Logical Methods in Computer Science
Volume 21, Issue 3, 2025, pp. 1:1–1:52
https://lmcs.episciences.org/

Submitted Jun. 21, 2024
Published Jul. 01, 2025

A BEHAVIORAL THEORY FOR DISTRIBUTED SYSTEMS WITH WEAK RECOVERY

GIOVANNI FABBRETTI [a], IVAN LANESE [b], AND JEAN-BERNARD STEFANI [a]

[a] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
e-mail address: giovanni.fabbretti@inria.fr, jean-bernard.stefani@inria.fr

[b] Olas Team, Univ. of Bologna/INRIA, 40137 Bologna, Italy
e-mail address: ivan.lanese@gmail.com

Abstract. Distributed systems can be subject to various kinds of partial failures, therefore 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 with recovery. The recovery model considered in the paper is *weak*, in the sense that it makes no assumption on the exact state in which a failed node resumes its execution, only its identity has to be distinguishable from past incarnations of itself. Our calculus is inspired in part by the Erlang programming language and in part by the distributed π-calculus with nodes and link failures (DπF) introduced by Francalanza and Hennessy. In order to reason about distributed systems with failures and recovery we develop a behavioral theory for our calculus, in the form of a contextual equivalence, and of a fully abstract coinductive characterization of this equivalence by means of a labelled transition system semantics and its associated weak bisimilarity. This result is valuable for it provides a compositional proof technique for proving or disproving contextual equivalence between systems.

1. Introduction

Goal and motivations. 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 or management system intervention) is an important model of failure to consider because of its relevance in practice. For instance, it is the failure model assumed by the Paxos protocol used in many cloud services around the world [Pax], and the basic failure model considered by the Kubernetes cloud configuration system [Kub23] for its self-healing facility. However, the correct design of systems based on this model is far from trivial, as a recent study on

Partially supported by the EU Marie Skłodowska-Curie action ReGraDe-CS (No: 101106046), by French ANR project SmartCloud ANR-23-CE25-0012, by French ANR project DCore ANR-18-CE25-0007, and by INdAM – GNCS 2024 project MARVEL, code CUP E53C23001670001.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-21(3:1)2025

bugs affecting crash recovery in distributed systems demonstrates [GDQ+18]. Developing a process calculus analysis 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, testers and debuggers.

Unfortunately, the literature contains precious few examples of process calculi accounting for crash failures with recovery, and those that do [FGL+96, Ama97, BH00, BLTV23] are not really satisfactory: they fail to support what we call a weak recovery model as well as two other key features of actual distributed systems, namely dynamic nodes and links, and imperfect knowledge of their environment. The goal of this paper, then, is to introduce a process calculus exhibiting the following key features:

(1) *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.
(2) *Crash failures*: when a node or a link fails it does so silently. A failed link simply ceases to suppport communication between nodes. A failed node simply ceases to execute the processes it hosts.
(3) *Imperfect knowledge*: in general, in a distributed system, a node has only a partial and possibly outdated knowledge of the overall state of the system.
(4) *Weak recovery model*: when a link fails it can be re-established. When a node fails it can be restarted with an arbitrary, possibly empty, set of hosted processes. There is no assumption that the state of hosted processes prior to the node crash has been preserved, only that it is possible to distinguish previous failed instances of a node from the current running one.

Our calculus is positioned so that it stays close to the DπF process calculus by Francalanza and Hennessy [FH08] while being faithful to the failure behavior of systems built with the Erlang [Arm07] programming language. We do so for the following reasons. On the one hand, DπF constitutes a good benchmark for comparison. While not tackling recovery, it exhibits 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. On the other hand, Erlang and its environment are representative of modern distributed programming facilities. Erlang is a functional, concurrent and distributed language based on the actor model, and it is used in several large distributed projects [Ces19]. 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, and restarted by its supervisor. Staying close to Erlang ensures our modeling remains faithful to actual distributed systems.

The table below summarizes our analysis of the state of the art with respect to the key features mentioned above.

	[FH08]	[FGL+96]	[Ama97]	[BH00]	[BLTV23]
Dynamic nodes	yes	yes	yes	no	no
Dynamic links	partially	no	no	no	no
Crash failures	yes	yes	yes	yes	yes
Imperfect knowledge	partially	no	no	no	no
Weak recovery model	no	no	no	no	no

In DπF [FH08] fresh links can be established only when a new location is created and only between the new location and other locations reachable from the creating one. Moreover, once a link between two locations is broken it cannot be restored. Hence, the partial meeting of the dynamic link criterion. Also, in DπF, locations can have a partial view of the system but cannot hold a (wrong) belief on the status of a remote location, hence the partial meeting of the imperfect knowledge criterion.

Approach and contributions. Coming up with a small calculus, meeting all the above criteria, is far from trivial because of the interplay between the different features we target. Highlighting key elements of our approach can give a sense of the difficulties.

The starting point for our process calculus, which we call DπFR for Distributed π-calculus with Failures and Recovery, is the DπF calculus presented in [FH08], however our technical developments are markedly different.

First, DπF is a dynamically-typed calculus: whole systems are typed by the public part of the network of locations and links on which they run, and new location names are typed with their status (alive or dead) and the links connecting new locations to other existing ones in the network (locations are abstractions of nodes in DπF and DπFR). Types in DπF are annotations that encode information about the network structure. As such they do not constrain DπF systems, but they constrain their equivalence: two equivalent systems in DπF are by definition equally typed, and thus their networks have the same public part. But this is precisely a result which we would expect to derive from a behavioral theory (cf. Proposition 4.13). So we opt instead for an untyped approach, where we can indeed prove such a result for (strongly) equivalent systems.

Second, the dynamicity of networks in DπF is limited by the fact that when creating a new location, links can only be established with existing locations which are reachable by the creating location. Hence, if a location is or becomes disconnected from all the other locations in a network, it will remain so forever. But if we want to allow for link recovery this should clearly not be the case. DπFR thus allows for links to be (re)established freely, with no constraint on reachability. As a result, as we will see later in Section 5 (cf. Example 5.5), our behavioral theory is more discriminating than that of DπF even in absence of location recovery, which is an unexpected result.

Third, the types in DπF induce complexity in the handling of their restriction operator and in the definition of a fully abstract labelled transition system semantics and a weak bisimilarity equivalence. Francalanza and Hennessy argue that this leads to gains in the form of reduced sizes of the bisimulation relations they need to consider in proofs of equivalence between systems. However, they can obtain this only because their network (modulo the creation of fresh nodes) can only diminish. In our setting, since nodes and links can be reestablished, their approach would not work anyway. We thus introduce in DπFR a novel handling of restriction, eschewing in particular traditional close rule for scope extrusion from the π-calculus, and gaining in the process simpler developments for our behavioral theory (a simpler labelled transition system semantics, a simpler notion of weak bisimulation, and a much simpler proof of completeness for our weak bisimilarity equivalence).

A key construct for supporting recovery in DπFR is the notion of *incarnation number*. Briefly, each location in DπFR is identified by a name and an incarnation number. When a failed location recovers, its incarnation number is incremented, to distinguish the recovered location from its previous instances. We introduce this notion in DπFR for two main reasons. First, we do so to faithfully reflect the behavior of Erlang systems in presence of failure. Incarnation numbers are called creation numbers in Erlang [Erlb] and allow the Erlang environment to safely drop messages issued by a previous incarnation of a node, avoiding message confusion across different incarnations of the same node. Second, incarnation numbers are key elements used in several failure recovery schemes and distributed algorithms. Under various names (epoch numbers, incarnation numbers) they are required in several rollback-recovery schemes surveyed in [EAWJ02], including optimistic recovery schemes [SY85, DTG99] and causal logging schemes [EZ92]. They are also a key ingredient in several distributed algorithms such as, e.g., algorithms for scalable distributed failure detection [GCG01], membership management [DGM02], and diskless crash-recovery [MPSS17]. With this notion, in DπFR we have basic support in place for encoding these recovery schemes and algorithms.

To summarize, the key contributions of our work are as follows:

(1) We formalize in DπFR support for a weak recovery model with incarnation numbers, commonly found in actual systems such as Erlang ones, and at the basis of several recovery schemes which have been proposed in the literature.
(2) To reason about failures and recoveries in distributed systems we develop a behavioral theory for DπFR including a contextual equivalence in the form of a weak barbed congruence, and its coinductive fully abstract characterization in the form of a labelled transition system semantics and its weak bisimilarity equivalence. Our behavioral theory agrees with the one for DπF on key examples without recovery.

Organization. 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 and its reduction semantics. Section 4 equips our calculus with a notion of weak barbed congruence and characterizes it by a notion of weak bisimilarity to obtain a proof technique for checking system equivalence. Section 5 shows our behavioral theory in action on the motivating example from Section 2 and on other examples from [FH08]. Section 6 discusses key design choices in the definition of our calculus. Section 7 discusses the experiments we did with Erlang to validate our semantics. Finally, Section 8 discusses related work and concludes. Details of proofs and complementary material are available in the Appendices.

2. Crash and Recovery: Motivating Example

Before presenting our calculus, we discuss a motivating example which serves also as an informal introduction to the DπFR calculus (for the time being, we omit some details). An Erlang implementation of the example is available in the companion repository [Erl23].

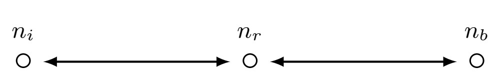
Consider the following system:

$$\mathbf{servD} \stackrel{\text{def}}{=} \nu\, \tilde{u}.\Delta \triangleright [\, I\,]^{n_i} \parallel [\, R\,]^{n_r} \parallel [\, B\,]^{n_b}$$

where:

$$\begin{aligned}
&\tilde{u} = n_r, n_b, r_1, r_2, b \\
&I = req(y,z).\mathsf{spawn}\ n_r.\overline{r_1}\langle y, z\rangle \qquad B = b(y,z).\mathsf{spawn}\ n_r.\overline{r_2}\langle z, w_y\rangle \\
&R = (r_1(y,z).\mathsf{spawn}\ n_b.\bar{b}\langle y,z\rangle) \mid (r_2(z,w).\mathsf{spawn}\ n_i.\overline{z}\langle w\rangle)
\end{aligned}$$

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 the processes inside the location cease to function. Located processes take the form $[\,P\,]^n$, where P is a process and n is the location name. There can be several processes located at the same location: for instance, $[\,P\,]^n \parallel [\,Q\,]^n$ 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 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**

$$[\,\mathsf{spawn}\ n_i.(\overline{req}\langle h, z\rangle \mid z(w).Q)\,]^{n_k}$$

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:

$$\mathbf{servDF} \stackrel{\text{def}}{=} \nu\,\tilde{u}, n_c, retry.\Delta' \triangleright [\,J\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!C\,]^{n_c}$$

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 another location 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* $[\,!C\,]^{n_c}$ 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, which makes available an unbounded number of copies of the trailing process), 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, sending also a message $retry$ to communicate to the interface n_i to 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 a 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:

$$\mathbf{servDFR} \stackrel{\text{def}}{=} \nu\, \tilde{u}, n_c, retry, c.\Delta' \triangleright [\, K\,]^{n_i} \parallel [\, R\,]^{n_r} \parallel [\, !B\,]^{n_b} \parallel [\, \mathsf{kill}\,]^{n_r} \parallel [\, !C\,]^{n_c}$$

where:

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

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 (according to the terminology in [Gär99]), equivalent to the ideal system **servD**. This claim could also be formally verified by leveraging in our setting the work done in [FH07], in which a behavioral theory capable of discriminating systems up to n faults is given. 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 names*, 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*. Generally, we span over C with x, y, z and their decorated versions, over N with l, m, n and their decorated versions, over I with ι and its decorated versions, and over $\mathbb{Z}$ with λ, κ and their decorated versions. However, in some examples we abuse the notation and use evocative words or other identifiers to span over C and N to help the readability of the systems that we describe. 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 $\odot \notin \mathsf{N}$ and the symbol $\odot$ identifies a special location that cannot be killed. 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 incarnation numbers cannot. 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 number of elements. 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. If $f : \mathsf{A} \to \mathsf{B}$ is a function from A to B, we write $f[a \mapsto b]$ for the function from A to B that agrees with f on all elements of A different from a, and maps a to b, i.e. $\forall x \in \mathsf{A} \setminus \{a\}, f[a \mapsto b](x) = f(x)$, and $f[a \mapsto b](a) = b$.

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:

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

where

$$\begin{array}{l} \tilde{u} \subset \mathsf{C} \cup \mathsf{N}^{\odot} \cup \mathsf{I} \cup \mathbb{Z} \quad r, s \in \mathsf{C} \cup \mathsf{N}^{\odot} \cup \mathsf{I} \cup \mathbb{Z} \quad \tilde{v} \subset \mathsf{C} \cup \mathsf{N} \cup \mathsf{I} \\ w \in \mathsf{C} \cup \mathsf{N} \quad x \in \mathsf{C} \quad n \in \mathsf{N} \quad \iota \in \mathsf{I} \end{array}$$

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

The syntax of processes is that of the π-calculus with matching [SW01] and replicated receivers (first line of productions), extended with primitives for distributed computing inspired from the Erlang programming language (second line of productions), and primitives to activate locations, establish and remove links (third line of productions).

$\mathbf{0}$ is the null process which can take no action. Two processes $\overline{x}\langle\tilde{v}\rangle.P$ and $x(\tilde{u}).Q$ *residing on the same location* can communicate synchronously. More precisely, process $\overline{x}\langle\tilde{v}\rangle.P$ is an output on channel x of a tuple of values $\tilde{v}$ with continuation P. Dually, $x(\tilde{u}).Q$ receives on channel x a tuple of values and replaces them for elements of $\tilde{u}$. Q is the continuation. We write $\overline{x}\langle\tilde{u}\rangle$ for $\overline{x}\langle\tilde{u}\rangle.\mathbf{0}$, 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, as in [FH08], replicates itself once and then awaits for a local message on channel x before replicating again. 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 $\nu\, u_1. \dots \nu\, u_n.P$ if $\tilde{u} = (u_1, \dots, u_n)$. If $\tilde{u}$ is empty, $\nu\, \tilde{u}.P$ is just P. The construct $\textsf{if } r = s \textsf{ then } P \textsf{ else } Q$ 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 $\textsf{node}(n, \iota).P$ gives access to the name of the current location and its incarnation number, substituting them for n and ι inside the continuation P. Primitive $\textsf{forget}\, n.P$ removes the reference to the location with name n from the local view of the current location and continues as P. This primitive in Erlang is used to remove a connection to a remote location. Primitive $\textsf{spawn}\, n.P$ launches process P at the location named n, if the latter is accessible. Primitive $\textsf{kill}$ stops the current location in its current incarnation: no process can execute on a killed location; $\textsf{kill}$ models both a programmed stop and the crash of a location. Primitive $\textsf{create}\, n.P$ creates a new location n, or reactivates a killed location with a new incarnation number, and launches process P on it. Primitive $\textsf{link}\, n.P$ creates a connection between the current location and location n and continues as P, while, $\textsf{unlink}\, n.P$ breaks the link between the current location and location n and continues as P; $\textsf{unlink}$ also models the failure of a link.

The syntax of *configurations* is defined as follows:

$$L, M, N ::= \mathbf{0} \;\big|\; [\, P\,]^n_\lambda \;\big|\; \langle (m, \kappa) : P \rangle^n_\lambda \;\big|\; N \parallel M \qquad \text{where } n, m \in \mathsf{N}^{\odot} \quad \lambda, \kappa \in \mathbb{N}^+$$

A configuration can be the empty configuration $\mathbf{0}$, a located process $[\, P\,]^n_\lambda$, a spawning message $\langle (m, \kappa) : P \rangle^n_\lambda$, 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. A spawning message $\langle(m,\kappa) : P\rangle^n_\lambda$ denotes a message sent by location n in its incarnation λ to spawn process P on the target location m in its incarnation κ. 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 location. We denote by $\mathbb{L}$ the set of configurations.

The syntax of *systems* is defined as follows:

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

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 triple $\langle \mathcal{A}, \mathcal{L}, \mathcal{V}\rangle$ where

- $\mathcal{A} : \mathsf{N}^\odot \to \mathbb{Z}$ is a function 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. In examples, the $\mathcal{A}$ component of a network is represented as a finite set of pairs of the form $n \mapsto \lambda$, where $n \in \mathsf{supp}(\mathcal{A})$ and $\lambda = \mathcal{A}(m)$ (we usually omit the pair $\odot \mapsto 1$ from network descriptions, as it is systematically present, by definition).
- $\mathcal{L} \subseteq \mathsf{N}^\odot \times \mathsf{N}^\odot$ is the set of links between locations. $\mathcal{L}$ is a finite symmetric binary relation over location names. Since it is symmetric we model it as a set of unordered pairs of the form $n \leftrightarrow m$, namely $n \leftrightarrow m$ and $m \leftrightarrow n$ are the same element. We require $\mathsf{dom}(\mathcal{L}) \stackrel{\text{def}}{=} \{n \in \mathsf{N}^\odot \mid \exists m.n \leftrightarrow m \in \mathcal{L}\}$ to be 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 n believes location m to be alive at incarnation κ. If $\mathcal{V}(n)(m) = 0$, then location n holds no belief on the status of location m. In examples, the $\mathcal{V}$ component of a network is represented as a finite set of pairs of the form $n \mapsto \mathcal{V}(n)$, where $\mathcal{V}(n)$ is represented as a finite set of pairs of the form $m \mapsto \kappa$.

