tilde{u}\rangle.P \,]\!]^n_\lambda \parallel U$ for some P, U with $\Delta \vdash n_\lambda : \mathsf{alive}$, and $M \equiv [\![\, x(\tilde{u}).Q \,]\!]^n_\lambda \parallel V$ for some Q, V. Hence we have $S \equiv \Delta \triangleright [\![\, \overline{x}\langle \tilde{u}\rangle.P \,]\!]^n_\lambda \parallel [\![\, x(\tilde{u}).Q \,]\!]^n_\lambda \parallel U \parallel V$, one of the possible forms required.
- Rule L-SYNC$_{\mathrm{R}}$ is handled similarly as rule L-SYNC$_{\mathrm{L}}$.

- Rule L-PAR$_{\mathrm{L}}$: In this case we have $S = \Delta \triangleright N \parallel M$, with $\Delta \triangleright N \xrightarrow{\tau} \nu\,\tilde{u}.\Delta' \triangleright N'$, and $\mathsf{fn}(M) \cap \tilde{u} = \emptyset$. By induction assumption, we have $N \equiv L \parallel U$, where L is one of the located processes listed in the lemma assertions. Hence $S \equiv \Delta \triangleright L \parallel U \parallel M$, as required.
- Rule L-PAR$_{\mathrm{R}}$ is handled similarly as rule L-PAR$_{\mathrm{L}}$.
- Rules with $\xrightarrow{\tau}$ conclusion in Fig.10: All these rules conclusion are of the form $\Delta \triangleright L \xrightarrow{\tau} U$, where L is one of the single located processes listed in the lemma assertions. In these cases, we just have $S = \Delta \triangleright L$, as required.

□

Proposition 4 (Structural congruence is a strong bisimulation)**.** *We have* $\equiv \subseteq \sim$.

Proof. Since $\equiv$ is an equivalence relations, it suffices to prove that $\equiv$ is a strong simulation, namely that for any closed systems S, R, if $S \xrightarrow{\alpha} S'$ and $S \equiv R$, then there exists R' such that $R \xrightarrow{\alpha} R'$ and $S' \equiv R'$. We reason by induction on the derivation of $S \equiv R$, considering the last rule used in the proof tree:

Rule S.RES.C**:** In this case, $S = \nu\,u.\nu\,v.T$ and $R = \nu\,v.\nu\,u.T$. Since $S \xrightarrow{\alpha} S'$, this can only have been obtained by applying rule L-RES twice, rule L-RES$_{\mathrm{O}}$ twice, or a or a combination of rule L-RES and rule L-RRES$_{\mathrm{O}}$. We consider the four cases:

Rule L-RES **applied twice:** In this case, we have $T \xrightarrow{\omega} T'$, for some T', $u, v \notin \mathsf{fn}(\alpha)$, $\omega = \alpha$, and $S' = \nu\,u.\nu\,v.T'$. Now applying rule L-RES twice we get $R \xrightarrow{\alpha} \nu\,v.\nu\,u.T'$. Applying rule L-RES twice, we get $R' \equiv S'$, as required.

Rule L-RES$_{\mathrm{O}}$ **applied twice:** In this case, we have $T \xrightarrow{\omega} T'$ for some T', $v \in \pi\omega$, $u \in \pi\omega \setminus \{v\}$, $\alpha = \nu\,u.\nu\,v.\omega$, and $S' = T'$. Applying rule L-RES$_{\mathrm{O}}$ twice we get: $R \xrightarrow{\alpha} T'$, hence we have $R' \equiv S'$, as required.

Rule L-RES$_{\mathrm{O}}$ **followed by rule** L-RES**:** In this case we have $T \xrightarrow{\omega} T'$ for some T', $v \in \pi\omega$, $u \notin \pi\omega \setminus \{v\}$, $\alpha = \nu\,v.\omega$, and $S' = \nu\,u.T'$. Applying rule L-RES$_{\mathrm{O}}$ we get: $\nu\,u.T \xrightarrow{\omega} \nu\,u.T'$. Applying L-RES$_{\mathrm{O}}$ we get: $R = \nu\,v.\nu\,u.T \xrightarrow{\alpha} \nu\,u.T' = R'$. Hence we have $R' \equiv S'$, as required.

Rule L-RES **followed by rule** L-RES$_{\mathrm{O}}$**:** Similar to the previous case.

Rule S.RES.NIL**:** In this case, $S = \nu\,u.R$, $u \notin \mathsf{fn}(R)$. Since $S \xrightarrow{\alpha} S'$, this can only have been obtained by applying rule RESS as the last rule. Hence, we have $R \xrightarrow{\omega} R'$, $S' = \nu\,u.R'$, $\alpha = \omega\{*/u\}$, $u \notin \mathsf{fn}(\omega) \setminus \mathsf{po}(\omega)$. Now by Lemma 1, we have $\mathsf{fn}(R') \subseteq \mathsf{fn}(R) \cup \mathsf{pi}(\omega)$. Since $\mathsf{po}(\omega) \subseteq \mathsf{fn}(R)$ and $u \notin \mathsf{fn}(R)$, then $u \notin \mathsf{po}(\omega)$, hence $\alpha = \omega$. In addition, since $u \notin \mathsf{fn}(\omega)$, then $u \notin \mathsf{pi}(\omega)$, hence $u \notin \mathsf{fn}(R')$. We can then apply rule S.RES.NIL to get $S' \equiv R'$, as required.

Rule S.α**:** In this case $S =_{\alpha} R$. By rule L-α we get directly $R \xrightarrow{\alpha} S'$. Hence we have found $R' = S'$, as required.

Rule S.CTX**:** In this case, we have $S = \nu\,\tilde{u}.\Delta \triangleright L \parallel N$ and $R = \nu\,\tilde{u}.\Delta \triangleright L \parallel M$ with $N \equiv M$. We reason by induction on the structure of $\tilde{u}$:

$\tilde{u} = \emptyset$**:** In this case, $S \xrightarrow{\alpha} S'$ can only have been derived by an application of rule L-PAR$_{\mathrm{L}}$, rule L-PAR$_{\mathrm{R}}$, rule L-SYNC$_{\mathrm{R}}$, or rule L-SYNC$_{\mathrm{L}}$. We consider the four cases:

Rule L-PAR$_\mathrm{L}$**:** In this case, we have $\Delta \triangleright L \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright L'$ with $\tilde{v} \cap \mathsf{fn}(N)$, and $S' = \nu\,\tilde{v}.\Delta' \triangleright L' \parallel N$. Applying rule L-PAR$_\mathrm{L}$ we get $R \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright L' \parallel N = R'$, and by applying rule S.CTX we have $R' \equiv S'$, as required.

Rule L-PAR$_\mathrm{R}$**:** In this case we have $\Delta \triangleright N \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright N'$. We reason according to the last rule used to derive $N \equiv M$:

Rule S.PAR.N**:** In this case, we have $N = (M \parallel \mathbf{0})$. Now, $\Delta \triangleright N \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright N'$ can only have been obtained via an application of rule L-PAR$_\mathrm{L}$, with $\Delta \triangleright M \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright M'$, and $N' = M' \parallel \mathbf{0}$. But then applying L-PAR$_\mathrm{R}$ we get: $R \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright L \parallel M' = R'$. Since $N' \equiv M'$ by rule S.PAR.N, we have by rule S.CTX $S' \equiv R'$, as required.

Rule S.PAR.C**:** In this case, we have $N = U \parallel V$ and $M = V \parallel U$. Now the derivation $\Delta \triangleright N \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright N'$ can only have been obtained by the application of one of the rules L-PAR$_\mathrm{L}$, L-PAR$_\mathrm{R}$, L-SYNC$_\mathrm{L}$ or L-SYNC$_\mathrm{R}$. We consider the different cases.

Rule L-PAR$_\mathrm{L}$: In this case we have $\Delta \triangleright U \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright U'$, $\tilde{v} \cap V = \emptyset$ and $N' = U' \parallel V$. Applying rule L-PAR$_\mathrm{R}$ we obtain $\Delta \triangleright M \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright M'$ with $M' = V \parallel U'$. Applying rule L-PAR$_\mathrm{R}$ we get $R = \Delta \triangleright L \parallel M \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright L \parallel (V \parallel U') = R'$. Since we have $S' = \nu\,\tilde{v}.\Delta' \triangleright L \parallel (U' \parallel V)$, and by rule S.PAR.C $V \parallel U' \equiv U' \parallel V$, and since $\equiv$ is a congruence, we have by rule S.CTX $S' \equiv R'$, as required.

The case of rule L-PAR$_\mathrm{R}$ is handled similarly.