For convenience we use $\Delta_\mathcal{A}$, $\Delta_\mathcal{L}$, and $\Delta_\mathcal{V}$ to denote the individual components of a network 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 $n \leftrightarrow m \in \Delta_\mathcal{L}$

Example 3.1 (Network Representation)**.** In the example in Section 2, the network Δ, on which the system **servD** runs, can be defined as follows:

$$\begin{aligned}
\Delta_\mathcal{A} &= \{n_i \mapsto 1, n_r \mapsto 1, n_b \mapsto 1\}\\
\Delta_\mathcal{L} &= \{n_i \leftrightarrow n_r, n_r \leftrightarrow n_b\}\\
\Delta_\mathcal{V} &= \{n_i \mapsto \hat{\mathbf{0}}, n_r \mapsto \hat{\mathbf{0}}, n_b \mapsto \hat{\mathbf{0}}\}
\end{aligned}$$

assuming that locations have no belief on other locations.

We now define update operations over a network Δ:

Definition 3.2 (Network Updates)**.** Network update operations are defined as follows:

- $\Delta \oplus n \leftrightarrow m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}} \cup \{n \leftrightarrow m\}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \ominus n \leftrightarrow m = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}} \setminus \{n \leftrightarrow m\}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \oplus (n, \lambda) = \langle \Delta_{\mathcal{A}}[n \mapsto \lambda], \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}}[n \mapsto \hat{\mathbf{0}}] \rangle$
- $\Delta \ominus (n, \lambda) = \langle \Delta_{\mathcal{A}}[n \mapsto -\lambda], \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}} \rangle$
- $\Delta \oplus n \succ (m, \lambda) = \langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{L}}, \Delta_{\mathcal{V}}[n \mapsto \Delta_{\mathcal{V}}(n)[m \mapsto \lambda]] \rangle$, 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 resets 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 notions of *closed* and *well-formed* system, which we now define:

Definition 3.3. A system $S = \nu\, \widetilde{w}.\Delta \triangleright M \in \mathbb{S}$ is *closed* iff it does not have any free incarnation variable. It is *well-formed* iff the belief that each location n has on any remote location m is less or equal than the current incarnation number of m, that is $\Delta_{\mathcal{V}}(n)(m) \leq |\Delta_{\mathcal{A}}(m)|$, and for any occurrence of the spawning message $\langle (m, \kappa) : P \rangle^n_\lambda$ in M, we have $\lambda \leq |\Delta_{\mathcal{A}}(n)|$ and $\kappa \leq |\Delta_{\mathcal{A}}(m)|$, where $|\cdot|$ computes the absolute value.

The notion of well-formed system ensures that spawning messages in a system are consistent with the state of its network, and in particular that they do not come from, nor target, locations at future incarnations. From now on, we will only consider well-formed systems.

The definition of free incarnation 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 functions inside the network. However, free names can be easily defined by leveraging two auxiliary functions that return, respectively, the support set and the domain set of a given function. Details are given in Appendix A. Throughout the entire paper we use the Barendregt convention, that is if some terms occur in some context, then in these terms all bound variables are chosen to be different from the free ones.

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 well-formed 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 since it acts also on

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

[S.Res.C] $\nu\, u.\nu\, v.S \equiv \nu\, v.\nu\, u.S$ [S.Res.Nil] $\dfrac{u \notin \mathsf{fn}(S)}{\nu\, u.S \equiv S}$ [S.α] $\dfrac{S =_\alpha R}{S \equiv R}$ [S.Ctx] $\dfrac{N \equiv M}{\mathbb{C}[N] \equiv \mathbb{C}[M]}$

Figure 1: Structural Congruence Rules

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

SPAWN-L
$$\dfrac{}{\Delta \triangleright [\, \mathsf{spawn}\ n.P \,]^n_\lambda \longrightarrow \Delta \triangleright [\, P \,]^n_\lambda}$$

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

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

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

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

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

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

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

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

Figure 2: Local Rules

Δ, which contains functions. However it can be defined straightforwardly (see Appendix A for details).

The reduction relation $\longrightarrow$ is defined by the rules in Figs. 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 spawn-l defines the local launch of process P. Rule bang defines the expansion of a replicated input process. Rule new performs the scope extrusion of a name from a process to the system level. Rule fork turns a parallel composition into parallel processes 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. Rule forget deletes from the local view of the current location the belief it may hold about a given location m. Finally, rule msg defines the receipt of a local message by an input process. We introduced rules new, bang and fork as computational steps instead of as structural congruence rules for it simplifies our proofs.

Fig. 3 depicts the *distributed* rules, i.e., rules that involve locations and spawning messages, or modify the network.

Rules link and unlink define respectively the establishment and the removal of a link. Rule create-s defines the successful creation of a location if it never existed or its reactivation if it 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. Rule create-f defines the silent failure of a create operation, which can

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

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

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

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

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

SPAWN-C-S
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \vdash n \leftrightarrow m \qquad n \neq m \qquad \Delta_{\mathcal{V}}(n)(m) = \kappa \in \mathbb{N}}{\Delta \triangleright [\,\mathsf{spawn}\; m.P\,]^n_\lambda \longrightarrow \Delta \triangleright \langle (m,\kappa) : P \rangle^n_\lambda}$$

SPAWN-C-F
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \not\vdash n \leftrightarrow m \qquad n \neq m}{\Delta \triangleright [\,\mathsf{spawn}\; m.P\,]^n_\lambda \longrightarrow \Delta \ominus n \succ m \triangleright \mathbf{0}}$$

SPAWN-S
$$\frac{\Delta_{\mathcal{A}}(m) = \kappa \qquad \kappa > 0 \qquad (\kappa^* = \kappa \vee \kappa^* = 0) \qquad \Delta \vdash n \leftrightarrow m \qquad \Delta_{\mathcal{V}}(m)(n) \leq \lambda}{\Delta \triangleright \langle (m,\kappa^*) : P \rangle^n_\lambda \longrightarrow \Delta \oplus m \succ (n,\lambda) \triangleright [\,P\,]^m_\kappa}$$

SPAWN-F
$$\frac{(\Delta_{\mathcal{A}}(m) \neq \kappa \wedge \kappa \neq 0) \ \vee\ \Delta \not\vdash n \leftrightarrow m \ \vee\ \lambda < \Delta_{\mathcal{V}}(m)(n)}{\Delta \triangleright \langle (m,\kappa) : P \rangle^n_\lambda \longrightarrow \Delta \ominus n \succ m \triangleright \mathbf{0}}$$

Figure 3: Distributed Rules

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

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

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

Figure 4: Contextual Rules

fail because the location to activate may already be alive. Rule KILL defines the killing of a location.

Rule SPAWN-C-S, for successful *spawn-commit*, generates a spawning message, where the target location comes with the incarnation number found in the local view of the spawning location. The spawning message will later be (resp. fail to be) received by the target location (rule SPAWN-S, resp. rule SPAWN-F). Rule SPAWN-C-F accounts for an unsuccessful spawn commit, due to the lack of a link between the spawning location and the target one.

Note that, because of the non atomicity of the transfer of the spawning message to the target location, the rules have to account for a possible mismatch between the incarnation number λ of the spawning location n, and the incarnation number recorded at the target location m for location n. In other words, target location m may receive a spawning message from an old incarnation of n, i.e. one which has crashed after the release of the spawning message. Thus, the spawn operation ultimately succeeds (rule SPAWN-S) only if the incarnation number λ of the spawning location n in the spawning message is greater or equal than the incarnation number $\Delta_{\mathcal{V}}(m)(n)$ currently attributed by m to n, preventing the receipt of spawning messages from a known crashed location. Arguably, this is what happens in practice in Erlang and the side conditions involving incarnation numbers in rules SPAWN-S and SPAWN-F are checks which are performed by the Erlang environment to prevent duplication of spawning messages in case of crash and subsequent recovery of a node.

Fig. 4 shows the contextual rules of our calculus. Rules PAR and RES define, respectively, parallel execution and execution under restriction. Rule STR allows the use of structural congruence before and after a reduction step. 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 component promotes a restriction at system level (via an application of rule NEW). In this case we have to avoid name capture by the idle component.

Example 3.4. Here we show some reductions in the interaction between **servD**, from Section 2, and its client $[\,\mathsf{spawn}\; n_i.(\overline{req}\langle h, z\rangle \mid z(w).Q)\,]^{n_k}$ on network Δ, defined as follows:

$$\Delta_{\mathcal{A}} = \{n_i \mapsto 1, n_r \mapsto 1, n_b \mapsto 1, n_k \mapsto 1\} \qquad \Delta_{\mathcal{L}} = \{n_i \leftrightarrow n_r, n_r \leftrightarrow n_b, n_k \leftrightarrow n_i\}$$

$$\Delta_{\mathcal{V}} = \{n_i \mapsto \hat{\mathbf{0}}, n_r \mapsto \hat{\mathbf{0}}, n_b \mapsto \hat{\mathbf{0}}, n_k \mapsto \hat{\mathbf{0}}\}$$

To ease the parsing of the reduction we underline the terms that get reduced and indicate the key reduction rule applied.

$$\nu\,\tilde{u}.\Delta \triangleright \underline{[\,\mathsf{spawn}\; n_i.(\overline{req}\langle h, z\rangle \mid z(w).Q)\,]^{n_k}} \parallel [\,req(y,z).\mathsf{spawn}\; n_r.\overline{r_1}\langle y, z\rangle\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,B\,]^{n_b} \quad \longrightarrow \quad \text{SPAWN-C-S}$$

$$\nu\,\tilde{u}.\Delta \triangleright \underline{\langle (n_i, 0) : \overline{req}\langle h, z\rangle \mid z(w).Q\rangle^{n_k}} \parallel [\,req(y,z).\mathsf{spawn}\; n_r.\overline{r_1}\langle y, z\rangle\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,B\,]^{n_b} \quad \longrightarrow \quad \text{SPAWN-S}$$

$$\nu\,\tilde{u}.\Delta' \triangleright \underline{[\,\overline{req}\langle h, z\rangle \mid z(w).Q\,]^{n_i}} \parallel [\,req(y,z).\mathsf{spawn}\; n_r.\overline{r_1}\langle y, z\rangle\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,B\,]^{n_b} \quad \longrightarrow \quad \text{FORK}$$

$$\nu\,\tilde{u}.\Delta' \triangleright \underline{[\,\overline{req}\langle h, z\rangle\,]^{n_i}} \parallel [\,z(w).Q\,]^{n_i} \parallel \underline{[\,req(y,z).\mathsf{spawn}\; n_r.\overline{r_1}\langle y, z\rangle\,]^{n_i}} \parallel [\,R\,]^{n_r} \parallel [\,B\,]^{n_b} \quad \longrightarrow \quad \text{MSG}$$

$$\nu\,\tilde{u}.\Delta' \triangleright [\,z(w).Q\,]^{n_i} \parallel [\,\mathsf{spawn}\; n_r.\overline{r_1}\langle h, z\rangle\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,B\,]^{n_b}$$

where network Δ' is similar to Δ but with the following view:

$$\Delta'_{\mathcal{V}} = \{n_i \mapsto \{n_k \mapsto 1\}, n_r \mapsto \hat{\mathbf{0}}, n_b \mapsto \hat{\mathbf{0}}, n_k \mapsto \hat{\mathbf{0}}\}$$

On the first reduction the client successfully spawns a message, bearing the request $\overline{req}\langle h, z\rangle$, towards the interface of **servD**. After the spawn, the spawn message is correctly delivered to its receiver. The freshly spawned process first action is to fork. Finally, the process descending from the fork, bearing the request, and the process awaiting for a request on the interface, synchronize, so that the request is now ready to be forwarded to the router.

A check on the consistency of our reduction semantics is given by the following proposition.

Proposition 3.5. *If S is a closed well-formed system, and $S \longrightarrow T$, then T is a closed well-formed system.*

Proof. A simple induction on the derivation of $S \longrightarrow T$, noting for well-formedness that the only rules updating the view of a location, FORGET and SPAWN-S, preserve well-formedness and ensure that incarnation numbers attributed to locations in views are actual, and that the only rule creating a new spawning message, SPAWN-C, henceforth preserves well-formedness. □

Our semantics for the parallel composition operator, coupled with the systematic presence of the distinguished location $\odot$ in any network, spares us the need to introduce a parallel composition operator 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. Indeed, given a network $\Delta = \langle \mathcal{A}, \mathcal{L}, \mathcal{V}\rangle$, we can change it to network $\Delta' = \langle \mathcal{A}', \mathcal{L}', \mathcal{V}'\rangle$, provided the following consistency constraints are respected:

- The new function $\mathcal{A}'$, for each location n, must agree with $\mathcal{A}$ on private names and must map each free location n to either the same incarnation number or to a future one. Notice that, e.g., -3 is in the future of 2 because from 2 we can get to -3 if the location fails, is re-activated and fails again. Similarly, -3 is also in the future of 3 because from 3 we can get to -3 if the location fails.

- The new link set $\mathcal{L}'$ must agree with $\mathcal{L}$ on links which have at least a private end.
- The new view function $\mathcal{V}'$ must i) agree with $\mathcal{V}$ on private names; ii) for each location n, map each belief that n has about m to a non-negative incarnation number smaller or equal than the absolute value of the incarnation number of m in $\mathcal{A}'$.

This is formalized in Proposition B.1, whose proof is available in Appendix B as well. The constraints above are required to be sure that the target system is reachable by our reduction semantics. Note that a parallel composition operator acting on systems would have to enforce analogous consistency constraints between the composed systems.

4. Behavioral Theory

In this section we present the behavioral theory for our calculus. We first define a contextual equivalence in the classical form of a weak barbed congruence, a notion originally proposed by Milner and Sangiorgi in [MS92]. We then present a labelled transition system semantics (LTS) for our calculus and we show that weak bisimilarity between (closed well-formed) DπFR systems coincides with weak-barbed congruence (our full-abstraction result). This result is valuable for it provides a compositional proof technique for proving or disproving contextual equivalence between DπFR systems. E.g., later in Section 5 we will use weak bisimilarity to prove equivalent the ideal server **servD** with the more concrete server **servDFR**, hence proving the latter correct with respect to the former. The developments and the proof of full abstraction are relatively standard but the subtlety lies in correctly accounting for the contribution of the network in the LTS semantics, even though the network is not a running process in our calculus, and in dealing with the non-standard handling of restriction.

4.1. Weak Barbed Congruence.

We define a standard notion of contextual equivalence called *weak barbed congruence*, originally proposed in [MS92]. 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 4.1 (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 $S \equiv \nu\,\tilde{u}.\Delta \triangleright [\,\overline{x}\langle\tilde{v}\rangle.P\,]^n_\lambda \parallel N$, for some $\tilde{u}, \tilde{v}, P, N$, 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 4.2 (System Congruence)**.** A binary relation $\mathcal{R}$ over closed systems is a *system congruence* iff it is an equivalence relation and 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:

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

Definition 4.3 (Weak Barb-Preserving Relation)**.** A binary 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 4.4 (Weak Reduction-Closed Relation)**.** A binary 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 $S' \;\mathcal{R}\; R'$.

We can now define weak barbed congruence.

Definition 4.5 (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:

Proposition 4.6. *Structural congruence* $\equiv$ *is a weak barb-preserving reduction-closed system congruence.*

Proof. Structural congruence is a barb-preserving system congruence by definition. Structural congruence is weak reduction-closed thanks to Propositions C.4 and C.5 in Appendix C. □

Much as in [NGP07], a simpler kind of barbs gives rise to the same barbed congruence. The alternate definition of barb we consider in this section is one where a barb just displays the location of messages.

Definition 4.7 (Location Barb)**.** We say that a system S exhibits a barb at location n, in symbols $S{\downarrow_n}$, iff $S \equiv \nu\,\tilde{u}.\Delta \triangleright [\,\overline{x}\langle\tilde{v}\rangle.P\,]^n_\lambda \parallel N$, for some $x, \lambda, \tilde{u}, \tilde{v}, P, N$, where $x, n \notin \tilde{u}$, and $\Delta \vdash n_\lambda : \mathsf{alive}$. Also, $S{\Downarrow_n}$ iff $S \Rightarrow S'$ and $S'{\downarrow_n}$. We denote by $\dot{\approx}_l$ the weak barbed congruence obtained by considering barbs at locations instead of barbs at both channels and locations.

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

Proposition 4.8 (Equivalence of barbed congruences)**.** $\dot{\approx}_l = \dot{\approx}$

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 barbs at channels and locations, we have $S \equiv \nu\,\tilde{u}.\Delta \triangleright [\,\overline{x}\langle\tilde{v}\rangle.P\,]^n_\lambda \parallel N$ for some $\tilde{u}, \tilde{v}, P, N$. 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:

$$L = [\,\mathsf{create}\,m.\mathsf{link}\,n.\mathsf{spawn}\,n.(\mathsf{node}(\_,\iota).\mathsf{if}\ \iota = \lambda\ \mathsf{then}\,x(\tilde{y}).\mathsf{spawn}\,m.\bar{t})\,]^{\odot}_1$$

$$\text{with } m, t \notin \mathsf{fn}(S) \cup \mathsf{fn}(T)$$

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

$$S \parallel L \Longrightarrow \nu\,\tilde{u}.\Upsilon \triangleright N \parallel [\,P\,]^n_\lambda \parallel [\,\bar{t}\,]^m_1 = S'$$

where Υ is Δ extended with live location m at incarnation number 1 and link $m \leftrightarrow n$. We thus have $S \parallel L \Longrightarrow S'$ and $S'{\downarrow_m}$. But since $\neg(T{\Downarrow_{x@n_\lambda}})$, and t, m are fresh for T, there can be no T' such that $T \parallel L \Longrightarrow T'$ and $T'{\Downarrow_m}$, contradicting the fact that $S\ \dot{\approx}_l\ T$. □

One may notice that in the definition of the less precise location barbs, in the barb itself, there is no mention of the incarnation number of the observed locality and wonder if an alternative definition with mention of it may have different observational power. The quick answer to this is no. Indeed, with the current definition of location barbs, even without mentioning the incarnation number in the observable, it is possible to distinguish two systems that differ because of the presence of a same locality with different incarnation numbers. The intuition is that it is always possible to build a context that grabs the incarnation number, tests it and then branches according to its value. A context of this kind is also visible in the above proof, when we need to make sure that the incarnation number of locality n is the

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

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

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

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

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

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

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

Figure 5: LTS: Local Rules

same on both systems, and we achieve so by first grabbing the incarnation number through the node primitive and then we test it with the if construct.