Rule L-SYNC$_\mathrm{L}$: In this case, we have $\alpha = \tau$, $\vec{v} = \emptyset$, $\Delta \triangleright U \xrightarrow{\overline{\omega}} \Delta \triangleright U'$, $\Delta \triangleright V \xrightarrow{\omega} \Delta \triangleright V'$, where $\overline{\omega}$ is an output action and ω is the matching input action. But then we can apply rule L-SYNC$_\mathrm{R}$ to obtain $\Delta \triangleright M \xrightarrow{\tau} \Delta \triangleright M'$ where $M' = V' \parallel U'$, and then apply rule L-PAR$_\mathrm{R}$ to get $R = \Delta \triangleright L \parallel (V \parallel U) \xrightarrow{\tau} \Delta \triangleright L \parallel (V' \parallel U')$ Since we have $S' = \Delta \triangleright L \parallel (U' \parallel V')$, $U' \parallel V' \equiv V' \parallel U'$ by rule S.PAR.C, and since $\equiv$ is a congruence, we obtain by rule S.CTX $S' \equiv R'$, as required.

The case of rule L-SYNC$_\mathrm{R}$ is handled similarly.

Rule S.PAR.A**:** this case is handled similarly to the case of rule S.PAR.C above.

Rule L-SYNC$_\mathrm{L}$**:** In this case we have $\alpha = \tau$, $\Delta \triangleright L \xrightarrow{\overline{\omega}} \Delta \triangleright L'$, $\Delta \triangleright N \xrightarrow{\omega} \Delta \triangleright N'$, and $S' = \Delta \triangleright L' \parallel N'$. We reason according to the last rule used to derive $N \equiv M$:

- Rule S.PAR.N: In this case, we have $N = (M \mid \mathbf{0})$. The transition $\Delta \triangleright N \xrightarrow{\overline{\omega}} \Delta \triangleright N'$ can only have been obtained by an application of rule L-PAR$_\mathrm{L}$, with $\Delta \triangleright M \xrightarrow{\omega} \Delta \triangleright M'$ for some M'. Thus we have $S' = \Delta \triangleright L' \parallel (M' \parallel \mathbf{0})$. Now by applying rule L-SYNC$_\mathrm{R}$ we get $R = \Delta \triangleright L \parallel M \xrightarrow{\tau} \Delta \triangleright L' \parallel M' = R'$. By rule S.PAR.N, we have $M' \parallel \mathbf{0} \equiv M'$. Since $\equiv$ is a congruence we have $L' \parallel (M' \parallel \mathbf{0}) \equiv L' \parallel M'$, and by rule S.CTX we get $S' \equiv R'$, as required.
- Rule S.PAR.C: we reason exactly as in the subcase S.PAR.C in the case of rule L-PAR$_\mathrm{L}$ above, except that the only rules to consider are L-PAR$_\mathrm{L}$ and L-PAR$_\mathrm{R}$.
- Rule S.PAR.A: this case is handled similarly to the case of rule S.PAR.C above.

Rule L-SYNC$_\mathrm{R}$**:** this case is handled similarly to the case of rule L-SYNC$_\mathrm{L}$ above.

$\vec{u} = v, \vec{w}$**:** In this case $R = \nu\,v.U$ with $T \equiv U$ and $S = \nu\,v.T \xrightarrow{\alpha} S'$. The latter can only have been obtained by rule L-RES or by rule L-RES$_\mathrm{O}$. We consider the two cases:

Rule L-RES**:** In this case, we have $T \xrightarrow{\alpha} T'$, $v \notin \pi\alpha$, $S' = \nu\,v.T'$. By the induction hypothesis, we have $U \xrightarrow{\alpha} U'$ for some $U' \equiv T'$. By rule L-RES we get $R \xrightarrow{\alpha} \nu\,v.U' = R'$ and since $\equiv$ is a congruence, we have $R' \equiv S'$, as required.

Rule L-RES$_{\mathrm{O}}$**:** In this case, we have $T \xrightarrow{\omega} T'$, $v \in \pi\omega$, $\alpha = \nu\, v.\omega$, $S' = T'$. By the induction hypothesis, we have $U \xrightarrow{\omega} U'$ for some $U' \equiv T'$. By rule L-RES$_{\mathrm{O}}$ we get $R \xrightarrow{\alpha} U' = R'$. hence we have $R' \equiv S'$, as required.

□

Proposition 5. *If* $S \longrightarrow S'$ *then* $S \xrightarrow{\tau} \equiv S'$.

Proof. We proceed by induction on the inference $S \longrightarrow S'$.

Case inferred by IF-EQ Then S is $\Delta \triangleright [\![\mathsf{if}\ u = u.P\ \mathsf{else}\ Q]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-IF-EQ.

Case inferred by IF-NEQ Then S is $\Delta \triangleright [\![\mathsf{if}\ u = v.P\ \mathsf{else}\ Q]\!]^n$, where $u \neq v$, and the transition $S \xrightarrow{\tau} S'$ follows by L-IF-NEQ.

Case inferred by BANG Then S is $\Delta \triangleright [\![!x(\tilde{u}.P)]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-BANG.

Case inferred by NODE Then S is $\Delta \triangleright [\![\mathsf{node}(m, \kappa).P]\!]^n$, and the transition $S \xrightarrow{\tau} S'$ follows by L-NODE.

Case inferred by MSG Then S is $\Delta \triangleright [\![\overline{x}\langle y \rangle]\!]^n_\lambda \parallel [\![x(z).P]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-OUT, L-IN, and L-SYNC$_{\mathrm{L}}$.

Case inferred by NEW Then S is $\Delta \triangleright [\![\nu\, x.P]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-NEW.

Case inferred by SPAWN-S Then S is $\Delta \triangleright [\![\mathsf{spawn}\ n.P]\!]^n$, and the transition $S \xrightarrow{\tau} S'$ follows by L-SPAWN-S.

Case inferred by SPAWN-F Then S is $\Delta \triangleright [\![\mathsf{spawn}\ n.P]\!]^n$, and the transition $S \xrightarrow{\tau} S'$ follows by L-SPAWN-F.

Case inferred by UNLINK Then S is $\Delta \triangleright [\![\mathsf{unlink}\ m.P]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-BREAK.

Case inferred by LINK Then S is $\Delta \triangleright [\![\mathsf{link}\ m.P]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-LINK.

Case inferred by KILL Then S is $\Delta \triangleright [\![\mathsf{kill}]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-KILL.

Case inferred by REMOVE Then S is $\Delta \triangleright [\![\mathsf{remove}\, m.P]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-REMOVE.

Case inferred by CREATE-S Then S is $\Delta \triangleright [\![\mathsf{create}\, m.P]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-CREATE-S.

Case inferred by CREATE-F Then S is $\Delta \triangleright [\![\mathsf{create}\, m.P]\!]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-CREATE-F.

Case inferred by PAR Then S is $\nu\, \tilde{u}.\Delta \triangleright N \parallel M$, and the transition $S \xrightarrow{\tau} S'$ follows by L-PAR$_{\mathrm{L}}$ and the inductive hypothesis.

Case inferred by RES Then S is $\nu\, u.\Delta \triangleright N$, and the transition $S \xrightarrow{\tau} S'$ follows by L-RESS and the inductive hypothesis.

Case inferred by STR Then we have $S \equiv T$, $T \longrightarrow T'$ and $T' \equiv S'$. By induction hypothesis, we have $T \xrightarrow{\tau} T'$. By Proposition 4 we have $S \xrightarrow{\tau} S''$ with $S'' \equiv T'$. Hence $S \xrightarrow{\tau} S'' \equiv S'$, as required.

□

Proposition 6. *If* $S \xrightarrow{\tau} S'$ *then* $S \longrightarrow S'$.

Proof. We proceed by induction on the inference $S \xrightarrow{\tau} S'$.

Case L-IF-EQ Then S is$\Delta \triangleright [\![\mathsf{if}\ u = u\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]^n$, and the reduction $S \longrightarrow S'$ follows by IF-EQ.

Case L-IF-NEQ Then S is $\Delta \triangleright [\![\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]^n$, where $u \neq v$, and the reduction $S \longrightarrow S'$ follows by IF-NEQ.

Case L-BANG Then S is $\Delta \triangleright [\![!x(\tilde{u}).P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by BANG.

Case L-NODE Then S is $\Delta \triangleright [\![\mathsf{node}(m, \kappa).P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by NODE.

Case L-NEW Then S is $\Delta \triangleright [\![\nu\, x.P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by NEW.

Case L-SPAWN-S Then S is $\Delta \triangleright [\![\mathsf{spawn}\, m.P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by SPAWN-S.

Case L-SPAWN-F Then S is $\Delta \triangleright [\![\mathsf{spawn}\, m.P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by SPAWN-F.

Case L-UNLINK Then S is $\Delta \triangleright [\![\mathsf{unlink}\, m.P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by BREAK.

Case L-LINK Then S is $\Delta \triangleright [\![\mathsf{link}\, m.P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by LINK.

Case L-KILL Then S is $\Delta \triangleright [\![\mathsf{kill}]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by KILL.

Case L-CREATE-S Then S is $\Delta \triangleright [\![\mathsf{create}\, n.P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by CREATE-S.

Case L-CREATE-F Then S is $\Delta \triangleright [\![\mathsf{create}\, n.P]\!]^n_\lambda$, and the reduction $S \longrightarrow S'$ follows by CREATE-F.

Case L-SYNC$_\mathrm{L}$ Then S is $\Delta \triangleright N_1 \parallel N_2$, and we have

$$\Delta \triangleright N_1 \xrightarrow{\overline{x}\langle \tilde{v} \rangle @ n_\lambda} \Delta \triangleright N_1' \qquad\qquad \Delta \triangleright N_2 \xrightarrow{x(\tilde{v}) @ n_\lambda} \Delta \triangleright N_2'$$

Now, by Lemma 2 we know the following:

$$\Delta \triangleright N_1 \equiv \Delta \triangleright [\![\overline{x}\langle v \rangle]\!]^n_\lambda \parallel M_1 \qquad \Delta \triangleright N_2 \equiv \Delta \triangleright [\![x(u).P]\!]^n_\lambda \parallel M_2 \qquad S' = \Delta \triangleright N_1' \parallel N_2'$$

Hence $S \equiv \Delta \triangleright [\![\overline{x}\langle v \rangle]\!]^n_\lambda \parallel [\![x(u).P]\!]^n_\lambda \parallel M_1 \parallel M_2$. Applying MSG and STR, we get $S \equiv \longrightarrow S'$.

Case L-SYNC$_\mathrm{R}$ Similar to case L-SYNC$_\mathrm{L}$.

Case L-PAR$_\mathrm{L}$ Then S is $\Delta \triangleright N_1 \parallel N_2$, S' is $\nu\, \tilde{v}.\Delta' \triangleright N_1' \parallel N_2$, where $\Delta \triangleright N_1 \xrightarrow{\tau} \nu\, \tilde{v}.\Delta' \triangleright N_1'$, and $\tilde{v} \cap \mathsf{fn}(N_2) = \emptyset$. Now, by using the inductive hypothesis and PAR we have $\Delta \triangleright N_1 \parallel N_2 \longrightarrow \nu\, \tilde{v}.\Delta' \triangleright N_1' \parallel N_2$.

Case L-PAR$_\mathrm{R}$ Similar to case L-PAR$_\mathrm{L}$.

Case L-RES Then S is $\nu\, u.T$, S' is $\nu\, u.T'$, where $T \xrightarrow{\tau} T'$. Now by using the inductive hypothesis and RES we get $S = \nu\, u.T \longrightarrow \nu\, u.T' = S'$.

□

Lemma 4. *Let $S = \nu\,\tilde{s}.\Delta \triangleright N \parallel L$ be a closed system. If $S \xrightarrow{\alpha} S'$, where α is a silent action, an output action or an input action, then one of the following assertions holds:*

1. *$\alpha = \tau$, $\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{\tau} \nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright N'$, and $S' \equiv \nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright N' \parallel L$, with $\mathsf{fn}(L) \cap \tilde{w} = \emptyset$, and with rule* L-PAR$_\text{L}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rule* L-RES.

2. *$\alpha = \tau$, $\nu\,\tilde{s}.\Delta \triangleright L \xrightarrow{\tau} \nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright L'$, and $S' \equiv \nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright N \parallel L'$, with $\mathsf{fn}(N) \cap \tilde{w} = \emptyset$, and with rule* L-PAR$_\text{R}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rules* L-RES.

3. *$\alpha = \tau$, $\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{\nu\,\widetilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\widetilde{r}.\Delta \triangleright N'$, $\widetilde{r} = \tilde{s} \setminus \widetilde{w}$ and $\tilde{u} \cap \widetilde{r} = \emptyset$, $\Delta \triangleright L \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright L'$, and $S' \equiv \nu\,\tilde{s}.\Delta \triangleright N' \parallel L'$, with rule* L-SYNC$_\text{L}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rules* L-RES.

4. *$\alpha = \tau$, $\Delta \triangleright N \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright N'$, $\nu\,\tilde{s}.\Delta \triangleright L \xrightarrow{\nu\,\widetilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\tilde{r}.\Delta \triangleright L'$, $\widetilde{r} = \tilde{s} \setminus \widetilde{w}$ and $\tilde{u} \cap \widetilde{r} = \emptyset$, and $S' \equiv \nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright N' \parallel L'$, with rule* L-SYNC$_\text{R}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rules* L-RES.

5. *$\alpha = \nu\,\tilde{w}.\overline{x}\langle\tilde{u}@n_\lambda\rangle$, $\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{\nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\tilde{r}.\Delta \triangleright N'$, $\tilde{r} = \tilde{s} \setminus \tilde{w}$, $\tilde{u} \cap \tilde{r} = \emptyset$, and $S' \equiv \nu\,\tilde{s}.\Delta \triangleright N' \parallel L$ with rule* L-PAR$_\text{L}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rules* L-RES *or* L-RES$_\text{O}$.

6. *$\alpha = \nu\,\tilde{w}.\overline{x}\langle\tilde{u}@n_\lambda\rangle$, $\nu\,\tilde{s}.\Delta \triangleright L \xrightarrow{\nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\tilde{r}.\Delta \triangleright L'$, $\tilde{r} = \tilde{s} \setminus \tilde{w}$, $\tilde{u} \cap \tilde{r} = \emptyset$, and $S' \equiv \nu\,\tilde{s}.\Delta \triangleright N \parallel L'$ with rule* L-PAR$_\text{R}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rules* L-RES *or* L-RES$_\text{O}$.

7. *$\alpha = x(\tilde{u})@n_\lambda$, $\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{x(\tilde{u})@n_\lambda} \nu\,\tilde{s}.\Delta \triangleright N'$, $S' \equiv \nu\,\tilde{s}.\Delta \triangleright N' \parallel L$, and $\tilde{u} \cap \tilde{s} = \emptyset$, with rule* L-PAR$_\text{L}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rule* L-RES.

8. *$\alpha = x(\tilde{u})@n_\lambda$, $\nu\,\tilde{s}.\Delta \triangleright L \xrightarrow{x(\tilde{u})@n_\lambda} \nu\,\tilde{s}.\Delta \triangleright L'$, $S' \equiv \nu\,\tilde{s}.\Delta \triangleright N \parallel L'$, and $\tilde{u} \cap \tilde{s} = \emptyset$, with rule* L-PAR$_\text{R}$ *the last rule used in the derivation of $S \xrightarrow{\alpha} S'$ before possible applications of rule* L-RES.

Proof. By case analysis on α.

Case $\alpha = \tau$: in this case we reason by induction on the derivation of $S \xrightarrow{\tau} S'$, considering the last rule used in the proof tree:

Case L-PAR$_\text{L}$**:** In this case, we have $S = \Delta \triangleright N \parallel L$, $\Delta \triangleright N \xrightarrow{\tau} \nu\,\tilde{w}.\Delta' \triangleright N'$, and $S' = \Delta \triangleright N \xrightarrow{\tau} \nu\,\tilde{w}.\Delta' \triangleright N' \parallel L$, with $\mathsf{fn}(L) \cap \tilde{w} = \emptyset$, corresponding to assertion 1.

Case L-PAR$_\text{R}$**:** In this case, we have $S \equiv \Delta \triangleright N \parallel L$, $\Delta \triangleright L \xrightarrow{\tau} \nu\,\tilde{w}.\Delta' \triangleright L'$, and $S' = \Delta \triangleright N \xrightarrow{\tau} \nu\,\tilde{w}.\Delta' \triangleright N \parallel L'$, with $\mathsf{fn}(N) \cap \tilde{w} = \emptyset$, corresponding to assertion 2.

Case L-SYNC$_\text{L}$**:** In this case, we have $S = \Delta \triangleright N \parallel L$, $\Delta \triangleright N \xrightarrow{\overline{x}\langle\tilde{u}\rangle@n_\lambda} \Delta \triangleright N'$, $\Delta \triangleright L \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright L'$, and $S' = \Delta \triangleright N' \parallel L'$, corresponding to assertion 3.

Case L-SYNC$_{\mathrm{R}}$**:** In this case, we have $S = \Delta \triangleright N \parallel L$, $\Delta \triangleright N \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright N'$, $\Delta \triangleright L \xrightarrow{\overline{x}\langle\tilde{u}\rangle@n_\lambda} \Delta \triangleright L'$, and $S' = \Delta \triangleright N' \parallel L'$, corresponding to assertion 4.

Case L-RES**:** In this case, we have $S = \nu\, a.T$, $S' = \nu\, a.T'$, $T \xrightarrow{\tau} T'$. By induction hypothesis, we have for T one of the four cases 1 to 4 in the lemma. We consider only the cases L-PAR$_{\mathrm{L}}$ and L-SYNC$_{\mathrm{L}}$, the other ones are handled similarly.