From now on, we will use 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 subsection we present a labeled transition semantics for our calculus in order to have a co-inductive characterization of weak barbed congruence. Working with the labeled transition semantics is more convenient as it avoids the need to work with a universal quantification over contexts. Labels α in our LTS semantics take the following forms:

$$\begin{aligned} \alpha ::= \ & \tau \;\big|\; \nu\, \tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda \;\big|\; x(\tilde{u})@n_\lambda \;\big|\; \mathsf{kill}(n, \lambda) \;\big|\; \mathsf{create}(n, \lambda) \;\big| \\ & \oplus n_\lambda \mapsto m \;\big|\; \ominus n_\lambda \mapsto m \;\big|\; n_\lambda \succ m \end{aligned}$$

The first three labels are classical: silent action, output action (possibly with restricted names), and input action. Output and input actions mention the name and incarnation number 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 at its 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.

Free names in labels are defined as follows:

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

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

Transition relation $\xrightarrow{\alpha}$ in our LTS semantics is defined inductively by several sets of inference rules. The first set of rules, shown in Fig. 5, contains the equivalent of all the local rules of the reduction relation in Fig. 2 obtained by replacing $\longrightarrow$ by $\xrightarrow{\tau}$, except for rule

L-IN
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive}}{\Delta \triangleright [\, x(\tilde{v}).P \,]^n_\lambda \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright [\, P\{\tilde{u}/\tilde{v}\} \,]^n_\lambda}$$

L-OUT
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive}}{\Delta \triangleright [\, \overline{x}\langle\tilde{v}\rangle.P \,]^n_\lambda \xrightarrow{\overline{x}\langle\tilde{v}\rangle@n_\lambda} \Delta \triangleright [\, P \,]^n_\lambda}$$

L-LINK
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \nvdash n \leftrightarrow m}{\Delta \triangleright [\, \mathsf{link}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \oplus n \leftrightarrow m \triangleright [\, P \,]^n_\lambda}$$

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

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

L-CREATE-F
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \nvdash m : \mathsf{dead}}{\Delta \triangleright [\, \mathsf{create}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright \mathbf{0}}$$

L-SPAWN-L
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive}}{\Delta \triangleright [\, \mathsf{spawn}\, n.P \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright [\, P \,]^n_\lambda}$$

L-SPAWN-C-S
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \vdash n \leftrightarrow m \qquad n \neq m \qquad \Delta_{\mathcal{V}}(n)(m) = \kappa \in \mathbb{N}}{\Delta \triangleright [\, \mathsf{spawn}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \triangleright \langle (m,\kappa) : P \rangle^n_\lambda}$$

L-SPAWN-C-F
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad \Delta \nvdash n \leftrightarrow m \qquad n \neq m}{\Delta \triangleright [\, \mathsf{spawn}\, m.P \,]^n_\lambda \xrightarrow{\tau} \Delta \ominus n \succ m \triangleright \mathbf{0}}$$

L-SPAWN-S
$$\frac{\Delta_{\mathcal{A}}(m) = \kappa \qquad (\kappa^* = \kappa \vee \kappa^* = 0) \qquad \Delta \vdash n \leftrightarrow m \qquad \Delta_{\mathcal{V}}(m)(n) \leq \lambda}{\Delta \triangleright \langle (m,\kappa^*) : P \rangle^n_\lambda \xrightarrow{\tau} \Delta \oplus m \succ (n,\lambda) \triangleright [\, P \,]^m_\kappa}$$

L-SPAWN-F
$$\frac{(\Delta_{\mathcal{A}}(m) \neq \kappa \neq 0) \ \vee\ \Delta \nvdash n \leftrightarrow m \ \vee\ \lambda < \Delta_{\mathcal{V}}(m)(n)}{\Delta \triangleright \langle (m,\kappa) : P \rangle^n_\lambda \xrightarrow{\tau} \Delta \ominus n \succ m \triangleright \mathbf{0}}$$

Figure 6: LTS: Concurrent and Distributed Rules

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
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad (\Delta_{\mathcal{V}}(n)(m) = \Delta_{\mathcal{A}}(m) \neq 0 \ \vee\ \Delta_{\mathcal{V}}(n)(m) = 0)}{\Delta \triangleright N \xrightarrow{n_\lambda \succ m} \Delta \triangleright N}$$

Figure 7: LTS: Net Rules

msg. Fig. 6 depicts classical rules for local message exchange, which are as in the standard early instantation-style LTS for π-calculus [SW01] as well as the equivalents of distributed rules in Fig. 3, obtained by replacing $\longrightarrow$ by $\xrightarrow{\tau}$.

Fig. 7 depicts the rules modeling the effects of actions from 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 is used to impose equality of views of locations for two equivalent systems. Net rules do not have a counterpart in the reduction semantics as the effect of each rule can be achieved by an appropriate context.

Fig. 8 depicts composition rules for the labeled transition semantics, we only discuss some relevant examples, the others are straightforward. Rules L-par$_{\text{L}}$ is the rule for parallel

L-PAR$_\text{L}$

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

L-PAR$_\text{R}$

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

L-SYNC$_\text{L}$

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

L-SYNC$_\text{R}$

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

L-RES

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

L-RES$_\text{O}$

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

L-α

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

Figure 8: LTS: Composition Rules

composition allowing independent evolution of the left component. The side condition on the idle (right) component is required to avoid name capture when the other component introduces a restriction. Rule L-RES$_\text{O}$ is analogous to the classical OPEN rule in 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$_\text{O}$ just signals that a system can send a message at a given address, bearing private names in its payload.

4.3. **Full Abstraction.** This subsection presents the full abstraction result. We begin by recalling the definitions of *strong* and *weak bisimilarity.*

Definition 4.9 (Strong Bisimilarity)**.** A binary relation over closed systems $\mathcal{S} \subseteq \mathbb{S}^2$ is a *strong simulation* iff whenever $(S, R) \in \mathcal{S}$, and $S \xrightarrow{\alpha} S'$ then $R \xrightarrow{\alpha} R'$ for some R' with $(S', R') \in \mathcal{S}$. A binary relation $\mathcal{S}$ over closed systems is a *strong bisimulation* if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are strong simulations. *Strong bisimilarity*, denoted by $\sim$, is the largest strong bisimulation.

Definition 4.10 (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 $R \xRightarrow{\alpha} R'$ 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.

The fact that $\sim$ and $\approx$ are equivalence relations is a standard result for any labelled transition system. Our main result states that weak bisimilarity fully characterizes weak barbed congruence:

Theorem 4.11 (Full Abstraction)**.** *$S \dot{\approx} R$ iff $S \approx R$.*

Proof. The nature of the proof consists in showing that (weak) bisimilarity is included in (weak) barbed congruence, i.e., $\approx \subseteq \dot{\approx}$, and also that (weak) barbed congruence is included in (weak) barbed bisimilarity, i.e., $\dot{\approx} \subseteq \approx$. The first inclusion, in the literature referred to as soundness of bisimilarity, is proved by showing that if two systems are (weak) bisimilar then they are also (weak) barbed congruent. The second inclusion, in the literature referred to as completeness of bisimilarity, is proved by showing that if two systems are (weak) barbed congruent then they are also bisimilar. The spirit of the second part of the proof is to be able to build contexts that force a system to perform the expected labeled action. Soundness (Proposition C.8) and Completeness (Proposition C.12) are proved in Appendix C. □

For those readers familiar with the π-calculus behavioral theory, our full abstraction result may seem surprising, but note that we are only considering congruence with respect to operators acting at the level of systems or configurations, namely the parallel operator $\|$, and the restriction operator ν. In particular we are not considering congruence with respect to the input operator (it is well known that in π-calculus weak bisimilarity is not a congruence with respect to the input operator).

We conclude the section with two results, showing respectively that processes on dead localities cannot ever reduce and that two strong equivalent systems must have the same public network.

Proposition 4.12. *Given a well-formed system* $\nu\,\tilde{u}.\Delta \triangleright N \parallel [\,P\,]^n_\lambda$ *such that* $\Delta \vdash n_\lambda : \mathsf{dead}$ *we have*

$$\nu\,\tilde{u}.\Delta \triangleright N \parallel [\,P\,]^n_\lambda \sim \nu\,\tilde{u}.\Delta \triangleright N$$

Proof. The result follows by observing that

$$\mathcal{R} = \{\langle \nu\,\tilde{u}.\Delta \triangleright N \parallel [\,P\,]^n_\lambda, \nu\,\tilde{u}.\Delta \triangleright N\rangle \mid \nu\,\tilde{u}.\Delta \triangleright N \in \mathbb{S}, \Delta \vdash n_\lambda : \mathsf{dead}\}$$

is a strong bisimulation since all the labeled transitions involving a process require for it to be alive and, hence, no transition can be derived from process $[\,P\,]^n_\lambda$. We remark that a $\mathsf{create}\,n.P$ operation builds a new location n with a higher incarnation number without affecting the state of n_λ. □

Proposition 4.13 (Strong Bisimilar Systems Have The Same Public Network)**.** *If we have* $\nu\,\tilde{u}.\Delta \triangleright N \sim \nu\,\tilde{v}.\Delta' \triangleright M$ *then the following conditions hold:*

- $\forall n \notin \tilde{u}.\Delta_{\mathcal{A}}(n) = \Delta'_{\mathcal{A}}(n)$
- $\forall n \notin \tilde{u}, m \notin \tilde{u}.(n, m) \in \Delta_{\mathcal{L}} \Leftrightarrow (n, m) \in \Delta'_{\mathcal{L}}$
- $\forall n \notin \tilde{u}, m \notin \tilde{u}.\Delta_{\mathcal{V}}(n)(m) = \Delta'_{\mathcal{V}}(n)(m)$

Proof. We just make the case for the first condition, the others are similar. Assume to have two systems $S = \nu\,\tilde{u}.\Delta \triangleright N$, $R = \nu\,\tilde{v}.\Delta' \triangleright M$ such that $S \sim R$. Now suppose, towards a contradiction, that there exists $n \notin \tilde{u}$ such that $\Delta_{\mathcal{A}}(n) \neq \Delta'_{\mathcal{A}}(n)$. Let us first consider the case where $\Delta_{\mathcal{A}}(n) \leq 0$. Hence, S can perform a labeled transition derived using rules L-CREATE-EXT and L-RES, with label $\mathsf{create}(n, |\mathcal{A}(n)| + 1)$. Such a transition cannot be matched by R, against the hypothesis that $S \sim R$. The case where $\Delta_{\mathcal{A}}(n) > 0$ is similar, using rule L-KILL-EXT instead of rule L-CREATE-EXT. Hence, $\forall n \notin \tilde{u}.\Delta_{\mathcal{A}}(n) = \Delta'_{\mathcal{A}}(n)$. □

5. Behavioral Theory in Action

In this section we show our behavioral theory in action. First we apply it to the running example of Section 2. Then, we contrast our spawn primitive against a go primitive analogous to the one from DπF. We then rephrase a key example from DπF in DπFR where our behavioral theory agrees with the one of DπF. Finally we show, also using an example from DπF, that our behavioral theory is more discriminating than that of DπF, because of the possibility in DπFR to reestablish links and to restart locations.

Example 5.1 (**servD** and **servDFR** are bisimilar)**.** To prove **servD** $\approx$ **servDFR** it suffices to show a candidate bisimulation relation and then play the bisimulation game on its elements. Consider relation

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

where

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

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

$$\mathcal{S}_2 = \{(S_2, R_2) \mid (S_1, R_1) \in \mathcal{S}_1, S_1 \xRightarrow{\overline{z}\langle w_\lambda\rangle@n_i} S_1, R_1 \xRightarrow{\overline{z}\langle w_\lambda\rangle@n_i} R_2\}$$

Intuitively, from an external perspective, **servD** only inputs the request and exhibits the answer; hence, we need to prove that **servDFR** is able to match those two actions and has no other observable behavior. The proof is available in Appendix D. $\Diamond$

Example 5.2 (Non-atomicity of the spawn primitive)**.** We now contrast our spawn primitive against the go primitive in DπF [FH08]. The semantics of the spawn primitive given by the four rules SPAWN-C-S, SPAWN-C-F, SPAWN-S and SPAWN-F is different from the semantics of the go primitive in DπF [FH08]. In our setting, a go primitive analogous to the one in DπF can be defined by the following reduction rules:

GO-S
$$\frac{\Delta \vdash n_\lambda, m_\kappa : \mathsf{alive} \qquad \Delta \vdash n_\lambda \leftrightarrow m_\kappa \qquad \Delta_n(m) = \kappa}{\Delta \triangleright [\, \mathsf{go}\, m.P\,]^n_\lambda \longrightarrow \Delta \oplus m \succ (n, \lambda) \triangleright [\, P\,]^m_\kappa}$$

GO-F
$$\frac{\Delta \vdash n_\lambda : \mathsf{alive} \qquad (\Delta \vdash m : \mathsf{dead} \vee \Delta_n(m) \neq \Delta_{\mathcal{A}}(m) \vee \Delta \not\vdash n_\lambda \leftrightarrow m_\kappa)}{\Delta \triangleright [\, \mathsf{go}\, m.P\,]^n_\lambda \longrightarrow \Delta \ominus n \succ m \triangleright \mathbf{0}}$$

Rule GO-S defines a successful go, conditional upon the fact that locations n and m are both alive and connected, and that the spawning location rightly believes the target location to be alive with the incarnation number recorded in its view, 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:

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

Rule GO-F defines a failed go. Failure may be due to the remote location not being alive, to a wrong view, or to a missing link between the two locations. The atomicity of the go operation avoids the possibility of having the spawning message delivered when the sending location is already dead. The go primitive appears to be much simpler than our spawn primitive, but unfortunately the spawn operation and the go operation lead to observationally different behaviors. To see this consider the variant $\mathrm{D}\pi\mathrm{FR}_{\mathsf{go}}$ of DπFR, where the go operation has been added. The LTS semantics for $\mathrm{D}\pi\mathrm{FR}_{\mathsf{go}}$ is identical to that of DπFR, except for the addition of rules for silent transitions corresponding to the GO-S and

GO-F rules above. The notions of weak simulation, weak bisimulation and weak bisimilarity are the same for DπFR and DπFR$_{\mathsf{go}}$. Define the encoding function $(\!|\cdot|\!)$ from DπFR$_{\mathsf{go}}$ terms to DπFR terms that just replaces any occurrence of a go instruction in a DπFR$_{\mathsf{go}}$ term by the corresponding spawn instruction. Now we have the following result:

Proposition 5.3 (go does not simulate spawn)**.** *Let U be the following closed system in DπFR$_{\mathsf{go}}$:*

$$U = \Delta \triangleright [\,\mathsf{go}\; m.\overline{s}\,]_1^n \qquad (\!|U|\!) = \Delta \triangleright [\,\mathsf{spawn}\; m.\overline{s}\,]_1^n$$

$$\Delta_{\mathcal{A}} = \{n \mapsto 1, m \mapsto 1\} \qquad \Delta_{\mathcal{L}} = \{n \leftrightarrow m\} \qquad \Delta_{\mathcal{V}} = \{n \mapsto \hat{\mathbf{0}}, m \mapsto \hat{\mathbf{0}}\}$$

U cannot weakly simulate $(\!|U|\!)$.

Proof. Define $\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5$ as the following networks:

$$\Delta_1 = \Delta \ominus (n, 1) \qquad \Delta_2 = \Delta \ominus (m, 1) \qquad \Delta_3 = \Delta_2 \oplus (m, 2)$$

$$\Delta_4 = \Delta_1 \ominus (m, 1) \qquad \Delta_5 = \Delta_4 \oplus (m, 2)$$

Assume for the sake of contradiction that there exists a weak simulation $\mathcal{S}$ such that $\langle (\!|U|\!), U\rangle \in \mathcal{S}$. Now consider the following transition from $(\!|U|\!)$, obtained by applying rule L-SPAWN-C-S :

$$(\!|U|\!) \xrightarrow{\tau} T_1 = \Delta \triangleright \langle (m, 0) : \overline{s}\rangle_1^n$$

Because $\mathcal{S}$ is a weak simulation, we must have $U \Longrightarrow U_1$ and $\langle T_1, U_1\rangle \in \mathcal{S}$ for some U_1. There are in fact only two possibilities for U_1 since U has a single silent transition, obtained by applying rule L-GO-S:

(1) $U \xrightarrow{\tau} U_1 = \Delta \triangleright [\,\overline{s}\,]_1^m$: in this case, consider the transition, obtained by applying rule L-KILL-EXT:

$$T_1 \xrightarrow{\mathsf{kill}(m,1)} T_2 = \Delta_2 \triangleright \langle (m, 0) : \overline{s}\rangle_1^n$$

Since U_1 has no silent transition, the only possibility to match this transition from T_1 is that obtained from U_1 by applying L-KILL-EXT:

$$U_1 \xrightarrow{\mathsf{kill}(m,1)} U_2 = \Delta_2 \triangleright [\,\overline{s}\,]_1^m$$

and $\langle T_2, U_2\rangle \in \mathcal{S}$. Consider now the following transition from T_2 obtained by applying rule L-CREATE-EXT:

$$T_2 \xrightarrow{\mathsf{create}(m,2)} T_3 = \Delta_3 \triangleright \langle (m, 0) : \overline{s}\rangle_1^n$$

Since U_2 has no silent transition, the only possibility to match this transition from T_2 is that obtained from U_2 by applying L-CREATE-EXT:

$$U_2 \xrightarrow{\mathsf{create}(m,2)} U_3 = \Delta_3 \triangleright [\,\overline{s}\,]_1^m$$

and $\langle T_3, U_3\rangle \in \mathcal{S}$. Now U_3 has no silent transition and has no weak barb on $\overline{s}$ (m_1 is not alive in Δ_3). But this is impossible for T_3 has a weak barb on $\overline{s}$, as can be seen from the following transitions, obtained from T_3 applying L-SPAWN-S and L-OUT:

$$T_3 \xrightarrow{\tau} \Delta_3 \triangleright [\,\overline{s}\,]_2^m \xrightarrow{\overline{s}@m_2} \Delta_3 \triangleright [\,\mathbf{0}\,]_2^m$$

(2) $U_1 = U$: in this case, consider the following transition from T_1, obtained by applying rule L-KILL-EXT:

$$T_1 \xrightarrow{\mathsf{kill}(n,1)} T_2 = \Delta_1 \rhd \langle (m,0) : \overline{s} \rangle_1^n$$

We must have $U \xRightarrow{\mathsf{kill}(n,1)} U_2$ and $\langle U_2, T_2 \rangle \in \mathcal{S}$ for some U_2. Since there is only a single silent transition from U, we have two possibilities for U_2, obtained by applying rule L-KILL-EXT, or by applying rule L-GO-S followed by rule L-KILL-EXT:

(a) $U \xrightarrow{\mathsf{kill}(n,1)} U_2 = \Delta_1 \rhd [\,\mathsf{go}\, m.\overline{s}\,]_1^n$: in this case, consider the following transition from T_2 obtained by applying rule L-SPAWN-S:

$$T_2 \xrightarrow{\tau} T_3 = \Delta_1 \rhd [\,\overline{s}\,]_1^m$$

Since there is no silent transition from U_2 (n_1 is not alive in Δ_1), we must have $\langle U_2, T_3 \rangle \in \mathcal{S}$. But this is impossible for U_2 has no barb on $\overline{s}$ and we have the following transition from T_3, obtained by applying rule L-OUT:

$$T_3 \xrightarrow{\overline{s}@m_1} \Delta_1 \rhd [\,\mathbf{0}\,]_1^m$$

(b) $U \xrightarrow{\tau} \Delta \rhd [\,\overline{s}\,]_1^m \xrightarrow{\mathsf{kill}(n,1)} U_2 = \Delta_1 \rhd [\,\overline{s}\,]_1^m$: in this case, we reason as in the case above, with Δ_4 and Δ_5 in place of Δ_2 and Δ_3, to conclude to an impossibility.

All the cases above thus lead to an impossibility, which means no weak simulation $\mathcal{S}$ can exist. □

A consequence of this result is that the spawn primitive and the go primitive are not weakly bisimilar and are not even inter-similar (one being able to simulate the other and vice-versa). This suggests that a go primitive as defined above hides too much atomicity for a distributed setting, where location and link failures can disrupt interactions in various ways, which are more faithfully captured by our spawn primitive. We conclude by remarking that the spawn primitive is always able to weakly simulate the go primitive, indeed a successful go can be matched by a spawn primitive by applying rules SPAWN-C-S and SPAWN-S, while a failed go primitive can be matched by applying rule SPAWN-C-F. ◊

Example 5.4 (Distributed Server)**.** Here, we rephrase [FH08, Example 11], where Francalanza and Hennessy show that the behavioral theory they present is able to distinguish **servFHD**, a distributed server only able to reach its backend n by a direct connection, and **servFHD2Rt**, that, in addition to the direct connection, has also an indirect connection that goes through a third location m.

System **servFHD** in DπFR is defined as

$$\nu\, d, w_y.\Delta \rhd [\, req(x,y).\mathsf{spawn}\ n.\overline{d}\langle x,y\rangle\,]^l \parallel [\, d(x,y).\mathsf{spawn}\ l.\overline{x}\langle w_y\rangle\,]^n$$

while system **servFHD2Rt** is defined as

$$\nu\, d, w_y.\Delta \rhd \left(\begin{array}{l} \left[req(x,y).\nu\, s. \left(\begin{array}{l} \mathsf{spawn}\ n.\overline{d}\langle s,y\rangle \mid \\ \mathsf{spawn}\ m.\mathsf{spawn}\ n.\overline{d}\langle s,y\rangle \mid \\ s(w).\overline{x}\langle w\rangle \end{array} \right) \right]^l \parallel \\ \left[d(s,y). \left(\begin{array}{l} \mathsf{spawn}\ l.\overline{s}\langle w_y\rangle \mid \\ \mathsf{spawn}\ m.\mathsf{spawn}\ l.\overline{s}\langle w_y\rangle \end{array} \right) \right]^n \end{array} \right)$$

where network Δ is defined as

$$\Delta = \langle \{n \mapsto 1, m \mapsto 1, l \mapsto 1\}, \{n \leftrightarrow m, n \leftrightarrow l, m \leftrightarrow l\}, \{n \mapsto \hat{\mathbf{0}}, m \mapsto \hat{\mathbf{0}}, l \mapsto \hat{\mathbf{0}}\}\rangle$$

In both the cases, we use w_y to emphasize that the answer of the server depends on the query y. Notice in **servFHD2Rt** the use of forwarder $s(x).\overline{y}\langle x\rangle$ to ensure that if the requests via both the links succeed, only one answer is made available.

Now, **servFHD** and **servFHD2Rt**, as in [FH08], can be distinguished in DπFR by the following context

$$[\,\mathsf{unlink}\; n.\overline{req}\langle z, h\rangle\,]^l$$

as **servFHD** would stop working after the unlink reduces, while **servFHD2Rt** would keep working correctly since it could route the request through m. $\Diamond$

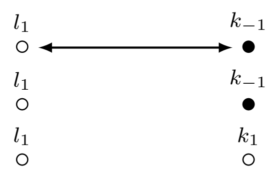
Example 5.5 (Network Observations)**.** Here we rephrase [FH08, Example 12] and show a crucial difference between the observational theory of DπF and that of DπFR which involves three different features, related respectively to: i) the presence of our create primitive; ii) the presence of our link primitive; and iii) their combination.

In [FH08, Example 12], Francalanza and Hennessy show that for any configuration N the three systems, $\nu\, k.\Delta_1 \triangleright N$, $\nu\, k.\Delta_2 \triangleright N$ and $\nu\, k.\Delta_3 \triangleright N$, where Δ_1, Δ_2, and Δ_3 are as given below, are equivalent according to their behavioral theory.

$$\Delta_1 = \langle \{l \mapsto 1, k \mapsto -1\}, \{l \leftrightarrow k\}, \{l \mapsto \hat{\mathbf{0}}\}\rangle =$$

$$\Delta_2 = \langle \{l \mapsto 1, k \mapsto -1\}, \emptyset, \{l \mapsto \hat{\mathbf{0}}\}\rangle =$$

$$\Delta_3 = \langle \{l \mapsto 1, k \mapsto 1\}, \emptyset, \{l \mapsto \hat{\mathbf{0}}, k \mapsto \hat{\mathbf{0}}\}\rangle =$$



The impossibility of distinguishing the three systems in [FH08] is due to the absence of recovery of both links and locations. Even if N is a configuration extruding the private name (e.g., pick N to be $[\,\overline{a}\langle k\rangle\,]^l_1$) and put in a context where another process can receive the name k, then this process cannot establish a connection or reactivate k and hence cannot observe any difference.

In more detail, in the first system, even if an observer can establish a link to k the location is dead and cannot be revived. In the second system k is dead and unreachable by any hypotethical observer. Indeed, the only way to establish new links in DπF is to create a fresh location, but such fresh location will only be connected to the set of locations reachable by the parent location. Since here l is not connected to k, l cannot interact with k, and in particular it is impossible to establish a link with it. Finally, in the third system, even if k is alive, it is still not reachable due to the lack of a connection as above and hence there is no way to discover that k is alive.

In DπFR, since we allow for creation of links without requiring prior connections and we allow for recovery of locations, if N extrudes k then we can easily distinguish the three networks. We do it by pairing them with a configuration that receives the extruded name k and then exploits it to assess the status of the link, trying to link or to unlink the location, and of the location, trying to restart or to kill it.

Thanks to the full-abstraction result, this can be seen directly on our LTS with the following labeled derivations:

$$\nu\, k.\Delta_1 \rhd [\,\overline{a}\langle k\rangle\,]^l_1 \xrightarrow{\nu\, k.\overline{a}\langle k\rangle @ l_1} \Delta_1 \rhd \mathbf{0} \xrightarrow{\ominus l \leftrightarrow k}$$
$$\nu\, k.\Delta_2 \rhd [\,\overline{a}\langle k\rangle\,]^l_1 \xrightarrow{\nu\, k.\overline{a}\langle k\rangle @ l_1} \Delta_2 \rhd \mathbf{0} \xrightarrow{\oplus l \leftrightarrow k} \Delta_2' \rhd \mathbf{0} \xrightarrow{\mathsf{create}(n,2)}$$
$$\nu\, k.\Delta_3 \rhd [\,\overline{a}\langle k\rangle\,]^l_1 \xrightarrow{\nu\, k.\overline{a}\langle k\rangle @ l_1} \Delta_3 \rhd \mathbf{0} \xrightarrow{\mathsf{kill}\ k}$$

Notice that each derivation cannot be matched by the other systems. ◊

6. Capturing Distributed Systems Characteristics

Here we discuss how our calculus captures various features which we believe typical of distributed systems.

Communication. First of all, communication between remote parties must be asynchronous. That is, interaction between remote processes proceeds by a non-atomic exchange of messages between the nodes that support them; messages can possibly be lost (because of link failure) or reordered while transiting to their destination. Moreover, each communication must target a single location, either local or remote. This ensures that the complexity of basic 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. [FG96] for a discussion).

In our calculus, interactions are local. Remote communication is obtained using the spawn operation to send to a target node a process performing an output action. As a result remote communication is indeed asynchronous, and there is a single location where the receiver can reside (since such location is specified in the spawn). This avoids the need of a type system to ensure that possible receivers are located on a same node, in contrast to calculi based on channel-based remote communication such as [Ama97].

Dynamic nodes and links. During execution, new nodes and links can be established and 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 and performance management (e.g. scaling decisions in cloud systems). The explicit presence of links is important because partial connections often affect large distributed systems. Dealing with link failures in addition of node failures can lead to a subtly different behavioral theory than dealing with node failures only, as shown in [FH08].

Example 6.1. The following configuration, which can be added as a context to any system where m is a free name, is an example of how a system can be extended with an unbounded number of fresh locations

$$[\,\nu\, c.(\overline{c} \mid !c.(\nu\, n.\mathsf{create}\ n.\mathbf{0} \mid \overline{c}))\,]^m$$

Indeed, the synchronization between the output $\overline{c}$ and the replicated input will create a new location with a fresh name n, and recreate the output for a further iteration, resulting in an infinite computation.

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 spawning message from a remote location l then n learns something on the context, namely that at some point in time l was alive and working since it sent a spawning message to n. Nonetheless, n cannot infer anything on the current status of l or the status of the connection. Indeed, location l could have stopped right after sending the spawning message or the link could have broken right after the spawning 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 spawning message exchange. Reflecting this in our calculus plays a role in the semantics of our spawn primitive whenever the view of the location 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.

Example 6.2. Consider a variant **servDFV** of **servDFR** where n_c is linked to n_i instead of n_r and 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:

$$C' = \nu\, x.\mathsf{create}\; n_r.(R \mid \mathsf{spawn}\; n_c.\overline{x}) \mid x.\mathsf{spawn}\; n_i.\overline{retry}$$

instead of

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

as in **servDFR**.

Now, **servDFV**, differently from **servDFR**, is not equivalent to **servD** because, in case of failure of n_r, the controller triggers the interface to restart the request instead of the router n_r, thus failing to update n_i's local view with the knowledge of n_r's new incarnation. A spawning message from the interface at this point would fail as its local view contains the previous incarnation of n_r. If not for the wrong belief of n_i, **servDFV** would have been equivalent to **servD**. In the companion repository [Erl23] we discuss an implementation of this system and we show that this behavior arises in reality too.

Networks in Equivalent Systems. We remarked in the Introduction that one ought to be able to prove from the behavioral theory that two weakly barbed congruent systems have networks with the same public part (i.e. those nodes and links whose names are not restricted). Let us first consider the case of strongly bisimilar systems: they do indeed have identical public network parts. This follows from the rules in Fig.7 and the rule L-RES as shown in Proposition 4.13. In particular, incarnation numbers of public nodes in barbed congruent systems must coincide. This may seem to be overly discriminating but in fact it is warranted: for instance, recovery protocols or failure handling protocols such as SWIM [DGM02], which rely on incarnation numbers, would operate differently in dissimilar systems. Another consequence, visible from rule L-VIEW, is that two weak bisimilar systems must weakly agree on their public correct local views, meaning that, in the local view of a given public location n, they either have the same correct beliefs on the status of a location, or they hold no belief on the status of a location, or one has a correct view on the status of a location and the other holds no belief on this location. In other terms, two equivalent systems cannot disagree in their local views, with one having a correct belief and the other one having an incorrect one.

Let us now consider weakly barbed congruent systems. A consequence of our full abstraction result is that the public parts of networks of weakly barbed congruent systems agree only up to reductions, namely there exist sequences of reductions leading to systems with the same public parts of networks. This is weaker than the requirement imposed by the decision in [FH08] to consider public networks as types, and requiring equivalent systems to have the same type. Indeed the latter approach, because of types, differentiates between systems which would otherwise be weakly barbed congruent.

Persistence. Various forms of recovery, such as checkpoint-rollback schemes, require some persistent storage to store information that will be preserved even if failures occur. DπFR has by construction a location $\odot$ which, since it cannot be killed, models a persistent location. However the only purpose of $\odot$ is to offer a way to extend a system with a running process, without having to worry about finding an alive location. Nonetheless, persistent memories can be encoded in DπFR by leveraging the restriction operator.

Example 6.3. As an example, consider the following piece of code, which implements a simple read and write persistent server

$$\nu\, l, m.\Delta \triangleright [\,!I\,]_1^l \parallel [\,M\,]_1^m$$

where

$$\Delta = \langle \{l \mapsto 1, m \mapsto 1\}, \{l \leftrightarrow m, n \leftrightarrow m\}, \{l \mapsto \hat{\mathbf{0}}, m \mapsto \hat{\mathbf{0}}\}\rangle \quad I \stackrel{\text{def}}{=} \mathsf{create}\; n.addr(x).(!W \mid !R)$$

$$M \stackrel{\text{def}}{=} (!\mathsf{spawn}\; n.\overline{addr}\langle m\rangle) \mid \overline{data}\langle\widetilde{u_0}\rangle \qquad W \stackrel{\text{def}}{=} write(\tilde{u}).\mathsf{spawn}\; x.data(\tilde{v}).\overline{data}\langle\tilde{u}\rangle$$

$$R \stackrel{\text{def}}{=} read(l, x).\mathsf{spawn}\; x.data(\tilde{u}).(\overline{data}\langle\tilde{u}\rangle \mid \mathsf{spawn}\; n.\mathsf{spawn}\; l.\overline{x}\langle\tilde{u}\rangle)$$

Process I, for *init*, creates a server running on location n. The server first awaits on channel *addr* to know the address of its memory. Once acquired, it offers to its clients the possibility to write (process $!W$) or to read (process $!R$) data. The data stored by the server are persistent, indeed, even if the server n fails it will be re-instantiated by $[\,!I\,]_1^l$, and if a read request is forwarded the data will be retrieved from the memory, i.e., location m. Location m, being a private location, cannot be killed or unlinked ever by any context, hence we can leverage it to encode a persistent memory. The behavior of M itself is simple: it repeatedly sends the name of the private memory m to its server n, and holds information at the channel *data*, which can be retrieved or updated by n via the *read* or *write* operations.

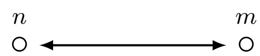
7. Erlang Experiments

In the Introduction we claimed that DπFR faithfully reflects the behavior of Erlang systems in case of failure. Erlang comes with a documentation [Erla] and several other resources that do their best to explain the principles behind the language and the semantics of each primitive. Nonetheless, there are plenty of corner cases that are not discussed and whose behavior is not explicitly documented. For instance the documentation for the spawn targeting a remote node does not mention that in case of failure the view of the local node is updated by removing any knowledge of the remote node. Nor is it mentioned that if the local node holds a wrong belief on the remote one then the spawn is doomed to fail, even if both nodes are alive and connected.

To shed light on these obscure corner cases and clarify the behavior of Erlang systems in presence of failures, we carried out some experiments on a (simulated) distributed Erlang environment. To simulate different machines connected by a network we leveraged Docker [Doc]. Docker is a tool that allows one to test and deploy applications relying on the concept of containerization. A container is a bundle packing together everything that is needed to run an application (code, libraries, system tools, etc.) which is then run isolated from other containers, sharing the hosting system's kernel. Containers can communicate with each other through a network, and Docker provides facilities to manage it. For the sake of our investigation, we used containers to run Erlang nodes and then observe their behavior in corner cases.

A description of all the experiments we did, including used code and scripts, is available in our companion repository [Erl23]. We describe below one of these examples, to give to the reader an intuition about how such experiments can be done, while referring to [Erl23] for a description of the other experiments, including Erlang implementations of our examples and case studies.