Rule L-PAR$_{\mathrm{L}}$**:** In this case we have $T = \nu\, \tilde{s}.\Delta \triangleright N \parallel L$, $T' \equiv \nu\, \widetilde{w}.\nu\, \tilde{s}.\Delta \triangleright N' \parallel L$, $\nu\, \tilde{s}.\Delta \triangleright N \xrightarrow{\tau} \nu\, \widetilde{w}.\nu\, \tilde{s}.\Delta \triangleright N'$, with $\mathsf{fn}(L) \cap \widetilde{w} = \emptyset$. Applying rule L-RES, we get $S' \equiv \nu\, a.\nu\, \tilde{r}.\nu\, \tilde{s}.\Delta \triangleright N' \parallel L \equiv \nu\, a.\nu\, \tilde{r}.\nu\, \tilde{s}.\Delta \triangleright N' \parallel L$, $\nu\, a.\nu\, \tilde{s}.\Delta \triangleright N \xrightarrow{\tau} \nu\, \widetilde{w}.\nu\, a.\nu\, \tilde{s}.\Delta \triangleright N'$, with $\mathsf{fn}(L) \cap \widetilde{w} = \emptyset$, corresponding to assertion 1.

Rule L-SYNC$_{\mathrm{L}}$**:** In this case we have $T = \nu\, \tilde{s}.\Delta \triangleright N \parallel L$, $T' \equiv \nu\, \tilde{s}.\Delta \triangleright N' \parallel L'$, $\nu\, \tilde{s}.\Delta \triangleright N \xrightarrow{\nu\, \widetilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\, \tilde{r}.\Delta \triangleright N'$, $\Delta \triangleright L \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright L'$, with $\tilde{r} = \tilde{s} \setminus \widetilde{w}$ and $\tilde{u} \cap \tilde{r} = \emptyset$. Applying rule L-RES, we get $S = \nu\, w.\nu\, \tilde{s}.\Delta \triangleright N \parallel L \xrightarrow{\tau} \nu\, w.\nu\, \tilde{s}.\Delta \triangleright N' \parallel L' = S'$. If $a \in \tilde{u}$, then applying L-RES$_{\mathrm{O}}$, we get $\nu\, a.\nu\, \tilde{s}.\Delta \triangleright N \xrightarrow{\nu\, a, \widetilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\, \tilde{r}.\Delta \triangleright N'$, with $\tilde{u} \cap \tilde{r} = \emptyset$ and $\tilde{r} = a, \tilde{s} \setminus a, \widetilde{w}$, as required. If $a \notin \tilde{u}$, then applying L-RES we get $\nu\, a.\nu\, \tilde{s}.\Delta \triangleright N \xrightarrow{\nu\, \widetilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\, a.\nu\, \tilde{r}.\Delta \triangleright N'$, with $\tilde{u} \cap a, \tilde{r} = \emptyset$ and $a, \tilde{r} = a, \tilde{s} \setminus \widetilde{w}$, as required.

Case $\alpha = \nu\, \widetilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda$**:** handled by induction on the derivation of $S \xrightarrow{\alpha} S'$ with cases similar to the cases τ.L-PAR$_{\mathrm{L}}$, τ.L-PAR$_{\mathrm{R}}$ and τ.L-RES above.

Case $\alpha = x(\tilde{u})@n_\lambda$**:** handled by induction on the derivation of $S \xrightarrow{\alpha} S'$ with cases similar to the cases τ.L-PAR$_{\mathrm{L}}$, τ.L-PAR$_{\mathrm{R}}$ and τ.L-RES above.

□

Proposition 7 (Weak System Congruence)**.** *Weak bisimilarity is a weak system congruence; that is, if* $\nu\, \tilde{u}.\Delta \triangleright N \approx \nu\, \tilde{v}.\Delta' \triangleright M$ *then for all* $\widetilde{w}$*,* L*, with* $\mathsf{fn}(L) \cap \tilde{u}, \tilde{v} = \emptyset$*, we have* $\nu\, \widetilde{w}.\nu\, \tilde{u}.\Delta \triangleright N \parallel L \approx \nu\, \widetilde{w}.\nu\, \tilde{v}.\Delta' \triangleright M \parallel L$

Proof. We prove that the relation

$$\mathcal{S} \stackrel{\text{def}}{=} \{\ \langle \nu\, \widetilde{w}.\nu\, \tilde{s}.\Delta_S \triangleright N_S \parallel L,\ \nu\, \widetilde{w}.\nu\, \tilde{r}.\Delta_R \triangleright N_R \parallel L \rangle \ \mid\ \nu\, \tilde{s}.\Delta_S \triangleright N_S \approx \nu\, \tilde{r}.\Delta_R \triangleright N_R \quad \mathsf{fn}(L) \cap (\tilde{s} \cup \tilde{r}) = \emptyset\ \}$$

is a weak bisimulation up to $\equiv$. Since $\mathcal{S}$ is symmetric, it suffices to prove that $\mathcal{S}$ is a weak bisimulation up to $\equiv$. Define

$$S = \nu\, \tilde{s}.\Delta_S \triangleright N_S \quad R = \nu\, \tilde{r}.\Delta_R \triangleright N_R \quad S_L = \nu\, \tilde{w}.\nu\, \tilde{s}.\Delta_S \triangleright N_S \parallel L \quad R_L = \nu\, \tilde{w}.\nu\, \tilde{r}.\Delta_R \triangleright N_R \parallel L$$

and consider a transition $S_L \xrightarrow{\alpha} U$. We proceed by induction on the structure of $\widetilde{w}$:

Case $\widetilde{w} = \emptyset$**:** We proceed by case analysis on α:

Case τ We consider the different cases listed in Lemma 4 for $S_L = \nu\, \tilde{s}.\Delta_S \triangleright N_S \parallel L$ and $S_L \xrightarrow{\tau} U$:

Case L-SYNC$_{\mathrm{L}}$ In this case, we have:

$$\begin{aligned}
&S = \nu\, \tilde{s}.\Delta_S \triangleright N_S \xrightarrow{\nu\, \vec{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\, \widetilde{z_s}.\Delta_S \triangleright N'_S = S' \\
&\Delta_S \triangleright L \xrightarrow{x(\tilde{u})@n_\lambda} \Delta_S \triangleright L' \\
&\widetilde{z_s} = \tilde{s} \setminus \tilde{a} \quad \tilde{u} \cap \tilde{z_s} = \emptyset \\
&U = \nu\, \tilde{s}.\Delta_S \triangleright N'_S \parallel L'
\end{aligned}$$

Since $S \approx R$, we have $R \stackrel{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda}{\Longrightarrow} \nu\,\widetilde{w_r}.\nu\,\widetilde{z_r}.\Delta'_R \triangleright N'_R = R'$ for some $\tilde{w_r}, \Delta'_R, N'_R$, with $\widetilde{z_r} = \tilde{r} \setminus \tilde{a}$, and $R' \approx S'$. Now, since $R \stackrel{\tau}{\Longrightarrow} R_1 \xrightarrow{\overline{x}\langle\tilde{u}\rangle@n_\lambda} R_2 \stackrel{\tau}{\Longrightarrow} R'$, where $R_1 = \nu\,\widetilde{w_1}.\nu\,\widetilde{z_r}.\Delta^1_R \triangleright N^1_R$ for some Δ^1_R, N^1_R, we are guaranteed that n_λ is alive, and that $\Delta^1_R \triangleright L \xrightarrow{x(\tilde{u})@n_\lambda} \Delta^1_R \triangleright L'$. Also, we have $\tilde{u} \cap \widetilde{z_r} = \emptyset$. By repeated applications of rule L-PAR$_\text{L}$ and by one application of rule L-SYNC$_\text{L}$, we obtain $R_L \stackrel{\tau}{\Longrightarrow} V$, with $V = \nu\,\widetilde{w_r}.\nu\,\tilde{r}.\Delta'_R \triangleright N'_R \parallel L'$ and with $\mathsf{fn}(L') \cap \widetilde{w_r} = \emptyset$ because of the conditions of rule L-PAR$_\text{L}$.
Summing up, we have:

$$
\begin{array}{l}
U = \nu\,\tilde{a}.\nu\,\widetilde{z_s}.\Delta_S \triangleright N'_S \parallel L' = \nu\,\tilde{a}.S' \\
V \equiv \nu\,\tilde{a}.\nu\,\widetilde{w_r}.\nu\,\widetilde{z_r}.\Delta'_R \triangleright N'_R \parallel L' = \nu\,\tilde{a}.R' \\
S' \approx R' \\
\mathsf{fn}(L') \subseteq \mathsf{fn}(L) \cup \tilde{u} \setminus \tilde{a} \quad \text{by Lemma 1} \\
\mathsf{fn}(L') \cap \widetilde{w_r} \\
\tilde{u} \cap \widetilde{z_s} = \tilde{u} \cap \widetilde{z_r} = \emptyset
\end{array}
$$

Hence we have $\mathsf{fn}(L') \cap \widetilde{z_s} = \emptyset$ and $\mathsf{fn}(L') \cap (\widetilde{w_r} \cup \widetilde{z_r}) = \emptyset$, and $S' = \nu\,\widetilde{z_s}.\Delta_S \triangleright N'_S \approx \nu\,\tilde{w_r}.\nu\,\widetilde{z_r}.\Delta'_R \triangleright N'_R = R'$, which means that $\langle U, V\rangle \in \mathcal{S}$, as required.