We describe an experiment showing how views can impact a system behavior. The scenario we consider is the following: two locations n and m, running on two separate containers, connected by a network link $n \leftrightsquigarrow m$. Graphically this system could be represented as:



We then consider the following sequence of events:

(1) location m spawns a process on location n
(2) location m is unlinked from location n
(3) location m dies
(4) location m is recreated
(5) location m is linked again to location n
(6) location n spawns a process on location m

A system in DπFR with such a configuration, whose behavior includes a sequence of events as above can be defined as follows:

$$\mathbf{WrongSp} \stackrel{\text{def}}{=} \Delta \triangleright [\,\mathsf{spawn}\, n.P\,]^m_1 \parallel [\,\mathsf{spawn}\, m.Q\,]^n_1 \parallel [\,\mathsf{unlink}\, n\,]^m_1 \parallel [\,\mathsf{kill}\,]^m_1 \parallel [\,\mathsf{create}\, m.\mathsf{link}\, n\,]^{\odot}$$

with

$$\Delta \equiv \langle \{n \mapsto 1, m \mapsto 1\}, \{n \leftrightarrow m\}, \hat{\mathbf{0}} \rangle$$

At first glance, the outcome of each of these operations, of the two spawns in particular, seems to be pretty straightforward to determine. Both spawns are carried out with both locations alive and connected, hence one would expect both of them to succeed. However, there are circumstances under which the last spawn may fail. In particular, this occurs when the sequence of events is carried out rapidly and location n does not have enough time to detect that the previous incarnation of m – the one it interacted with during the first spawn – is dead. As a result, it will attempt to spawn the process on the old incarnation instead of the new one.

We now show how our experiment can be performed with Erlang. The first step consists of setting up the system. To do so, we use a “docker-compose” file, which is a YAML file describing the system configuration. The configuration file we used can be found at [Erl23], under the `wrong-spawn` folder. The following command sets up the system.

```
$ docker-compose up -d
```

The following commands attach a remote console to each of location n and location m, so as to be able to perform operations from them.

```
$ docker exec -it loc_n.com erl -name test@loc_n.com \
  -setcookie cookie -remsh app@loc_n.com -hidden
```

```
$ docker exec -it loc_m.com erl -name test@loc_m.com \
  -setcookie cookie -remsh app@loc_m.com -hidden
```

Event (1) is executed from the remote console of location m as follows

```
(app@loc_m.com)2> erlang:spawn('app@loc_n.com', erlang, self, []).
```

Events (2,3,4,5) are executed by the following commands, where `$1` is replaced with the network name and `$2` is replaced with the container name.

```
docker network disconnect $1 $2
docker container stop $2
docker container start $2
docker network connect $1 $2
```

However, rather than executing them as single commands we conveniently bundle them in the script `restart.sh`, available in the repository. Executing them with a script ensures little delay between each command's execution, leaving us enough time to attempt another spawn from location n towards location m before location n's reaction to the absence of location m's heartbeat. The script is then executed as follows.

```
$ ./restart.sh wrong-spawn_net1 loc_m.com
```

Finally, event (6) is executed as follows.

```
(app@loc_n.com)2> erlang:spawn('app@loc_m.com', erlang, self, []).
<0.108.0>
(app@loc_n.com)3> =WARNING REPORT==== 12-Sep-2024::13:22:40.233966 ===
** Can not start erlang:self,[] on 'app@loc_m.com' **
```

As we can see, the spawn from location n towards location m fails because the spawn message sent by location n is targeting an old instance of location m.

The above experiment mixes the use of Erlang primitives to perform spawns with the use of Docker primitives to alter the network state, as one can do in DπFR. We now show how this behavior is faithfully captured in DπFR, by showing a possible reduction sequence for system **WrongSp**:

$$
\begin{array}{rl}
& \Delta \triangleright \underline{[\,\mathsf{spawn}\ n.P\,]_1^m} \parallel [\,\mathsf{spawn}\ m.Q\,]_1^n \parallel [\,\mathsf{unlink}\ n\,]_1^m \parallel [\,\mathsf{kill}\,]_1^m \parallel [\,\mathsf{create}\ m.\mathsf{link}\ n\,]^{\odot} \\
\longrightarrow & \Delta \triangleright \underline{\langle (n,1) : P \rangle_1^m} \parallel [\,\mathsf{spawn}\ m.Q\,]_1^n \parallel [\,\mathsf{unlink}\ n\,]_1^m \parallel [\,\mathsf{kill}\,]_1^m \parallel [\,\mathsf{create}\ m.\mathsf{link}\ n\,]^{\odot} \\
\longrightarrow & \Delta_1 \triangleright [\,P\,]_1^n \parallel [\,\mathsf{spawn}\ m.Q\,]_1^n \parallel \underline{[\,\mathsf{unlink}\ n\,]_1^m} \parallel [\,\mathsf{kill}\,]_1^m \parallel [\,\mathsf{create}\ m.\mathsf{link}\ n\,]^{\odot} \\
\longrightarrow & \Delta_2 \triangleright [\,P\,]_1^n \parallel [\,\mathsf{spawn}\ m.Q\,]_1^n \parallel \underline{[\,\mathsf{kill}\,]_1^m} \parallel [\,\mathsf{create}\ m.\mathsf{link}\ n\,]^{\odot} \\
\longrightarrow & \Delta_3 \triangleright [\,P\,]_1^n \parallel [\,\mathsf{spawn}\ m.Q\,]_1^n \parallel \underline{[\,\mathsf{create}\ m.\mathsf{link}\ n\,]^{\odot}} \\
\longrightarrow & \Delta_4 \triangleright [\,P\,]_1^n \parallel [\,\mathsf{spawn}\ m.Q\,]_1^n \parallel \underline{[\,\mathsf{link}\ n\,]_2^m} \\
\longrightarrow & \Delta_5 \triangleright [\,P\,]_1^n \parallel \underline{[\,\mathsf{spawn}\ m.Q\,]_1^n} \\
\longrightarrow & \Delta_5 \triangleright [\,P\,]_1^n \parallel \underline{\langle (m,1) : Q \rangle_1^n} \\
\longrightarrow & \Delta_6 \triangleright [\,P\,]_1^n
\end{array}
$$

with

$$
\begin{array}{rcl}
\Delta & \equiv & \langle \{n \mapsto 1, m \mapsto 1\}, \{n \leftrightarrow m\}, \hat{\mathbf{0}} \rangle \\
\Delta_1 & \equiv & \langle \{n \mapsto 1, m \mapsto 1\}, \{n \leftrightarrow m\}, \{n \mapsto \{m \mapsto 1\}\} \rangle \\
\Delta_2 & \equiv & \langle \{n \mapsto 1, m \mapsto 1\}, \emptyset, \{n \mapsto \{m \mapsto 1\}\} \rangle \\
\Delta_3 & \equiv & \langle \{n \mapsto 1, m \mapsto -1\}, \emptyset, \{n \mapsto \{m \mapsto 1\}\} \rangle \\
\Delta_4 & \equiv & \langle \{n \mapsto 1, m \mapsto 2\}, \emptyset, \{n \mapsto \{m \mapsto 1\}\} \rangle \\
\Delta_5 & \equiv & \langle \{n \mapsto 1, m \mapsto 2\}, \{n \leftrightarrow m\}, \{n \mapsto \{m \mapsto 1\}\} \rangle \\
\Delta_6 & \equiv & \langle \{n \mapsto 1, m \mapsto 2\}, \{n \leftrightarrow m\}, \hat{\mathbf{0}} \rangle
\end{array}
$$

In the above trace, the error takes place in the last reduction when rule SPAWN-F is triggered, due to the spawn message targeting the wrong incarnation number of location m, which in turn is due to the wrong view location n yields about location m in Δ_5.

8. Related Work and Conclusion

We have presented in this paper DπFR, a distributed π-calculus with dynamic locations and links, crash failures and a weak recovery model, supporting location recovery by means of incarnation numbers. To the best of our knowledge, this is the first work that combines these different features, and the first to adopt a weak recovery model. We have conducted experiments which show our model can indeed represent faithfully the behavior of Erlang systems in presence of node failures and recoveries. We have developed a behavioral theory for DπFR, including a full abstraction result that characterizes weak barbed congruence for the calculus by a weak bimilarity. We have shown that our behavioral theory extends the one from the DπF paper [FH08], but that weak barbed congruence in DπFR is in general more discriminative than in DπF because of the recovery-specific features.

We highlighted in the Introduction how the previous process calculus analyses of distributed systems with crash failures and recovery broadly compared with respect to these key features. We provide below some additional discussion and consider other related works.

The work by Fournet et al. on the join calculus [FGL+96] 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 [Ama97] 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 (i.e., a failure detector that only detects faulty processes, with no false positives [GCG01]), and does not support links and link failures. 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. In [FG96, Ama97] failure/recovery is akin to an off/on switch: recovering a failed process is just a matter of restarting a process whose execution has been suspended by a failure. This is arguably extremely simplistic, and the underlying ability to recover a failed process in the exact state prior to failure is at best non-trivial or costly to implement in practice.

The recovery model in [BH00] and [BLTV23] improves on the on/off switch model from [FG96, Ama97]. To model recovery they rely on timed systems and a checkpointing primitive (their save primitives) that captures the execution state of a process and saves it in permanent storage for restart after failure. This is more realistic than the on/off model, however the checkpointing primitive is still an expensive construct and not necessarily found in practice. For instance, the Erlang environment does not include a checkpointing capability in its basic features. The approach in [BH00] considers only a fixed finite number of nodes. They develop a behavioral theory in the form of a sound higher-order bisimilarity. In [BLTV23] systems comprise a fixed finite number of nodes with links connecting each pair of nodes, together with a function giving the failure status of nodes and links at any given time. Their failure model is more sophisticated than just node and link crash failures, as it also allows to model link slowdowns, delaying the transmission of messages beyond the expected link latency. Their behavioral theory is limited to a notion of weak barbed bisimilarity (symmetric weak-barb preserving and reduction closed relation) and they do not present a labelled transition semantics for their model. In contrast, we are only concerned with crash failures and eschew the need for a timed model. Dealing with dynamic node and link creation is a prime objective of our work, which cannot be simply modelled by activating a location at a given time, and which has a strong bearing on our behavioral theory and our full-abstraction result.

A clear inspiration for this paper was the work by Francalanza and Hennessy [FH08] 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, as we described in the Introduction. Compared to theirs, our approach is untyped, with a simpler handling of scope extrusion, simpler labels in our LTS semantics, and no need to make explicit the partial view of a network available to an observer to obtain our full abstraction result. This has several effects on our behavioral theory. Because our networks are not types, more systems can be seen to be barbed congruent for they only need to weakly agree on the public part of their networks, and do not need to have them identical. At the same time, our behavioral theory is more discriminative due to our ability to establish links without the connectedness constraints in [FH08]. Also, we introduce explicit local views, which correspond to the belief that a location has of its neighbours and their current incarnation, which also have a bearing on our behavioral theory. 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 [FH08]. The notion of partial views for observers in [FH08] is different: it is a way to filter out information contained in their complex labels so as to obtain full abstraction. One should note also that [FH08] is concerned with network partitions, with network extensions in their DπF calculus designed to be partition-preserving. This is not a concern in DπFR. Our primitives for location and link creation and recovery can heal network partitions.

Another work dealing with explicit links between computational nodes is [NGP07], which presents a tuple-space based process calculus with dynamic locations and links. The paper does not consider location failure and recovery but provides a behavioral theory with a full abstraction result. Links in this work appear as terms of the process calculus, handling new link creation and name restriction in a manner similar to ours. In contrast, we do not consider links as explicit process terms but gather all the link information, together with location incarnation and status, in our networks. We found this lead to simpler reduction and LTS rules, and avoids the need to have a partial parallel composition operating only on compatible systems (if network information is distributed in terms in parallel with locations as in [NGP07], the definition of parallel composition must take into account the fact that the systems being composed may hold conflicting network information such as differing node and link status).

Formal models for distributed systems with failures can also be found in recent verification tools for distributed algorithms and distributed systems such as Disel [SWT18], Gobra [WAC$^+$21], Perennial [CTKZ19], Psync [DHZ16], TLC [GLSY20], Verdi [WWP$^+$15]. 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.

A formal analysis of fault-tolerant behaviors by means of simulation relations is provided in [DCMA17]. Their analysis aims to find algorithms on finite Kripke structures for verifying simulation relations capturing different notions of fault-tolerance (masking, non-masking, failsafe) taken from [Gär99]. They characterize faults by an explicit colouring (good or bad) of states in a Kripke structure. In contrast, DπFR systems can have infinite states, and a fault, error or failure would correspond to a deviation from a specified behavior, with location and link crashes as primitive faults in a system. Nevertheless, it would be interesting to develop in our setting similar characterizations of fault-tolerance, as well as to study notions of recoverability inspired by those in [BLTV23].

The calculus presented in this paper provides us with a basis for further studies. One of the consequences of the closeness to Erlang we have adopted in this work is the fact that the way we deal with local views corresponds to the Erlang policy. It would be interesting to study a variant of DπFR where programs can specify their own policies for managing local views. This can be achieved by adding further primitives to complement the forget primitive to explicitly manipulate local view information. This extension would allow one to encode different failure detector schemes, as well as distributed protocols relying on an explicit management of local views, such as epidemic protocols. It would certainly be interesting to further expand DπFR to cater for other failure models, including the kinds of grey failures tackled by Bocchi et al. [BLTV23], or Byzantine failures, probably making it parametric in failure models along the lines of Verdi [WWP$^+$15]. It would also be interesting to revisit ideas from [FH07] about the semantical characterization of, and proof techniques for, fault tolerance. Of particular interest would be the ability in our setting to drastically reduce the size of bisimulations in presence of unbounded occurrences of failures and recoveries as in failure and recovery loops.

References

[Ama97] 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. doi:10.1007/3-540-63383-9_92.

[Arm07] Joe Armstrong. *Programming Erlang – Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[BH00] 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. doi:10.1016/S1571-0661(05)82502-2.

[BLTV23] Laura Bocchi, Julien Lange, Simon Thompson, and A. Laura Voinea. A model of actors and grey failures. *Log. Methods Comput. Sci.*, 19(4), 2023. URL: https://doi.org/10.46298/lmcs-19(4:13)2023, doi:10.46298/LMCS-19(4:13)2023.

[Ces19] Francesco Cesarini. Which companies are using Erlang, and why? URL:https://erlang-solutions.com/blog/which-companies-are-using-erlang-and-why, 2019.

[CTKZ19] 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. doi:10.1145/3341301.3359632.

[DCMA17] Ramiro Demasi, Pablo F. Castro, Thomas Stephen Edward Maibaum, and Nazareno Aguirre. Simulation relations for fault-tolerance. *Formal Aspects Comput.*, 29(6):1013–1050, 2017. doi:10.1007/s00165-017-0426-2.

[DGM02] 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. doi:10.1109/DSN.2002.1028914.

[DHZ16] 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. doi:10.1145/2837614.2837650.

[Doc] Docker. URL:https://www.docker.com/.

[DTG99] 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. doi:10.1109/RELDIS.1999.805099.

[EAWJ02] 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. doi:10.1145/568522.568525.

[Erla] Erlang documentation. URL:https://www.erlang.org/docs.

[Erlb] Erlang documentation: creation numbers. URL: https://www.erlang.org/doc/apps/erts/erl_ext_dist.html#NEW_PID_EXT.

[Erl23] Erlang experiments. https://zenodo.org/doi/10.5281/zenodo.10895171, 2023.

[EZ92] 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. doi:10.1109/12.142678.

[FG96] 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. doi:10.1145/237721.237805.

[FGL+96] 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. doi:10.1007/3-540-61604-7_67.

[FH07] Adrian Francalanza and Matthew Hennessy. A theory for observational fault tolerance. *J. Log. Algebraic Methods Program.*, 73(1-2):22–50, 2007. URL: https://doi.org/10.1016/j.jlap.2007.03.003, doi:10.1016/J.JLAP.2007.03.003.

[FH08] 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. doi:10.1016/j.ic.2007.12.002.

[Gär99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999. doi:10.1145/311531.311532.

[GCG01] 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. doi:10.1145/383962.384010.

[GDQ+18] 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. doi:10.1145/3236024.3236030.

[GLSY20] 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. doi:10.1145/3409005.

[Kub23] Kubernetes overview. URL: https://kubernetes.io/docs/concepts/overview/, 2023.

[MPSS17] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering shared objects without stable storage. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 36:1–36:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.36.

[MS92] 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. doi:10.1007/3-540-55719-9_114.

[NGP07] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Basic observables for a calculus for global computing. *Inf. Comput.*, 205(10):1491–1525, 2007. doi:10.1016/j.ic.2007.03.004.

[Pax] Paxos protocols. URL: https://en.wikipedia.org/wiki/Paxos_(computer_science).

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[SWT18] 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. doi:10.1145/3158116.

[SY85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985. doi:10.1145/3959.3962.

[WAC+21] 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. doi:10.1007/978-3-030-81685-8_17.

[WWP+15] 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. doi:10.1145/2737924.2737958.

Appendix A. Free Names and Alpha-Conversion on Systems

The definition of free incarnation variables in configurations and systems is completely standard.

Free incarnation variables in processes are defined inductively as follows:

$$\begin{array}{ll}
\mathsf{fv}(\mathbf{0}) = \emptyset & \mathsf{fv}(\overline{x}\langle\tilde{u}\rangle.P) = \mathsf{fv}(P) \cup (\tilde{u} \cap \mathsf{I}) \\
\mathsf{fv}(x(\tilde{v}).P) = \mathsf{fv}(P) \setminus \tilde{v} & \mathsf{fv}(!x(\tilde{v}).P) = \mathsf{fv}(P) \setminus \tilde{v} \\
\mathsf{fv}(\nu\, w.P) = \mathsf{fv}(P) \setminus \{w\} & \mathsf{fv}(\mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q) = \mathsf{fv}(P) \cup \mathsf{fv}(Q) \cup (\{r, s\} \cap \mathsf{I}) \\
\mathsf{fv}(P \mid Q) = \mathsf{fv}(P) \cup \mathsf{fv}(Q) & \mathsf{fv}(\mathsf{node}(n, \iota).P) = \mathsf{fv}(P) \setminus \{\iota\} \\
\mathsf{fv}(\mathsf{forget}\, n.P) = \mathsf{fv}(P) & \mathsf{fv}(\mathsf{spawn}\, n.P) = \mathsf{fv}(P) \\
\mathsf{fv}(\mathsf{kill}) = \emptyset & \mathsf{fv}(\mathsf{create}\, n.P) = \mathsf{fv}(P) \\
\mathsf{fv}(\mathsf{link}\, n) = \emptyset & \mathsf{fv}(\mathsf{unlink}\, n) = \emptyset
\end{array}$$

Free incarnation variables in configurations are defined inductively as follows:

$$\begin{array}{ll}
\mathsf{fv}([\, P\,]^n_\lambda) = \mathsf{fv}(P) & \mathsf{fv}(\langle (m, \kappa) : P\rangle^n_\lambda) = \mathsf{fv}(P) \\
\mathsf{fv}(N \parallel M) = \mathsf{fv}(N) \cup \mathsf{fv}(M) & \mathsf{fv}(\mathbf{0}) = \emptyset
\end{array}$$

Free incarnation variables in networks and systems are defined inductively as follows:

$$\mathsf{fv}(\Delta \triangleright N) = \mathsf{fv}(N)$$
$$\mathsf{fv}(\nu\, \tilde{u}.S) = \mathsf{fv}(S) \setminus \tilde{u}$$

The notion of free names in configurations is completely standard too, while in systems is slightly unconventional because of the presence of a network, but it can be defined straightforwardly as follows.

Free names in processes are defined inductively as follows:

$$\begin{array}{ll}
\mathsf{fn}(\mathbf{0}) = \emptyset & \mathsf{fn}(\overline{x}\langle\tilde{u}\rangle.P) = (\tilde{u} \cap (\mathsf{N} \cup \mathsf{C})) \cup \{x\} \cup \mathsf{fn}(P) \\
\mathsf{fn}(x(\tilde{v}).P) = \{x\} \cup (\mathsf{fn}(P) \setminus \tilde{v}) & \mathsf{fn}(!x(\tilde{v}).P) = \{x\} \cup (\mathsf{fn}(P) \setminus \tilde{v}) \\
\mathsf{fn}(\nu\, w.P) = \mathsf{fn}(P) \setminus \{w\} & \mathsf{fn}(\mathsf{if}\ r = s\ \mathsf{then}\ P\ \mathsf{else}\ Q) = \begin{array}{l}\mathsf{fn}(P) \cup \mathsf{fn}(Q) \\ \quad \cup (\{r, s\} \cap (\mathsf{N} \cup \mathsf{C}))\end{array} \\
\mathsf{fn}(P \mid Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q) & \mathsf{fn}(\mathsf{node}(n, \iota).P) = \mathsf{fn}(P) \setminus \{n\} \\
\mathsf{fn}(\mathsf{forget}\, n.P) = \mathsf{fn}(P) \cup \{n\} & \mathsf{fn}(\mathsf{spawn}\, n.P) = \mathsf{fn}(P) \cup \{n\} \\
\mathsf{fn}(\mathsf{kill}) = \emptyset & \mathsf{fn}(\mathsf{create}\, n.P) = \mathsf{fn}(P) \cup \{n\} \\
\mathsf{fn}(\mathsf{link}\, n) = \{n\} & \mathsf{fn}(\mathsf{unlink}\, n) = \{n\}
\end{array}$$

Free names in configurations are defined inductively as follows:

$$\begin{array}{ll}
\mathsf{fn}([\, P\,]^n_\lambda) = \mathsf{fn}(P) \cup \{n\} & \mathsf{fn}(\langle (m, \kappa) : P\rangle^n_\lambda) = \mathsf{fn}(P) \cup \{m, n\} \\
\mathsf{fn}(N \parallel M) = \mathsf{fn}(N) \cup \mathsf{fn}(M) & \mathsf{fn}(\mathbf{0}) = \emptyset
\end{array}$$

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:

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

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

$$\mathcal{V}\{v/u\} = \{\mathcal{V}(n)\{v/u\} | n \in \mathsf{dom}(\mathcal{V}) \cup \{v\}\} \quad \text{if} \quad u, v \in \mathsf{N} \text{ and } v \notin \mathsf{supp}(\mathcal{A})$$

with:

$$\mathcal{L}(n) = \{(a, b) \in \mathcal{L} \mid a = n \text{ or } b = n\}$$

$$\mathcal{L}(n)\{m/n\} = \{(a\{m/n\}, b\{m/n\}) \mid (a, b) \in \mathcal{L}(n)\}$$

$$a\{m/n\} = \begin{cases} m & \text{if} \quad a = n \\ a & \text{otherwise} \end{cases}$$

$$\mathcal{V}(n)\{v/u\} = \begin{cases} \mathcal{V}(u) & \text{if} \quad n = v \\ \hat{\mathbf{0}} & \text{if} \quad n = u \\ \{n\{v/u\} \mapsto \lambda | n \mapsto \lambda \in \mathcal{V}(n)\} & \text{otherwise} \end{cases}$$

The effect of a capture avoiding substitution $\{v/u\}$ on a system $\Delta \triangleright N$ is defined inductively as follows.

$$(\Delta \triangleright N)\{v/u\} = \Delta\{v/u\} \triangleright N\{v/u\}$$

$$(\nu\, w.S)\{v/u\} = \begin{cases} \nu\, w.S\{v/u\} & \text{if } u, v \neq w \\ \nu\, w.S & \text{if } u = w \\ \nu\, w'.S\{w'/w\}\{v/u\} & \text{if } v = w \text{ with } w' \text{ fresh} \end{cases}$$

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

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

Appendix B. Modifying Network: Proof

Proposition B.1. *Let $S = \nu\,\tilde{u}.\Delta \triangleright N$ be a closed system, and $\Delta = \langle \mathcal{A}, \mathcal{L}, \mathcal{V} \rangle$, $\Delta' = \langle \mathcal{A}', \mathcal{L}', \mathcal{V}' \rangle$ be networks such that the following conditions hold:*

(1) $\forall n \in \tilde{u}.\ \mathcal{A}(n) = \mathcal{A}'(n)$
(2) $\forall n \in \mathsf{supp}(\mathcal{A}').\mathcal{A}(n) = \mathcal{A}'(n) \vee |\mathcal{A}(n)| < |\mathcal{A}'(n)| \vee (\mathcal{A}'(n) = -\mathcal{A}(n) \wedge \mathcal{A}(n) \geq 0)$
(3) $\forall n \leftrightarrow m \in (\mathcal{L}' \setminus \mathcal{L}) \cup (\mathcal{L} \setminus \mathcal{L}').\ n, m \notin \tilde{u} \wedge \exists l \in \{n, m\}.(\mathcal{A}(l) > 0 \vee |\mathcal{A}(l)| < |\mathcal{A}'(l)|)$
(4) $\forall n \in \mathcal{N}, m \in \tilde{u}.\ \mathcal{V}(m) = \mathcal{V}'(m), \mathcal{V}(n)(m) = \mathcal{V}'(n)(m)$
(5) $\forall n, m \in \mathcal{N}, \mathcal{V}(n)(m) \neq \mathcal{V}'(n)(m) = \lambda \wedge \lambda \neq 0.(\forall l \in \{n, m\}.\mathcal{A}(l) > 0 \vee |\mathcal{A}(l)| < |\mathcal{A}'(l)|) \vee \lambda \leq |\mathcal{A}'(m)|$
(6) $\forall n, m \in \mathcal{N}, \mathcal{V}'(n)(m) \neq \mathcal{V}'(n)(m) = 0.\ \mathcal{A}(n) > 0 \vee |\mathcal{A}(n)| < |\mathcal{A}'(n)|$

Then, there exists a configuration L with $fn(L) \cap \tilde{u} = \emptyset$ such that: $\nu\,\tilde{u}.\Delta \triangleright N \parallel L \Longrightarrow \nu\,\tilde{u}.\Delta' \triangleright N$.

Before presenting the proof we first comment on the required conditions and then provide some informal intuition on the nature of the proof.

Condition (1) requires $\mathcal{A}$ and $\mathcal{A}'$ to agree on private names and condition (2) requires for the new incarnation number of the target location to be equal or in the future of the previous one. Condition (3) requires $\mathcal{L}$ and $\mathcal{L}'$ to agree on links with at least a private end and for at least one of the two locations to be either already alive or to be alive during the transformation from $\mathcal{A}$ to $\mathcal{A}'$. Condition (4) requires $\mathcal{V}$ and $\mathcal{V}'$ to agree on private names, while conditions (5) and (6) require that in order for a view to change either both locations are alive at some point during the transformation (condition (5)) or that the location bearing the view is alive and it forgets about the other location (condition (6)).

We now give an intuition of the proof.

To appropriately update the network we build a context performing the desired changes, which modifies its three elements: i) the alive set; ii) the link set; and iii) the view. The context to modify each element originates from the special location $\odot$, since we know that this location is always alive.

Let us begin with the alive set. To update the incarnation number of a given location we kill it, by spawning on it a kill process, and recreate it the correct number of times from $\odot$. For this operation to be carried out successfully, it is necessary for the special location $\odot$ and the target one to be connected and in general this may or may not be the case. To obviate this problem, together with the appropriate number of killing and creation processes, we also spawn the following two processes

$$[\,\mathsf{link}\, n\,]^{\odot} \parallel [\,\mathsf{unlink}\, n\,]^{\odot}$$

where n is the target location. These two processes can either set up a connection, for the time it is needed to update the desired incarnation number of n, and then destroy it, in case this was not present, or destroy the connection and then restore it immediately after, bearing no changes on the system. In other words, these two processes alter temporarily the state of the connection, making both the presence and the absence of a link available (at different times).

Now, let us discuss the changes to the link set. The changes can be of two kinds, either a link between two locations is set up or it is destroyed. We discuss the first scenario, the second one is identical. Thanks to the condition on $\mathcal{L}'$ we know that the link in $\mathcal{L}$ does not exist and that at some point in the transformation from $\mathcal{A}$ to $\mathcal{A}'$ at least one of the two ends will be alive. The strategy we follow is the same as in the previous case: we add in parallel a process on the special location $\odot$ that spawns on one of the two ends a process setting up the connection. However, since we are only guaranteed that one of the two ends will be alive, but we do not know which one, we target both of them.

We also put in parallel two sets of processes which, as in the above case, "alter" the link between the special location $\odot$ and the two ends. By doing so, when one of the two ends will be alive during the transition from $\mathcal{A}$ to $\mathcal{A}'$, the special location $\odot$ will be able to send on it the process setting up the connection. The processes on $\odot$ targeting the other end can simply fail by triggering the spawn when the link between $\odot$ and the target location does not exist.

Finally, let us discuss the changes to the view. To update $\mathcal{V}$ into $\mathcal{V}'$ it is necessary to either remove elements from a location's view or to add new ones. To remove them it suffices to spawn on the target location n a process removing the belief about location m from the special location $\odot$. To update the knowledge of n with the belief of m at incarnation λ it suffices to put in parallel a spawn message, in case the belief must be set up to some λ that is obsolete, or to add a process on the special location $\odot$ that will migrate to m and then spawn a message on n, in case λ belongs to the future of the system (we can not add directly a spawn message, since it would result in a system not well-formed). Notice that also in this case the existence of the necessary links, i.e., between location $\odot$ and both locations n and m, is guaranteed thanks to the presence of processes that alter the state of these links as needed.

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

To transform $\mathcal{A}$ into $\mathcal{A}'$ we need to transform all the incarnation numbers of $\mathcal{A}$ into the ones of $\mathcal{A}'$. Here, we leverage function $\mathsf{transform}$ that for a location n returns us an environment that kills and creates location n to change its incarnation number from $\mathcal{A}(n)$ to $\mathcal{A}'(n)$. We formally define it as

$$\begin{aligned}
\mathsf{transform}(n, \lambda, \kappa) \text{ when } \lambda < 0 \wedge \lambda \neq \kappa &::= [\,\mathsf{create}\, n.\mathbf{0}\,]^{\odot} \parallel \mathsf{transform}(n, |\lambda| + 1, \kappa)\\
\mathsf{transform}(n, \lambda, \kappa) \text{ when } \lambda > 0 \wedge \lambda \neq \kappa &::= [\,\mathsf{spawn}\, n.\mathsf{kill}\,]^{\odot} \parallel \mathsf{transform}(n, -\lambda, \kappa)\\
\mathsf{transform}(n, \lambda, \lambda) &::= \mathbf{0}
\end{aligned}$$

Then, to effectively change all the incarnation numbers it suffices to build the following context

$$L_{\mathcal{A}} = \prod_{n \in \mathsf{supp}(\mathcal{A}').\mathcal{A}'(n) \neq 0} \mathsf{transform}(n, \mathcal{A}(n), \mathcal{A}'(n)) \parallel [\,\mathsf{link}\, n\,]^{\odot} \parallel [\,\mathsf{unlink}\, n\,]^{\odot}$$

To change $\mathcal{L}$ into $\mathcal{L}'$ we need to add and/or remove links. To remove links it suffices to build the following context

$$L^{-}_{\mathcal{L}} = \prod_{(n,m) \in \mathcal{L} \setminus \mathcal{L}'} \begin{array}{l} [\,\mathsf{spawn}\, n.\mathsf{unlink}\, m.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{link}\, n.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{unlink}\, n.\mathbf{0}\,]^{\odot} \parallel \\ [\,\mathsf{spawn}\, m.\mathsf{unlink}\, n.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{link}\, m.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{unlink}\, m.\mathbf{0}\,]^{\odot} \end{array}$$

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

$$L^{+}_{\mathcal{L}} = \prod_{(n,m) \in \mathcal{L}' \setminus \mathcal{L}} \begin{array}{l} [\,\mathsf{spawn}\, n.\mathsf{link}\, m.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{link}\, n.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{unlink}\, n.\mathbf{0}\,]^{\odot} \parallel \\ [\,\mathsf{spawn}\, m.\mathsf{link}\, n.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{link}\, m.\mathbf{0}\,]^{\odot} \parallel [\,\mathsf{unlink}\, m.\mathbf{0}\,]^{\odot} \end{array}$$

To modify the views we need to consider three possibilities, and we build a context for each of them:

- the belief of location n about m in $\mathcal{V}'$ is 0

$$L_{\mathcal{V}_1} = \prod_{\forall n,m \in \mathcal{N}^{\odot} \setminus \tilde{u}. \mathcal{V}'(n)(m)=0} [\,\mathsf{spawn}\ n.\mathsf{forget}\ m\,]^{\odot}$$

- the belief of location n about m in $\mathcal{V}'$ is an incarnation number λ such that $\mathcal{V}'(n)(m) \leq |\mathcal{A}(m)|$

$$L_{\mathcal{V}_2} = \prod_{\forall n,m \in \mathcal{N}^{\odot} \setminus \tilde{u}. 0 < \mathcal{V}'(n)(m) \leq |\mathcal{A}(m)|} [\,\mathsf{spawn}\ n.\mathsf{forget}\ m\,]^{\odot} \parallel \langle (n,0) : \mathbf{0} \rangle^{m}_{\mathcal{V}'(n)(m)}$$

- the belief of location n about m in $\mathcal{V}'$ is an incarnation number λ such that $|\mathcal{A}(m)| < \mathcal{V}'(n)(m) \leq |\mathcal{A}'(m)|$

$$L_{\mathcal{V}_3} = \prod_{\forall n,m \in \mathcal{N}^{\odot} \setminus \tilde{u}. |\mathcal{A}(m)| < \mathcal{V}'(n)(m) \leq |\mathcal{A}'(m)|} [\,\mathsf{spawn}\ m.\mathsf{spawn}\ n.\mathbf{0}\,]^{\odot}$$

In addition, we need to create links to ensure communications can be done, and remove them afterwards.

$$L_{\mathcal{V}_{links}} = \prod_{n,m \in \mathsf{dom}(\mathcal{A}) \cup \mathsf{dom}(\mathcal{A}')} \begin{array}{l} [\,\mathsf{link}\ m\,]^{\odot} \parallel [\,\mathsf{unlink}\ m\,]^{\odot} \parallel \\ [\,\mathsf{link}\ n\,]^{\odot} \parallel [\,\mathsf{unlink}\ n\,]^{\odot} \parallel \\ [\,\mathsf{spawn}\ m.\mathsf{link}\ n\,]^{\odot} \parallel [\,\mathsf{spawn}\ m.\mathsf{unlink}\ n\,]^{\odot} \end{array}$$

The first context, $L_{\mathcal{V}_1}$, handles the case in which the belief of location n about location m is set to 0. Notice that we cannot simply put in parallel a process $[\,\mathsf{forget}\ m\,]^{n}_{|\mathcal{A}'(n)|}$ since it may break the notion of well-founded system as $|\mathcal{A}'(n)|$ may possibly be greater than $|\mathcal{A}(n)|$. In other words, we cannot use processes from the future to set views.

The second context, $L_{\mathcal{V}_2}$, handles the case in which the belief of location n about location m is set to an old incarnation number of m. In this case we can simply extend our system with a message with the desired incarnation number. The forget process is required to erase any knowledge n may have about m which would impede the correct reception of the message.

The third context, $L_{\mathcal{V}_3}$, handles the case in which the belief of location n about location m is set to an incarnation number for m that still does not exist in the system. For this reason we cannot extend it with a message, otherwise we would invalidate the notion of well-founded systems, but rather with a process, which we create from $\odot$ because it possibly is a "future process", that by reducing at the right time will set the view as we desire.

The fourth context, $L_{\mathcal{V}_{links}}$, is required to activate a connection between n and m, again to not use future processes we are obliged to use $\odot$, and to deactivate it afterwards.

Finally, we can set $L = L_{\mathcal{A}} \parallel L^{-}_{\mathcal{L}} \parallel L^{+}_{\mathcal{L}} \parallel L_{\mathcal{V}_1} \parallel L_{\mathcal{V}_2} \parallel L_{\mathcal{V}_3} \parallel L_{\mathcal{V}_{links}}$. □

At first glance, the above result may seem weak since it states that there exists a reduction accomplishing the desired changes but does not say anything about possible other reductions which may not accomplish them.