Case L-SYNC$_\text{R}$ Similar to the case L-SYNC$_\text{L}$, but simpler.

Case L-PAR$_\text{L}$ In this case, we have $S_L \xrightarrow{\tau} U$, with $S_L = \Delta_S \triangleright N_S \parallel L$, $U = \nu\,\tilde{s}.\Delta'_S \triangleright N'_S \parallel L$, $S \xrightarrow{\tau} \nu\,\tilde{s}.\Delta'_S \triangleright N'_S$, with $\mathsf{fn}(L) \cap \tilde{s} = \emptyset$. Since $S \approx R$, we have $R \stackrel{\tau}{\Longrightarrow} R'$ with $R' \approx S'$ and $R' = \nu\,\tilde{r}.\Delta'_R \triangleright N'_R$. Now applying repeatedly rule L-PAR$_\text{L}$, we get $R_L \stackrel{\tau}{\Longrightarrow} \nu\,\tilde{r}.\Delta'_R \triangleright N'_R \parallel L = V$, with $\mathsf{fn}(L) \cap \tilde{r} = \emptyset$ because of the conditions of rules L-PAR$_\text{L}$. Now we have $\langle U, V\rangle \in \mathcal{S}$, as required.

Case L-PAR$_\text{R}$ Similar to the case L-PAR$_\text{L}$, but simpler.

Case $x(\tilde{u})@n_\lambda$ We consider the different cases listed in Lemma 4 for $S_L = \nu\,\tilde{s}.\Delta_S \triangleright N_S \parallel L$ and $S_L \xrightarrow{x(\tilde{u})@n_\lambda} U$:

Case L-PAR$_\text{L}$ In that case, we have $S = \nu\,\tilde{s}.\Delta_S \triangleright N_S$, $S \xrightarrow{x(\tilde{u})@n_\lambda} S'$ $S' = \nu\,\tilde{s}.\Delta_S \triangleright N'_S$. Since $S \approx R$, we have $R \stackrel{x(\tilde{u})@n_\lambda}{\Longrightarrow} R'$, where $R' = \nu\,\tilde{z}.\nu\,\tilde{r}.\Delta'_R \triangleright N'_R$. Now, by repeated application of rule L-PAR$_\text{L}$, we get $R_L \stackrel{x(\tilde{u})@n_\lambda}{\Longrightarrow} \nu\,\tilde{w}.\nu\,\tilde{r}.\Delta'_R \triangleright N'_R \parallel L = V$, with $\mathsf{fn}(L) \cap \nu\,\tilde{z}. = \emptyset$ because of the conditions of rule L-PAR$_\text{L}$. Since we also have $\mathsf{fn}(L) \cap \tilde{r} = \emptyset$ by definition, we have $\langle U, V\rangle \in \mathcal{S}$, as required.

Case L-PAR$_\text{R}$ Similar to the case L-PAR$_\text{L}$, but simpler.

Case $\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda$ We consider the different cases listed in Lemma 4 for $S_L = \nu\,\tilde{s}.\Delta_S \triangleright N_S \parallel L$ and $S_L \xrightarrow{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} U$:

Case L-PAR$_\text{L}$ In that case, we have $S = \nu\,\tilde{s}.\Delta_S \triangleright N_S$, $S \xrightarrow{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} S'$ $S' = \nu\,\widetilde{s_a}.\Delta_S \triangleright N'_S$, $U = \nu\,\widetilde{s_a}\Delta_S. \triangleright N'_S \parallel L$, $\widetilde{s_a} = \tilde{s} \setminus \tilde{a}$. Since $S \approx R$, we have $R \stackrel{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda}{\Longrightarrow} R'$, where $R' \equiv \nu\,\tilde{w}.\nu\,\widetilde{r_a}.\Delta'_R \triangleright N'_R$, $\widetilde{r_a} = \tilde{r} \setminus \tilde{a}$, and $S' \approx R'$. Now, by repeated application of rule L-PAR$_\text{L}$, we get $R_L \stackrel{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda}{\Longrightarrow} \nu\,\tilde{z}.\nu\,\widetilde{r_a}.\Delta'_R \triangleright N'_R \parallel L = V$, with $\mathsf{fn}(L) \cap \nu\,\tilde{z}. = \emptyset$ because of the conditions of rule L-PAR$_\text{L}$. Since we also have $\mathsf{fn}(L) \cap \tilde{r} = \emptyset$ by definition, we have $\langle U, V\rangle \in \mathcal{S}$, as required.

Case L-PAR$_\text{R}$ Similar to the case L-PAR$_\text{L}$, but simpler.

Case $\mathsf{create}(n,\lambda)$ Easy since S_L and S have the same network.

Case $\mathsf{kill}(n,\lambda)$ Easy since S_L and S have the same network.

Case $\oplus n \mapsto m$ Easy since S_L and S have the same network.

Case $\ominus n \mapsto m$ Easy since S_L and S have the same network.

Case $n \succ m$ Easy since S_L and S have the same network.

Case $\widetilde{w} = w, \widetilde{z}$**:** Let $\langle \nu\, w., \nu\, \widetilde{z}.S, \nu\, w., \nu\, \widetilde{z}.R \rangle \in \mathcal{S}$. Then we have by construction $\langle \nu\, \widetilde{z}.S, \nu\, \widetilde{z}.R \in \mathcal{S}$. By induction hypothesis, if $\nu\, \widetilde{z}.S \xrightarrow{\alpha} U$, then $\nu\, \widetilde{z}.R \xrightarrow{\alpha} V$ for some V with $\langle \langle U, V \rangle \in \mathcal{S}$. Now using rule L-RES or rule L-RES$_\mathrm{O}$, we obtain either $S_L \xrightarrow{\alpha} \nu\, w.U$ or $S_L \nu\, w. \xrightarrow{\nu\, w.\alpha} U$, which are matched respectively by $R_L \xrightarrow{\alpha} \nu\, w.V$ or $R_L \xrightarrow{\nu\, w.\alpha} V$. In both cases, we have $\langle U, V \rangle \in \mathcal{S}$ and $\langle \nu\, w.U, \nu\, w.V \rangle \in \mathcal{S}$ ($\mathcal{S}$ is closed under restriction by construction), as required.

□

Proposition 8 (Soundness of weak bisimilarity)**.** *Weak bisimilarity is sound with respect to weak barbed congruence, i.e.* $\approx \subseteq \dot{\approx}$.

Proof. Weak bisimilarity is weak barb-preserving thanks to Lemma 2. It is weak reduction-closed thanks to Proposition 6. It is a system congruence thanks to Proposition 7. Thus $\approx \subseteq \dot{\approx}$ by definition of weak barbed congruence as the largest of weak barb-preserving, reduction-closed, system congruence. □

E.2 Completeness

In the remainder of this section, we use the following notation: if $S = \nu\, \tilde{s}.\Delta \triangleright N$ is a closed system, we write $S \parallel M$ for the system $\triangleright \triangleright \nu\, \tilde{s}.\Delta N \parallel M$ provided $\mathsf{fn}(M) \cap \tilde{s} = \emptyset$.

Lemma 5 (Inducing Network Changes)**.**

- *Suppose* $\Delta \vdash (n,\lambda) : \mathsf{alive}$
  - $S \xrightarrow{\mathsf{kill}(n,\lambda)} S'$ *implies* $S \parallel [\![\, \mathsf{kill} \,]\!]^n_\lambda \longrightarrow S'$
  - $S \parallel [\![\, \mathsf{kill} \,]\!]^n_\lambda \longrightarrow S'$, *where* $S' \equiv \nu\, \tilde{u}.\Delta \triangleright N$, $\Delta \not\vdash (n,\lambda) : \mathsf{alive}$ *implies* $S \xrightarrow{\mathsf{kill}(n,\lambda)} S'$
- *Suppose* $\Delta \vdash (n,\lambda) : \mathsf{alive}$
  - $S \xrightarrow{\mathsf{create}(n,\lambda)} S'$ *implies* $S \parallel [\![\, \mathsf{create}\ n.P \,]\!]^m_\kappa \longrightarrow S' \parallel [\![\, P \,]\!]^n_\lambda$
  - $S \parallel [\![\, \mathsf{create}\ n.P \,]\!]^m_\kappa \longrightarrow S' \parallel [\![\, P \,]\!]^n_\lambda$, *where* $S' \equiv \nu\, \tilde{u}.\Delta \triangleright N$, $\Delta \vdash n_\lambda : \mathsf{alive}$ *implies* $S \xrightarrow{\mathsf{create}(n,\lambda)} S'$
- *Suppose* $S = \nu\, \tilde{u}\Delta.N$ *and* $\Delta \vdash n \leftrightarrow m$
  - $S \xrightarrow{\ominus n_\lambda \mapsto m} S'$, *where* $S' \equiv \nu\, \tilde{u}.\Delta' \triangleright N$ *implies* $S \parallel [\![\, \mathsf{unlink}\ m.P \,]\!]^n_\lambda \longrightarrow S' \parallel [\![\, P \,]\!]^n_\lambda$
  - $S \parallel [\![\, \mathsf{unlink}\ m.P \,]\!]^n_\lambda \longrightarrow S' \parallel [\![\, P \,]\!]^n$, *where* $S' \equiv \nu\, \tilde{u}.\Delta' \triangleright N$, $\Delta' \vdash n \leftrightarrow m$ *implies* $S \xrightarrow{\ominus n_\lambda \mapsto m} S'$
- *Suppose* $S = \nu\, \tilde{u}\Delta.N$ *and* $\Delta \vdash n \leftrightarrow m$
  - $S \xrightarrow{\oplus n_\lambda \mapsto m} S'$, *where* $S' \equiv \nu\, \tilde{u}.\Delta' \triangleright N$ *implies* $S \parallel [\![\, \mathsf{link}\ m.P \,]\!]^n_\lambda \longrightarrow S' \parallel [\![\, P \,]\!]^n_\lambda$

– $S \parallel [\![\mathsf{link}\ m.P]\!]^n_\lambda \longrightarrow S' \parallel [\![P]\!]^n_\lambda$, *where* $S' \equiv \nu\,\tilde{u}.\Delta' \triangleright N$, $\Delta' \vdash n \leftrightarrow m$ *implies* $S \xrightarrow{\oplus n_\lambda \mapsto m} S'$

Proof. The first clause for the action $\mathsf{kill}(n,\lambda)$ is proved by induction on the derivation $S = \nu\,\tilde{u}.\Delta \triangleright N \xrightarrow{\mathsf{kill}(n,\lambda)} \nu\,\tilde{u}.\Delta' \triangleright N = S'$. The second clause uses induction on the derivation of $\nu\,\tilde{u}.\Delta \triangleright N \parallel [\![\mathsf{kill}]\!]^n_\lambda \longrightarrow \nu\,\tilde{u}.\Delta' \triangleright N'$. The proof for the other clauses is similar. □

Given a system $S \equiv \nu\,\tilde{u}.\Delta \triangleright N$ and l s.t. $l \notin fn(S) \cup bn(S)$ we define S^l as $\nu\,\tilde{u}.\Delta \oplus (l,1) \triangleright N$.

Proposition 9. *If S is a closed system and $l, x \notin \mathsf{fn}(S)$ then $\nu\,x, l.S^l \parallel [\![\overline{x}]\!]^l_\lambda \sim S$.*

Proof. A direct proof that $\mathcal{R} = \{\langle \nu\,x, l.S^l \parallel [\![\overline{x}]\!]^l_\lambda, S\rangle \mid S \in \mathbb{S} \text{ closed},\ x, l \notin \mathsf{fn}(S)\}$ is a strong bisimulation, noting that any transition $\nu\,x, l.S^l \parallel [\![\overline{x}]\!]^l_\lambda \xrightarrow{\alpha} T^l$ must have been obtained by a derivation with $S^l \xrightarrow{\alpha} U^l$ as a premise, with $T^l = \nu\,x, l.U^l$, $x, n \notin \mathsf{fn}(\alpha)$, and that for any such transition we have $S \xrightarrow{\alpha} T$. □

Proposition 10. *If we have $S^l \parallel [\![\overline{x}]\!]^l_\lambda \mathrel{\dot{\approx}} R^l \parallel [\![\overline{x}]\!]^l_\lambda$ with x, l fresh for S, R then $S \mathrel{\dot{\approx}} R$.*

Proof. If $S^l \parallel [\![\overline{x}]\!]^l_\lambda \mathrel{\dot{\approx}} R^l \parallel [\![\overline{x}]\!]^l_\lambda$ then by system congruence we also have

$$\nu\,x, l.S^l \parallel [\![\overline{x}]\!]^l_\lambda \mathrel{\dot{\approx}} \nu\,x, l.R^l \parallel [\![\overline{x}]\!]^l_\lambda$$

and then by Proposition 9, Proposition 8, and transitivity of $\dot{\approx}$ we have $S \mathrel{\dot{\approx}} R$ as required. □

Theorem 2 (Completeness of $\approx$ w.r.t. $\dot{\approx}$)**.** *If $S \mathrel{\dot{\approx}} R$ then $S \approx R$.*

Proof. To prove the statement it suffices to show that $\dot{\approx} \cup \approx$ is a bisimulation up-to $\equiv$. Take $S \mathrel{\dot{\approx}} R$ and suppose that $S \xrightarrow{\alpha} S'$; we reason by case analysis on α.

- Case $\alpha = \tau$. Thanks to Proposition 5 and Proposition 6, the thesis follow by the reduction closure property.
- Case $\alpha = \overline{x}\langle\tilde{u}\rangle@n_\lambda$. Consider the context

  $$L = [\![x(\tilde{y}).\mathsf{if}\ \tilde{y} = \tilde{u}\ \mathsf{then}\ \mathsf{create}\ l.(\overline{fail} \mid fail.\overline{succ})]\!]^n_\lambda$$

  with l, $succ$ and $fail$ fresh and the reduction $S \parallel L \Longrightarrow T_1$, where $T_1 \equiv S'^l \parallel [\![\overline{succ}]\!]^l$. Now, since $S \mathrel{\dot{\approx}} R$, $T_1{\downarrow_{succ@l}}$ and $T_1{\not\downarrow_{fail@l}}$, we must have a transition $R \parallel L \Longrightarrow T_2$ s.t. $T_2{\downarrow_{succ@l}}$ and $T_2{\not\downarrow_{fail@l}}$. We remark here that we have $T_2{\downarrow_{succ@l}}$, with a strong barb, because the only way for $R \parallel L$ to make disappear $\downarrow_{fail@l}$ is to have consumed it on the fresh location l, thereby showing $\downarrow_{succ@l}$.

  The only way to obtain this is if $R \xRightarrow{\overline{x}\langle\tilde{u}\rangle@n_\lambda} R'$, hence $T_2 \equiv R'^l \parallel [\![\overline{succ}]\!]^l$. Now, since $(S'^l \parallel [\![\overline{succ}]\!]^l, R'^l \parallel [\![\overline{succ}]\!]^l) \in \dot{\approx}$, by Proposition 10 we have $(S', R') \in \dot{\approx}$ as required.
- Case $\alpha = x(\tilde{u})@n_\lambda$. Consider the context

  $$L = [\![\overline{x}\langle\tilde{u}\rangle.\mathsf{create}\ l.(\overline{fail} \mid fail.\overline{succ})]\!]^n_\lambda$$

  with l, $fail$, $succ$ fresh and the reduction $S \parallel L \Longrightarrow T_1$, where $T_1 \equiv S'^l \parallel [\![\overline{succ}]\!]^l$. Now, since $S \mathrel{\dot{\approx}} R$, $T_1{\downarrow_{succ@l}}$ and $T_1{\not\downarrow_{fail@l}}$, we must have a transition $R \parallel L \Longrightarrow T_2$ s.t. $T_2{\downarrow_{succ@l}}$ and $T_2{\not\downarrow_{fail@l}}$. The only way to obtain this is if $R \xRightarrow{x(\tilde{u})@n_\lambda} R'$, hence $T_2 \equiv R'^l \parallel [\![\overline{succ}]\!]^l$. Now, since $(S'^l \parallel [\![\overline{succ}]\!]^l, R'^l \parallel [\![\overline{succ}]\!]^l) \in \dot{\approx}$, by Proposition 10 we have $(S', R') \in \dot{\approx}$ as required.

- Case $\alpha = \oplus n_\lambda \mapsto m$. By Lemma 5 we know that $\llbracket \mathsf{link}\, m.P \rrbracket^n_\lambda$ induces the desired labeled action. Consider the context

$$L = \llbracket \mathsf{link}\, m.\mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ}) \rrbracket^n_\lambda$$

  with l, $fail$, $succ$ fresh and the reduction $S \parallel L \Longrightarrow T_1$, where $T_1 \equiv S'^l \parallel \llbracket \overline{succ} \rrbracket^l$. Now, since $S \dot{\approx} R$, $T_1 \downarrow_{succ@l}$ and $T_1 \not\downarrow_{fail@l}$, we must have a transition $R \parallel L \Longrightarrow T_2$ s.t. $T_2 \downarrow_{succ@l}$ and $T_2 \not\downarrow_{fail@l}$. The only way to obtain this is if $R \Longrightarrow R_1 \parallel \llbracket \mathsf{link}\, m.\mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ}) \rrbracket^n_\lambda \longrightarrow R_1 \parallel \llbracket \mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ}) \rrbracket^n \Longrightarrow T_2$, where $T_2 \equiv R'^l \parallel \llbracket \overline{succ} \rrbracket^l$.

  Then, by Lemma 5 we know that $R \stackrel{\tau}{\Longrightarrow} R_i \stackrel{\ominus n_\lambda \mapsto m}{\longrightarrow} R_j \stackrel{\tau}{\Longrightarrow} R'$.