However, when testing the equivalence of two systems, since the equivalence is universally quantified over all the possible reductions, the guaranteed existence of one computational path leading to the desired changes is enough to ensure that the two systems are also being tested under those changes.

Appendix C. Full Abstraction

C.1. **Soundness.** 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 $\mathsf{po}(\alpha)$, $\mathsf{pi}(\alpha)$ and $\mathsf{pe}(\alpha)$ as follows:

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

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

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

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:

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

Lemma C.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*

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

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

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

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 $S = \Delta \triangleright [\,\overline{x}\langle\tilde{u}\rangle.P\,]^n_\lambda$, $\Delta \vdash n_\lambda : \mathsf{alive}$, as required.
- Rule L-par$_\text{L}$: 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 $N \equiv [\,\overline{x}\langle\tilde{u}\rangle.P\,]^n_\lambda \parallel L$ for some L, with $\Delta \vdash n_\lambda : \mathsf{alive}$. Hence $S \equiv \Delta \triangleright [\,\overline{x}\langle\tilde{u}\rangle.P\,]^n_\lambda \parallel L \parallel M$, $\Delta \vdash n_\lambda : \mathsf{alive}$, as required.
- Rule L-par$_\text{R}$: same as L-par$_\text{L}$.
- Rule L-res: in this case we have $S = \nu\, z.S'$ with $S' \xrightarrow{\alpha} S''$, for some S', S'', and $z \notin \mathsf{fn}(\alpha)$. 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 $x, n \notin z, \tilde{w}$, and $\tilde{v} \subseteq z, \tilde{w}$ as required.
- Rule L-res$_\text{O}$: 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 $T \equiv \nu\, \tilde{w}.\Delta \triangleright [\,\overline{x}\langle\tilde{r}\rangle.P\,]^n_\lambda \parallel N$ for some $\Delta, \tilde{r}, \tilde{w}, P, N$ with $x, n \notin \tilde{w}$, $\tilde{r} \setminus \{z\} \subseteq \tilde{w}$, $\Delta \vdash n_\lambda : \mathsf{alive}$. Hence, we have $S \equiv \nu\, z, \tilde{w}.\Delta \triangleright [\,\overline{x}\langle\tilde{r}\rangle.P\,]^n_\lambda \parallel N$ for some $\Delta, \tilde{r}, \tilde{w}, P, N$ with $x, n \notin z, \tilde{w}$, $\tilde{v} \subseteq z, \tilde{w}$, $\Delta \vdash n_\lambda : \mathsf{alive}$ as required. □

Proposition C.3 (Structural congruence is a strong bisimulation)**.** *We have $\equiv \subseteq \sim$.*

Proof. Since $\equiv$ is an equivalence relation, 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$_\text{O}$ twice, or a combination of rule L-RES and rule L-RES$_\text{O}$. 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 $R \xrightarrow{\alpha} \nu\, v.\nu\, u.T'$. Hence, by applying rule S.RES.C we have $R' \equiv S'$, as required.

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

Rule L-RES$_\text{O}$ **followed by rule** L-RES **:** In this case we have $T \xrightarrow{\omega} T'$ for some T', $v \in \mathsf{po}(\omega)$, $u \notin \mathsf{po}(\omega) \setminus \{v\}$, $\alpha = \nu\, v.\omega$, and $S' = \nu\, u.T'$. Applying rule L-RES we get: $\nu\, u.T \xrightarrow{\omega} \nu\, u.T'$. Applying L-RES$_\text{O}$ 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$_\text{O}$ **:** 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 L-RES as the last rule (rule L-RES$_\text{O}$ is not a possibility for $u \notin \mathsf{fn}(R)$). Hence, we have $R \xrightarrow{\omega} R'$, $S' = \nu\, u.R'$, $u \notin \mathsf{fn}(\omega)$. Now by Lemma C.1, we have $\mathsf{fn}(R') \subseteq \mathsf{fn}(R) \cup \mathsf{pi}(\omega) \cup \mathsf{pe}(\omega)$. Since $u \notin \mathsf{fn}(\omega)$, then $u \notin \mathsf{pi}(\omega)$. Also, since $u \notin \mathsf{fn}(R)$ then $u \notin \mathsf{pe}(\omega)$. 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$_\text{L}$, rule L-PAR$_\text{R}$, rule L-SYNC$_\text{R}$, or rule L-SYNC$_\text{L}$. We consider the four cases:

  Rule L-PAR$_\text{L}$**:** In this case, we have $\Delta \triangleright L \xrightarrow{\alpha} \nu\, \tilde{v}.\Delta' \triangleright L'$ with $\tilde{v} \cap \mathsf{fn}(N) = \emptyset$, and $S' = \nu\, \tilde{v}.\Delta' \triangleright L' \parallel N$. Applying rule L-PAR$_\text{L}$ 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$_\text{R}$**:** 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$_\text{L}$, with $\Delta \triangleright M \xrightarrow{\alpha} \nu\, \tilde{v}.\Delta' \triangleright M'$, and $N' = M' \parallel \mathbf{0}$. But then applying L-PAR$_\text{R}$ 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$_\text{L}$, L-PAR$_\text{R}$, L-SYNC$_\text{L}$ or L-SYNC$_\text{R}$. We consider the different cases:

Rule L-PAR$_\mathrm{L}$**:** In this case we have $\Delta \triangleright U \xrightarrow{\alpha} \nu\,\tilde{v}.\Delta' \triangleright U'$, $\tilde{v} \cap \mathsf{fn}(V) = \emptyset$ 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.

Rule L-PAR$_\mathrm{R}$**:** As above.

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.

Rule L-SYNC$_\mathrm{R}$**:** As above.

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.

- $\tilde{u} = v, \tilde{w}$: 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 $T \xrightarrow{\alpha} T'$, $S' = \nu\,v.T'$. 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 $T \xrightarrow{\omega} T'$, $v \in \mathsf{po}(\omega)$, $\alpha = \nu\,v.\omega$, $S' = T'$. 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. □

Lemma C.4 (Reductions are silent steps)**.** *If $S \longrightarrow S'$ then $S \xrightarrow{\tau} \equiv S'$.*

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

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

Case inferred by SPAWN-C-S**:** Then S is $\Delta \triangleright [\,\mathsf{spawn}\, m.P\,]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-SPAWN-C-S.

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

Case inferred by SPAWN-S**:** Then S is $\Delta \triangleright \langle (m,\kappa) : P\rangle^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-SPAWN-S.

Case inferred by SPAWN-F**:** Then S is $\Delta \triangleright \langle (m,\kappa) : P\rangle^n_\lambda$, 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-UNLINK.

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 FORGET**:** Then S is $\Delta \triangleright [\,\mathsf{forget}\, m.P\,]^n_\lambda$, and the transition $S \xrightarrow{\tau} S'$ follows by L-FORGET.

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-RES 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 C.3 we have $S \xrightarrow{\tau} S''$ with $S'' \equiv T'$. Hence $S \xrightarrow{\tau} S'' \equiv S'$, as required. □

Lemma C.5 (Silent steps are reductions)**.** *If $S \xrightarrow{\tau} S'$ then $S \longrightarrow S'$.*

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

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-C-S**:** Then $S = \Delta \triangleright [\,\mathsf{spawn}\ m.P\,]^n_\lambda$ and $S' = \Delta \triangleright \langle (m : \kappa) : P \rangle^n_\lambda$ with $\kappa = \Delta_{\mathcal{V}}(n)(m)$. The reduction $S \longrightarrow S'$ follows by SPAWN-C-S (the premises in rule SPAWN-C-S are the same as in rule L-SPAWN-C-S).

Case L-SPAWN-C-F**:** Then $S = \Delta \triangleright [\,\mathsf{spawn}\, m.P\,]^n_\lambda$ and $S' = \Delta \ominus n \succ m \triangleright \mathbf{0}$. The reduction $S \longrightarrow S'$ follows by SPAWN-C-F (the premises in rule SPAWN-C-F are the same as in rule L-SPAWN-C-F).

Case L-SPAWN-S**:** Then S is $\Delta \triangleright \langle (m, \kappa) : P \rangle^n_\lambda$, $S' = \nu\, \widetilde{w}.\Delta \oplus m \succ (n, \lambda) \triangleright [\,P\,]^m_\kappa$. The reduction $S \longrightarrow S'$ follows by SPAWN-S (the premises in rule SPAWN-S are the same as in rule L-SPAWN-S).

Case L-SPAWN-F**:** : Then S is $\Delta \triangleright [\,\mathsf{spawn}\, m.P\,]^n_\lambda$, and $S' = \Delta \ominus n \succ m \triangleright \mathbf{0}$ the reduction $S \longrightarrow S'$ follows by SPAWN-F (the premises in rule SPAWN-F are the same as in rule L-SPAWN-F).

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

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 C.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, PAR and STR, we get $S \longrightarrow S'$.

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 C.6. *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 the derivation of $S \xrightarrow{\alpha} S'$ terminates with an application of rule* L-PAR$_{\mathrm{L}}$*, possibly followed by 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 the derivation of* $S \xrightarrow{\alpha} S'$ *terminates with an application of rule* L-PAR$_{\mathrm{R}}$*, possibly followed by applications of rule* L-RES*.*

(3) $\alpha = \tau$, $\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{\nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\tilde{r}.\Delta \triangleright N'$, $\tilde{r} = \tilde{s} \setminus \tilde{w}$ *and* $\tilde{u} \cap \tilde{r} = \emptyset$, $\Delta \triangleright L \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright L'$, *and* $S' \equiv \nu\,\tilde{s}.\Delta \triangleright N' \parallel L'$, *and the derivation of* $S \xrightarrow{\alpha} S'$ *terminates with an application of rule* L-SYNC$_{\mathrm{L}}$*, possibly followed by applications of rule* L-RES*.*

(4) $\alpha = \tau$, $\Delta \triangleright N \xrightarrow{x(\tilde{u})@n_\lambda} \Delta \triangleright N'$, $\nu\,\tilde{s}.\Delta \triangleright L \xrightarrow{\nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\tilde{r}.\Delta \triangleright L'$, $\tilde{r} = \tilde{s} \setminus \tilde{w}$ *and* $\tilde{u} \cap \tilde{r} = \emptyset$, *and* $S' \equiv \nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright N' \parallel L'$, *and the derivation of* $S \xrightarrow{\alpha} S'$ *terminates with an application of rule* L-SYNC$_{\mathrm{R}}$*, possibly followed by applications of rule* L-RES*.*

(5) $\alpha = \nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda$, $\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{\nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\tilde{r}.\Delta \triangleright N'$, $\tilde{r} = \tilde{s} \setminus \tilde{w}$, $\tilde{u} \cap \tilde{r} = \emptyset$, *and* $S' \equiv \nu\,\tilde{r}.\Delta \triangleright N' \parallel L$, *and the derivation of* $S \xrightarrow{\alpha} S'$ *terminates with an application of rule* L-PAR$_{\mathrm{L}}$*, possibly followed by applications of rule* L-RES *or* L-RES$_{\mathrm{O}}$*.*

(6) $\alpha = \nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda$, $\nu\,\tilde{s}.\Delta \triangleright L \xrightarrow{\nu\,\tilde{w}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} \nu\,\tilde{r}.\Delta \triangleright L'$, $\tilde{r} = \tilde{s} \setminus \tilde{w}$, $\tilde{u} \cap \tilde{r} = \emptyset$, *and* $S' \equiv \nu\,\tilde{r}.\Delta \triangleright N \parallel L'$, *and the derivation of* $S \xrightarrow{\alpha} S'$ *terminates with an application of rule* L-PAR$_{\mathrm{R}}$*, possibly followed by applications of rule* L-RES *or* L-RES$_{\mathrm{O}}$*.*

(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* $\tilde{u} \cap \tilde{s} = \emptyset$, *and the derivation of* $S \xrightarrow{\alpha} S'$ *terminates with an application of rule* L-PAR$_{\mathrm{L}}$*, possibly followed by 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* $\tilde{u} \cap \tilde{s} = \emptyset$, *and the derivation of* $S \xrightarrow{\alpha} S'$ *terminates with an application of rule* L-PAR$_{\mathrm{R}}$*, possibly followed by 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$_{\mathrm{L}}$**:** 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$_{\mathrm{R}}$**:** 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$_{\mathrm{L}}$**:** 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 $T = \nu\,\tilde{s}.\Delta \triangleright N \parallel L$, $T' \equiv \nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright N' \parallel L$, $\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{\tau} \nu\,\widetilde{w}.\nu\,\tilde{s}.\Delta \triangleright N'$, with $\mathsf{fn}(L) \cap \widetilde{w} = \emptyset$. Applying rule L-RES, we get $S' \equiv \nu\,a.\nu\,\tilde{w}.\nu\,\tilde{s}.\Delta \triangleright N' \parallel L$, $\nu\,a.\nu\,\tilde{s}.\Delta \triangleright N \xrightarrow{\tau} \nu\,\widetilde{w}.\nu\,a.\nu\,\tilde{s}.\Delta \triangleright N'$, 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 $S = \nu\,w.\nu\,\tilde{s}.\Delta \triangleright N \parallel L \xrightarrow{\tau} \nu\,a.\nu\,\tilde{s}.\Delta \triangleright N' \parallel L' = S'$. 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 $a, \tilde{r} = a, \tilde{s} \setminus \widetilde{w}$, corresponding to assertion 3.

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 C.7 (Bisimilarity is a 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

$$\mathcal{S} \stackrel{\text{def}}{=} \left\{ \langle \nu\,\widetilde{w}.\nu\,\tilde{s}.\Delta_S \triangleright N_S \parallel L,\ \nu\,\widetilde{w}.\nu\,\tilde{r}.\Delta_R \triangleright N_R \parallel L \rangle \ \middle|\ \begin{array}{c} \nu\,\tilde{s}.\Delta_S \triangleright N_S \approx \nu\,\tilde{r}.\Delta_R \triangleright N_R \\ \mathsf{fn}(L) \cap (\tilde{s} \cup \tilde{r}) = \emptyset \end{array} \right\}$$

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

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

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 C.6 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:

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

Since $S \approx R$, we have $R \xRightarrow{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle @ n_\lambda} \nu\,\widetilde{w_r}.\nu\,\widetilde{z_r}.\Delta'_R \triangleright N'_R = R'$ for some $\tilde{w}_r, \Delta'_R, N'_R$, with $\widetilde{z_r} = \tilde{r} \setminus \tilde{a}$, and $R' \approx S'$. Now, since $R \xRightarrow{\tau} R_1 \xrightarrow{\overline{x}\langle\tilde{u}\rangle @ n_\lambda} R_2 \xRightarrow{\tau} R'$, 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 $V = \nu\,\widetilde{w_r}.\nu\,\tilde{r}.\Delta'_R \rhd N'_R \parallel L'$ and with $\mathsf{fn}(L') \cap \widetilde{w_r} = \emptyset$ because of the conditions of rule L-PAR$_\text{L}$.
Summing up, we have:

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

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 $S' = \nu\,\widetilde{z_s}.\Delta_S \rhd N'_S \approx \nu\,\tilde{w}_r.\nu\,\widetilde{z_r}.\Delta'_R \rhd N'_R = R'$, 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 $S_L = \Delta_S \rhd N_S \parallel L$, $U = \nu\,\tilde{s}.\Delta'_S \rhd N'_S \parallel L$, $S \xrightarrow{\tau} \nu\,\tilde{s}.\Delta'_S \rhd N'_S$, with $\mathsf{fn}(L) \cap \tilde{s} = \emptyset$. Since $S \approx R$, we have $R \stackrel{\tau}{\Longrightarrow} R'$ with $R' \approx S'$ and $R' = \nu\,\tilde{r}.\Delta'_R \rhd N'_R$. Now applying repeatedly rule L-PAR$_\text{L}$, we get $R_L \stackrel{\tau}{\Longrightarrow} \nu\,\tilde{r}.\Delta'_R \rhd N'_R \parallel L = V$, 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 C.6 for $S_L = \nu\,\tilde{s}.\Delta_S \rhd N_S \parallel L$ and $S_L \xrightarrow{x(\tilde{u})@n_\lambda} U$:

Case L-PAR$_\text{L}$**:** In that case, we have $S = \nu\,\tilde{s}.\Delta_S \rhd N_S$, $S \xrightarrow{x(\tilde{u})@n_\lambda} S'$, $S' = \nu\,\tilde{s}.\Delta_S \rhd N'_S$. Since $S \approx R$, we have $R \stackrel{x(\tilde{u})@n_\lambda}{\Longrightarrow} R'$, where $R' = \nu\,\tilde{z}.\nu\,\tilde{r}.\Delta'_R \rhd N'_R$. Now, by repeated application of rule L-PAR$_\text{L}$, we get $R_L \stackrel{x(\tilde{u})@n_\lambda}{\Longrightarrow} \nu\,\tilde{w}.\nu\,\tilde{r}.\Delta'_R \rhd N'_R \parallel L = V$, with $\mathsf{fn}(L) \cap \tilde{z} = \emptyset$ 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 C.6 for $S_L = \nu\,\tilde{s}.\Delta_S \rhd N_S \parallel L$ 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 $S = \nu\,\tilde{s}.\Delta_S \rhd N_S$, $S \xrightarrow{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda} S'$ $S' = \nu\,\widetilde{s_a}.\Delta_S \rhd N'_S$, $U = \nu\,\tilde{s_a}.\Delta_S \rhd N'_S \parallel L$, $\widetilde{s_a} = \tilde{s} \setminus \tilde{a}$. Since $S \approx R$, we have $R \stackrel{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda}{\Longrightarrow} R'$, where $R' \equiv \nu\,\tilde{w}.\nu\,\widetilde{r_a}.\Delta'_R \rhd N'_R$, $\widetilde{r_a} = \tilde{r} \setminus \tilde{a}$, and $S' \approx R'$.
Now, by repeated application of rule L-PAR$_\text{L}$, we get $R_L \stackrel{\nu\,\tilde{a}.\overline{x}\langle\tilde{u}\rangle@n_\lambda}{\Longrightarrow} \nu\,\tilde{z}.\nu\,\widetilde{r_a}.\Delta'_R \rhd N'_R \parallel L = V$, with $\mathsf{fn}(L) \cap \tilde{z} = \emptyset$ because of the conditions

of rule L-PAR_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_R**:** Similar to the case L-PAR_L, but simpler.