  Now, since $(S'^l \parallel \llbracket \overline{succ} \rrbracket^l, R'^l \parallel \llbracket \overline{succ} \rrbracket^l) \in \dot{\approx}$, by Proposition 10 we have $(S', R') \in \dot{\approx}$ as required.

- Case $\alpha = \ominus n_\lambda \mapsto m$. Consider the context

$$L = \llbracket \mathsf{unlink}\, m.\mathsf{create}\, l.(\overline{fail} \mid fail.succ) \rrbracket^n_\lambda$$

  with l, $succ$ and $fail$ fresh. The reasoning is similar to above.

- Case $\alpha = \mathsf{kill}(n, \lambda)$. By Lemma 5 we know that $\llbracket \mathsf{kill} \rrbracket^n_\lambda$ is the process that induces the reduction we are looking for. Consider the context

$$L = \llbracket \overline{fail} \rrbracket^n_\lambda \parallel \llbracket \mathsf{kill} \rrbracket^n_\lambda$$

  with $fail$ fresh and reduction $S \parallel L \Longrightarrow T_1$.

  Now, since $S \dot{\approx} R$ and $T_1 \not\downarrow_{fail@n}$ there is a matching move $R \parallel L \Longrightarrow T_2$ s.t. $T_2 \not\downarrow_{fail}$. Since $fail$ is fresh the only way to have $T_2 \not\downarrow_{fail@n}$ is to have $R \parallel L \Longrightarrow R_1 \parallel \llbracket \overline{fail} \rrbracket^n_\lambda \parallel \llbracket \mathsf{kill} \rrbracket^n_\lambda \longrightarrow R'_1 \parallel \llbracket \overline{fail} \rrbracket^n_\lambda \Longrightarrow T_2$.

  By Lemma 5 we know that $R \stackrel{\tau}{\Longrightarrow} R_1 \stackrel{\mathsf{kill}(n,\lambda)}{\Longrightarrow} R'_1 \stackrel{\tau}{\Longrightarrow} R'$.

  Now, we know that $T_1 \dot{\approx} T_2$, where $T_1 \equiv S' \parallel \llbracket \overline{fail} \rrbracket^n_\lambda$ and $T_2 \equiv R' \parallel \llbracket \overline{fail} \rrbracket^n_\lambda$. We know $S' \parallel \llbracket \overline{fail} \rrbracket^n_\lambda \sim S'$ since n_λ is not alive in S' hence $S \parallel \llbracket \overline{fail} \rrbracket^n_\lambda \dot{\approx} S$ and by transitivity of $\dot{\approx}$ we get $S \dot{\approx} R \parallel \llbracket \overline{fail} \rrbracket^n_\lambda$. By using a similar reasoning we get $(S', R') \in \dot{\approx}$ as required.

- Case $\alpha = \mathsf{create}(n, \lambda)$. Consider the context

$$L = \llbracket \mathsf{create}\, n.(\mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ})) \rrbracket^\odot$$

  with l, $fail$ and $succ$ fresh; the reasoning is similar to above.

- Case $\alpha = n_\lambda \succ m$. Consider the context

$$L = \left[\nu\, x.\left(x \mid \begin{array}{l} x.\mathsf{link}\, m.(\mathsf{spawn}\, m.(\mathsf{unlink}\, n.\mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ_1})) \mid \\ x.\mathsf{spawn}\, m.(\mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ_2})) \end{array}\right)\right]^n_\lambda$$

  with l, $fail$, $succ_1$ and $succ_2$ fresh. Now, since $S \dot{\approx} R$ their public network must agree on aliveness of localities, links, and views. There are two cases to be considered: when the link $n \leftrightarrow m$ is alive is S and when it is not.

– Case $n \leftrightarrow m$.
Consider the transition $S \parallel L \Longrightarrow T_1 \equiv \nu\, x.S^l \parallel [\![\, x.\mathsf{spawn}m.\mathsf{create}l.\overline{fail} \mid fail.\overline{succ} \,]\!]^n_\lambda \parallel [\![\, \overline{succ_1} \,]\!]^l$. Now, since $S \dot{\approx} R$ and $T_1{\downarrow_{succ_1@l}}$ and $T_1 \not\downarrow_{fail@l}$ we have a matching transition $R \parallel L \Longrightarrow T_2$ s.t. $T_2{\downarrow_{succ_1@l}}$ and $T \not\downarrow_{fail@l}$. Hence we have $T_2 \equiv \nu\, x.R'^l \parallel [\![\, x.\mathsf{spawn}\ m.\mathsf{create}\ l.\overline{fail} \mid fail.\overline{succ} \,]\!]^n_\lambda \parallel [\![\, \overline{succ_1} \,]\!]^l$ (x is fresh in R). This is possible only if $R \stackrel{n_\lambda \succ m}{\Longrightarrow} R'$.
Now, $\nu\, x.S^l \parallel [\![\, x.\mathsf{spawn}\ m.\mathsf{create}\ l.\overline{fail} \mid fail.\overline{succ} \,]\!]^n_\lambda \parallel [\![\, \overline{succ_1} \,]\!]^l \sim S^l \parallel [\![\, \overline{succ_1} \,]\!]^l$, because $x \notin fn(S)$.
By using a similar reasoning for T_2 and transitivity of $\dot{\approx}$ we have $(S'^l \parallel [\![\, \overline{succ_1} \,]\!]^l_\lambda, R'^l \parallel [\![\, \overline{succ_1} \,]\!]^l_\lambda) \in \dot{\approx}$. Finally by Proposition 10 we have $(S', R') \in \dot{\approx}$ as required.

– Case $n \not\leftrightarrow m$.
Consider the transition $S \parallel L \Longrightarrow T_1 \equiv S^l \parallel [\![\, x.\mathsf{link}m.(\mathsf{spawn}m.(\mathsf{unlink}n.\mathsf{create}l.(\overline{fail} \mid fail.\overline{succ_1})) \,]\!]^n_\lambda \parallel [\![\, \overline{succ_2} \,]\!]^l$. Now, since $S \dot{\approx} R$ and $T_1{\downarrow_{succ_2@l}}$ and $T_1 \not\downarrow_{fail@l}$ we have a matching transition $R \parallel L \Longrightarrow T_2$ s.t. $T_2{\downarrow_{succ_2@l}}$ and $T \not\downarrow_{fail@l}$. Hence we have $T_2 \equiv R'^l \parallel [\![\, x.\mathsf{link}\ m.(\mathsf{spawn}\ m.(\mathsf{unlink}\ n.\mathsf{create}\ l.(\overline{fail} \mid fail.\overline{succ_1})) \,]\!]^n_\lambda \parallel [\![\, \overline{succ_2} \,]\!]^l$. This is possible only if $R \stackrel{n_\lambda \succ m}{\Longrightarrow} R'$.
Now, $\nu\, x.S^l \parallel [\![\, \overline{succ_2} \,]\!]^l \parallel [\![\, x.\mathsf{link}\ m.(\mathsf{spawn}\ m.(\mathsf{unlink}\ n.\mathsf{create}\ l.(\overline{fail} \mid fail.\overline{succ_1})) \,]\!]^n_\lambda \sim S^l \parallel [\![\, \overline{succ_2} \,]\!]^l$, because $x \notin fn(S)$.
By using a similar reasoning for T_2 and transitivity of $\dot{\approx}$ we have $(S'^l \parallel [\![\, \overline{succ_2} \,]\!]^l, R'^l \parallel [\![\, \overline{succ_2} \,]\!]^l) \in \dot{\approx}$. Finally by Proposition 10 we have $(S', R') \in \dot{\approx}$ as required.

□

F Bisimulation Of Running Example

In this section we show that a bisimulation relating the example without failure and the example with recovery.

We recall briefly the two systems.

$$\mathbf{servD} = \nu\, n_r, n_b, r_1, r_2, b.\Delta \triangleright [\![I]\!]^{n_i} \parallel [\![R]\!]^{n_r} \parallel [\![B]\!]^{n_b}$$

$$\mathbf{servDFR} = \nu\, n_r, n_b, n_c, r_1, r_2, b, c, retry.\ \Delta' \triangleright \left(\begin{array}{l} [\![K]\!]^{n_i} \parallel [\![R]\!]^{n_r} \parallel [\![\mathsf{kill}]\!]^{n_r} \parallel [\![!B]\!]^{n_b} \parallel [\![C]\!]^{n_c} \end{array} \right)$$

where:

$$\begin{aligned} C &= \nu\, c.(\overline{c} \mid !c.(\mathsf{create}\ n_r.(R \mid \mathsf{spawn}\ n_i.\overline{retry}) \mid \overline{c}) \\ I &= req(y,z).\mathsf{spawn}\ n_r.\overline{r_1}\langle y, z\rangle \\ R &= (r_1(y,z).\mathsf{spawn}\ n_b.\overline{b}\langle y, z\rangle) \mid (r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle) \\ B &= b(y,z).\mathsf{spawn}\ n_r.\overline{r_2}\langle z, w_y\rangle \\ K &= req(y,z).((\mathsf{spawn}\ n_r.\overline{r_1}\langle y, c\rangle) \mid c(w).\overline{z}\langle w\rangle \mid retry.\mathsf{spawn}\ n_r.\overline{r_1}\langle y, c\rangle) \end{aligned}$$

Proposition 11. ***servD*** $\approx$ ***servDFR***

Proof. In the following, for simplicity reasons, we will omit in the action label the incarnation number of the location doing the action.

Consider relation $\mathcal{R} = \{(\mathbf{servD}, \mathbf{servDFR})\} \cup \mathcal{S}_0 \cup \mathcal{S}_1 \cup \mathcal{S}_2$ where

$$\mathcal{S}_0 = \{(\mathbf{servD}, R_0) \mid \mathbf{servDFR} \stackrel{\tau}{\Longrightarrow} R_0\}$$

$$\mathcal{S}_1 = \{(S_1, R_1) \mid (S_0, R_0) \in \mathcal{S}_0, S_0 \stackrel{req(x,y)@n_i}{\Longrightarrow} S_1 \wedge R_0 \stackrel{req(x,y)@n_i}{\Longrightarrow} R_1\}$$

$$\mathcal{S}_2 = \{(S_2, R_2) \mid (S_1, R_1) \in \mathcal{S}_1, S_1 \stackrel{\overline{z}\langle w\rangle@n_1}{\Longrightarrow} S_1 \wedge R_1 \stackrel{\overline{z}\langle w\rangle@n_1}{\Longrightarrow} R_2\}$$

Now we analyze the moves of the various pairs to show that indeed $\mathcal{R}$ is a bisimulation.

- Pair $(\mathbf{servD}, \mathbf{servDFR})$. Now we proceed by case analysis on the possible transitions.
  - Case $x(y,z)@n_i$.
    - Case $\mathbf{servD} \xrightarrow{x(y,z)@n_i} T_1$.
      Consider transition

      $$\mathbf{servD} \xrightarrow{x(y,z)@n_i} T_1 \equiv \nu\, n_r, n_b, r_1, r_2, b.\Delta \triangleright [\![\mathsf{spawn} n_r.\overline{r_1}\langle y, z\rangle]\!]^{n_i} \parallel [\![R]\!]^{n_r} \parallel [\![B]\!]^{n_b}$$

      The move can then be matched by

      $$\begin{array}{l} \mathbf{servDFR} \stackrel{x(y,z)@n_i}{\Longrightarrow} \\ T_2 \equiv \nu\, n_r, n_b, n_c, r_1, r_2, b, c, retry.\ \Delta' \triangleright \\ \left(\begin{array}{l} [\![(\mathsf{spawn}\ n_r.\overline{r_1}\langle y, c\rangle) \mid c(w).\overline{z}\langle w\rangle \mid retry.\mathsf{spawn}\ n_r.\overline{r_1}\langle y, c\rangle]\!]^{n_c} \\ \parallel [\![R]\!]^{n_r} \parallel [\![\mathsf{kill}]\!]^{n_r} \parallel [\![!B]\!]^{n_b} \parallel [\![C]\!]^{n_c} \end{array} \right) \end{array}$$

      and $(T_1, T_2) \in \mathcal{R}$.

∗ Case **servDFR** $\xrightarrow{x(y,z)@n_i}$ T_1. Similar to above.

– Case τ.

∗ Consider transition

$$\begin{array}{l}\mathbf{servDFR} \xrightarrow{\tau} \\ T_1 \equiv \nu\, n_b, n_r, b, s, retry, n_c, c.\Delta \rhd \\ \lfloor K \rfloor^{n_i} \parallel \lfloor R \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_2} \parallel \lfloor !B \rfloor^{n_b} \parallel \\ \lfloor \overline{c}\, |!c.(\mathsf{create}\ n_r.(R \mid \mathsf{spawn}\ n_i.\overline{retry}) \mid \overline{c}) \rfloor^{n_c}\end{array}$$

Then, the move can be matched by the other system by doing nothing, indeed $(\mathbf{servD}, T_1) \in \mathcal{R}$

∗ Consider transition

$$\begin{array}{rl}\mathbf{servDFR} \xrightarrow{\tau} & T_1 \equiv \nu\, n_r, n_b, a, b, s, retry, n_c.\Delta - (n_b, \lambda_b) \rhd \\ & \lfloor K \rfloor^{n_i} \parallel \lfloor R \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c}\end{array}$$

Then, the move can be matched by the other system by doing nothing, indeed $(\mathbf{servD}, T_1) \in \mathcal{R}$

- Pairs $(S_1, R_1) \in \mathcal{S}_1$

Case τ Here, any possible τ transition can be matched by the other system by doing nothing.

Case $\overline{z}\langle w\rangle@n_i$ Here there is only one possible system S that can perform the step $\overline{z}\langle w\rangle@n_i$, that is

$$\nu\, \tilde{u}.\Delta \rhd \lfloor \overline{z}\langle w\rangle \rfloor^{n_1}$$

Now, there are an unbounded number of different system R_1 that can match the move, according to the state of the recovery. We only show the following example. Consider system

$$\begin{array}{l}R_1 \equiv \nu\, n_r, n_b, n_c, r_1, r_2, b, c, retry.\Delta'' \rhd \\ \left(\begin{array}{l}\lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor retry.\mathsf{spawn}\ n_r.\overline{r_1}\langle y, c\rangle \rfloor^{n_i} \parallel \\ \lfloor \overline{retry} \mid R \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c}\end{array}\right)\end{array}$$

In R_1 the failure has occurred (we omit dead locations) and the recovery process has

started already. By doing the following steps R can match the move.

$$
\begin{aligned}
& R_1 \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor retry.\mathsf{spawn}\ n_r.\overline{r_1}\langle y,c\rangle \rfloor^{n_i} \parallel \lfloor \overline{retry} \rfloor^{n_r} \parallel \lfloor R \rfloor^{n_r} \parallel \\
& \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \mathsf{spawn}\ n_r.\overline{r_1}\langle y,c\rangle \rfloor^{n_i} \parallel \lfloor R \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \overline{r_i}\langle y,c\rangle \rfloor^{n_r} \parallel \lfloor R \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangle\ & \left(\begin{array}{l} \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \overline{r_1}\langle y,c\rangle \rfloor^{n_r} \parallel \lfloor r_1(y,z).\mathsf{spawn}\ n_b.\overline{b}\langle y,z\rangle \rfloor^{n_r} \parallel \\ \lfloor r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \end{array} \right) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \left(\begin{array}{l} \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \mathsf{spawn}\ n_b.\overline{b}\langle y,c\rangle \rfloor^{n_r} \parallel \lfloor r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle \rfloor^{n_r} \parallel \\ \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \end{array} \right) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \left(\begin{array}{l} \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \overline{b}\langle y,c\rangle \rfloor^{n_b} \parallel \lfloor r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle \rfloor^{n_r} \parallel \\ \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \end{array} \right) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \left(\begin{array}{l} \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \overline{b}\langle y,c\rangle \rfloor^{n_b} \parallel \lfloor r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle \rfloor^{n_r} \parallel \\ \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor b(y,z).(!B \mid \mathsf{spawn}\ n_r.\overline{r_2}\langle z,w_y\rangle) \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \end{array} \right) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \left(\begin{array}{l} \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \\ \lfloor !B \mid \mathsf{spawn}\ n_r.\overline{r_2}\langle c,w_y\rangle) \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \end{array} \right) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \left(\begin{array}{l} \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \\ \lfloor !B \rfloor^{n_b} \parallel \lfloor \mathsf{spawn}\ n_r.\overline{r_2}\langle c,w_y\rangle \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c} \end{array} \right) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & \left(\begin{array}{l} \lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \\ \lfloor !B \rfloor^{n_b} \parallel \lfloor \overline{r_2}\langle c,w_y\rangle \rfloor^{n_r} \parallel \lfloor C \rfloor^{n_c} \end{array} \right) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & (\lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \mathsf{spawn}\ n_i.\overline{c}\langle w_y\rangle \rfloor^{n_r} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c}) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & (\lfloor c(w).\overline{z}\langle w\rangle \rfloor^{n_i} \parallel \lfloor \overline{c}\langle w_y\rangle \rfloor^{n_i} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c}) \xrightarrow{\tau} \\
\nu\,\tilde{u}.\Delta'' \triangleright\ & (\lfloor \overline{z}\langle w_y\rangle \rfloor^{n_i} \parallel \lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c}) \xrightarrow{\overline{z}\langle w_y\rangle @ n_i} \\
& T_2 \equiv \nu\,\tilde{u}.\Delta'' \triangleright (\lfloor \mathsf{kill} \rfloor^{n_r} \parallel \lfloor !B \rfloor^{n_b} \parallel \lfloor C \rfloor^{n_c})
\end{aligned}
$$

with $(T_1, T_2) \in \mathcal{R}$

- Pairs $S_2 \times R_2$

Case τ Here there exist infinite R_2 that can perform a τ (the controller still attempting to recreate the location), anyway all the moves can matched by S_2 by doing nothing.

□

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399