Case $\mathsf{create}(n, \lambda)$**:** In this case, we have:

$$S_L = \nu\,\tilde{s}.\Delta_S \triangleright N_S \parallel L \xrightarrow{\mathsf{create}(n,\lambda)} \nu\,\tilde{s}.\Delta'_S \triangleright N_S \parallel L = U$$

$$S = \nu\,\tilde{s}.\Delta_S \triangleright N_S \xrightarrow{\mathsf{create}(n,\lambda)} \nu\,\tilde{s}.\Delta'_S \triangleright N_S = S'$$

Since $S \approx R$, we have $R \xRightarrow{\mathsf{create}(n,\lambda)} R' = \nu\,\tilde{r}.\Delta'_R \triangleright N'_R$, with $S' \approx R'$. Now, by one application of rule L-CREATE-EXT, we obtain $R_L \xRightarrow{\mathsf{create}(n,\lambda)} \nu\,\tilde{r}.\Delta'_R \triangleright N'_R \parallel L = V$. Since we have $\mathsf{fn}(L) \cap \tilde{r} = \emptyset$ by definition we have $\langle U, V\rangle \in \mathcal{S}$.

Case $\mathsf{kill}(n, \lambda)$**:** Similar to above.

Case $\oplus n \mapsto m$**:** Similar to above.

Case $\ominus n \mapsto m$**:** Similar to above.

Case $n \succ m$**:** Similar to above.

Case $\widetilde{w} = w, \widetilde{z}$**:** Let $\langle \nu\,w.\nu\,\widetilde{z}.S, \nu\,w.\nu\,\widetilde{z}.R\rangle \in \mathcal{S}$. By construction $\langle \nu\,\widetilde{z}.S, \nu\,\widetilde{z}.R\rangle \in \mathcal{S}$. By induction hypothesis, if $\nu\,\widetilde{z}.S \xrightarrow{\alpha} U$, then $\nu\,\widetilde{z}.R \xrightarrow{\alpha} V$ for some V with $\langle U, V\rangle \in \mathcal{S}$. Now using rule L-RES or rule L-RES_O, we obtain either $S_L \xrightarrow{\alpha} \nu\,w.U$ or $S_L \xrightarrow{\nu\,w.\alpha} U$, 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 C.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 C.2. It is weak reduction-closed thanks to Lemma C.5. It is a system congruence thanks to Proposition C.7. Thus $\approx \subseteq \dot{\approx}$ by definition of weak barbed congruence as the largest of weak barb-preserving, reduction-closed, system congruence. □

C.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 $\nu\,\tilde{s}.\Delta \triangleright N \parallel M$ provided $\mathsf{fn}(M) \cap \tilde{s} = \emptyset$.

Lemma C.9 (Inducing Network Changes)**.**

- *Suppose* $S = \nu\,\tilde{u}.\Delta \triangleright N$ *and* $\Delta \vdash (n, \lambda) : \mathsf{alive}$
  - $S \xrightarrow{\mathsf{kill}(n,\lambda)} S'$ *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* $S = \nu\,\tilde{u}.\Delta \triangleright N$ *and* $\Delta \vdash (n, \lambda) : \mathsf{dead}$
  - $S \xrightarrow{\mathsf{create}(n,\lambda)} S'$ *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* $S = \nu\,\tilde{u}.\Delta \triangleright N$ *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* $S' \equiv \nu\,\tilde{u}.\Delta' \triangleright N$, $\Delta' \vdash n \not\leftrightarrow m$ *implies* $S \xrightarrow{\ominus n_\lambda \mapsto m} S'$

- *Suppose* $S = \nu\,\tilde{u}.\Delta \triangleright N$ *and* $\Delta \vdash n \not\leftrightarrow m$
  - $S \xrightarrow{\oplus n_\lambda \mapsto m} S'$, *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 of $S = \nu\,\tilde{u}.\Delta \triangleright N \xrightarrow{\mathsf{kill}(n,\lambda)} \nu\,\tilde{u}.\Delta' \triangleright N = S'$. 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 $\nu\,\tilde{u}.\Delta \oplus (l, 1) \triangleright N$.

Lemma C.10. *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. Easy, since $\mathcal{R} = \{\langle \nu\, x, l.S^l \parallel [\,\overline{x}\,]^l_\lambda, S\rangle \mid S \in \mathbb{S} \text{ closed},\ x, l \notin \mathsf{fn}(S)\}$ 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 $S \xrightarrow{\alpha} T$. □

Lemma C.11. *If we have $S^l \parallel [\,\overline{x}\,]^l_\lambda \dot{\approx} R^l \parallel [\,\overline{x}\,]^l_\lambda$ with x, l fresh for S, R then $S \dot{\approx} R$.*

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

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

and then by Lemma C.10, Proposition C.8, and transitivity of $\dot{\approx}$ we have $S \dot{\approx} R$ as required. □

Proposition C.12 (Completeness of $\approx$ w.r.t. $\dot{\approx}$)**.** $\dot{\approx}\ \subseteq \approx$.

Proof. To prove the statement, since $\dot{\approx}$ is an equivalence relation and hence symmetric, it suffices to show that $\dot{\approx}$ is a weak simulation up-to $\equiv$. Take $S \dot{\approx} R$ and suppose that $S \xrightarrow{\alpha} S'$; we reason by case analysis on α.

Case $\alpha = \tau$: thanks to Lemma C.4 and Lemma C.5, the thesis follow by the reduction closure property.

Case $\alpha = \overline{x}\langle\tilde{u}\rangle@n_\lambda$: consider the context

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

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 $S \dot{\approx} R$, we must have a transition $R \parallel L \Longrightarrow T_2$ with $T_1 \dot{\approx} T_2$. Then, since $T_1{\downarrow_{succ@l}}$, $T_1{\not\Downarrow_{fail@l}}$ and $T_1 \dot{\approx} T_2$ we must have $T_2{\Downarrow_{succ@l}}$ 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 derivation 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 Lemma C.11 we have $(S', R') \in \dot{\approx}$ as required.

Case $\alpha = \nu\,\tilde{v}.\overline{x}\langle\tilde{u}\cdot\tilde{v}\rangle@n_\lambda$: For this case we define the following macros

$$\mathsf{if}\ x \notin \tilde{v}\ \mathsf{then}\,P\ \mathsf{else}\ Q \equiv \mathsf{if}\ x = \tilde{v}_1\ \mathsf{then}\,Q\ \mathsf{else}\ \mathsf{if}\ldots\mathsf{else}\ \mathsf{if}\ x = \tilde{v}_n\ \mathsf{then}\,P\ \mathsf{else}\ Q$$

$$\mathsf{if}\ \tilde{u}\cap\tilde{v} = \emptyset\ \mathsf{then}\,P\ \mathsf{else}\ Q \equiv \mathsf{if}\ \tilde{u}_1 \notin \tilde{v}\ \mathsf{then}\,Q\ \mathsf{else}\ \mathsf{if}\ldots\mathsf{if}\ \tilde{u}_n \notin \tilde{v}\ \mathsf{then}\,P\ \mathsf{else}\ Q\ldots\mathsf{else}\ Q$$

where by $\tilde{v}_i$ we mean the i-th element of $\tilde{v}$.

In addition, for simplicity, here we assume that the private names in $\tilde{u}$ are in the final part of the vector, i.e., $\tilde{u} = \tilde{u}'\cdot\tilde{v}$. This implies a simple nesting of the if primitives to test the privateness of names. In the other cases, when private names are freely mixed with public ones, a testing context can always be found by nesting the ifs following the order of the names. Consider the context

$$L = [\,x(\tilde{y}\cdot\tilde{y}').\mathsf{if}\ \tilde{y} = \tilde{u}\ \mathsf{then}\,.\mathsf{if}\ \tilde{y}'\cap\tilde{w} = \emptyset\ \mathsf{then}\,\mathsf{create}\,l.(\overline{fail} \mid fail.\overline{succ})\,]^n_\lambda$$

with l, $fail$, $succ$ fresh and $\tilde{w} = fn(S,R)$. The reasoning then proceeds mostly as in the previous case.

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 $S \dot{\approx} R$, we must have a transition $R \parallel L \Longrightarrow T_2$ s.t. $T_1 \dot{\approx} T_2$. Now, since $T_1{\downarrow_{succ@l}}$, $T_1{\not\Downarrow_{fail@l}}$ and $T_1 \dot{\approx} T_2$ we must also have $T_2{\downarrow_{succ@l}}$ and $T_2{\not\Downarrow_{fail@l}}$ (the barb on $succ@l$ is strong for reasons given in the second case above). 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 C.11 we have $(S', R') \in \dot{\approx}$ as required.

Case $\alpha = \oplus n_\lambda \mapsto m$: by Lemma C.9 we know that $[\,\mathsf{link}\,m.P\,]^n_\lambda$ induces the desired labeled action. Consider the context

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

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 $S \dot{\approx} R$ we must have a transition $R \parallel L \Longrightarrow T_2$ s.t. $T_1 \dot{\approx} T_2$. Now, since $T_1{\downarrow_{succ@l}}$, $T_1{\not\Downarrow_{fail@l}}$ and $T_1 \dot{\approx} T_2$ we must also have $T_2{\downarrow_{succ@l}}$, $T_2{\not\Downarrow_{fail@l}}$ (the barb on $succ@l$ is strong for reasons given in the second case above). The only way to obtain this is if $R \Longrightarrow R_1 \parallel [\,\mathsf{link}\,m.\mathsf{create}\,l.(\overline{fail} \mid fail.\overline{succ})\,]^n_\lambda \longrightarrow R_1 \parallel [\,\mathsf{create}\,l.(\overline{fail} \mid fail.\overline{succ})\,]^n \Longrightarrow T_2$, where $T_2 \equiv R'^l \parallel [\,\overline{succ}\,]^l$.

Then, by Lemma C.9 we know that $R \xRightarrow{\tau} R_i \xrightarrow{\ominus n_\lambda \mapsto m} R_j \xRightarrow{\tau} R'$.

Now, since $(S'^l \parallel [\,\overline{succ}\,]^l, R'^l \parallel [\,\overline{succ}\,]^l) \in \dot{\approx}$, by Lemma C.11 we have $(S', R') \in \dot{\approx}$ as required.

Case $\alpha = \ominus n_\lambda \mapsto m$: consider the context

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

with l, $succ$ and $fail$ fresh. The reasoning is similar to the previous case.

Case $\alpha = \mathsf{kill}(n,\lambda)$: by Lemma C.9 we know that $[\,\mathsf{kill}\,]^n_\lambda$ is the process that induces the reduction we are looking for. Consider the context

$$L = [\,\overline{fail}\,]^n_\lambda \parallel [\,\mathsf{kill}\,]^n_\lambda$$

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

Now, since $S \dot{\approx} R$ there must be a matching move $R \parallel L \Longrightarrow T_2$ s.t. $T_1 \dot{\approx} T_2$. Then, since $T_1 \not\Downarrow_{fail@n}$ and $T_1 \dot{\approx} T_2$ we must have $T_2 \not\Downarrow_{fail}$. Since $fail$ is fresh the only way to have $T_2 \not\Downarrow_{fail@n}$ is to have $R \parallel L \Longrightarrow R_1 \parallel [\,\overline{fail}\,]^n_\lambda \parallel [\,\mathsf{kill}\,]^n_\lambda \longrightarrow R_1 \parallel [\,\overline{fail}\,]^n_\lambda \Longrightarrow T_2$.

By Lemma C.9 we know that $R \overset{\tau}{\Longrightarrow} R_1 \xrightarrow{\mathsf{kill}(n,\lambda)} R_1' \overset{\tau}{\Longrightarrow} R'$.

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

Case $\alpha = \mathsf{create}(n, \lambda)$**:** consider the context

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

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

Case $\alpha = n_\lambda \succ m$**:** in this case the context is slightly more involved than the other ones. This because the label $n_\lambda \succ m$ only informs us about the correct view of location n_λ toward location name m, but does not tell us anything about the aliveness of the link between the two. Hence, we need to build a context that accounts for the case in which the two are already connected and for the case in which a connection between the two must be established. Consider the context

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

with l, $fail$, $succ_1$ and $succ_2$ fresh. Now, since $S \dot{\approx} R$ then the public network of the two systems must agree on aliveness of locations, links, and views. We consider now the two cases.

Case $n \not\leftrightarrow m$**:** consider the transition

$$S \parallel L \Longrightarrow T_1 \equiv \nu\, x.S^l \parallel \left[\begin{array}{l} x.\mathsf{spawn}\, m. \\ \mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ}) \end{array}\right]^n_\lambda \parallel [\,\overline{succ_1}\,]^l$$

where $\overline{x}$ synchronizes with the first process (and the second one remains inactive). Now, since $S \dot{\approx} R$ there must be a matching move $R \parallel L \Longrightarrow T_2$ such that $T_1 \dot{\approx} T_2$. Then, since $T_1 \downarrow_{succ_1@l}$ and $T_1 \not\Downarrow_{fail@l}$ and $T_1 \dot{\approx} T_2$ we also have $T_2 \downarrow_{succ_1@l}$ and $T_2 \not\Downarrow_{fail@l}$. 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 $R \xRightarrow{n_\lambda \succ m} R'$.
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 Lemma C.11 we have $(S', R') \in \dot{\approx}$ as required.

Case $n \leftrightarrow m$**:** consider the transition

$$S \parallel L \Longrightarrow T_1 \equiv S^l \parallel \left[\begin{array}{l} x.\mathsf{link}\, m.(\mathsf{spawn}\, m.(\mathsf{unlink}\, n. \\ \mathsf{create}\, l.(\overline{fail} \mid fail.\overline{succ_1})) \end{array}\right]^n_\lambda \parallel [\,\overline{succ_2}\,]^l$$

where $\overline{x}$ synchronizes with the second process (and the first one remains inactive). Now, since $S \dot{\approx} R$ we have a transition $R \parallel L \Longrightarrow T_2$ such that $T_1 \dot{\approx} T_2$. By $T_1 \downarrow_{succ_2@l}$, $T_1 \not\Downarrow_{fail@l}$ and $T_1 \dot{\approx} T_2$ we have $T_2 \downarrow_{succ_2@l}$ and $T \not\Downarrow_{fail@l}$. Hence we have

$$T_2 \equiv R'^l \parallel \left[\begin{array}{l} x.\mathsf{link}\ m.(\mathsf{spawn}\ m.(\mathsf{unlink}\ n. \\ \mathsf{create}\ l.(\overline{fail} \mid fail.\overline{succ_1})) \end{array} \right]^n_\lambda \parallel [\,\overline{succ_2}\,]^l$$

This is possible only if $R \xrightarrow{n_\lambda \succ m} R'$.

Now,

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

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 Lemma C.11 we have $(S', R') \in \dot{\approx}$ as required. □

Appendix D. Bisimulation Of Running Example

We now show the proof of Example 5.1

We recall briefly the two systems.

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

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

where:

$$I = req(y,z).\mathsf{spawn}\ n_r.\overline{r_1}\langle y,z\rangle$$

$$R = (r_1(y,z).\mathsf{spawn}\ n_b.\overline{b}\langle y,z\rangle) \mid (r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle)$$

$$B = b(y,z).\mathsf{spawn}\ n_r.\overline{r_2}\langle z, w_y\rangle$$

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

$$C = \nu\, c.\overline{c}\ \mid !c.(\overline{c}\ \mid \mathsf{create}\ n_r.(R \mid \mathsf{spawn}\ n_i.\overline{retry}))$$

and where the networks Δ and Δ' are defined as follows:

$$\Delta_{\mathcal{A}} = \{n_i \mapsto 1, n_r \mapsto 1, n_b \mapsto 1\}$$

$$\Delta_{\mathcal{L}} = \{n_i \leftrightarrow n_r, n_r \leftrightarrow n_b\}$$

$$\Delta_{\mathcal{V}} = \{n_i \mapsto \{n_r \mapsto 1\}, n_b \mapsto \{n_r \mapsto 1\}, n_r \mapsto \{n_i \mapsto 1, n_b \mapsto 1\}\}$$

$$\Delta'_{\mathcal{A}} = \{n_i \mapsto 1, n_r \mapsto 1, n_b \mapsto 1, n_c \mapsto 1\}$$

$$\Delta'_{\mathcal{L}} = \{n_i \leftrightarrow n_r, n_r \leftrightarrow n_b\}$$

$$\Delta'_{\mathcal{V}} = \{n_i \mapsto \{n_r \mapsto 1\}, n_b \mapsto \{n_r \mapsto 1\}, n_r \mapsto \{n_i \mapsto 1, n_b \mapsto 1\}, n_c \mapsto \hat{\mathbf{0}}\}$$

Proposition D.1. ***servD*** $\approx$ ***servDFR***

Proof. In the following, for the sake of legibility, 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

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

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

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

Note that $(\mathbf{servD}, \mathbf{servDFR}) \in \mathcal{S}_0$. 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 $req(y,z)@n_i$.
    - Case $\mathbf{servD} \xrightarrow{req(y,z)@n_i} T_1$,
      where

$$T_1 \equiv \nu\, n_r, n_b, r_1, r_2, b.\Delta \triangleright [\,\mathsf{spawn}\ n_r.\overline{r_1}\langle y, z\rangle\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,B\,]^{n_b}$$

      The move can then be matched by

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

      and $(T_1, T_2) \in \mathcal{S}_1 \subseteq \mathcal{R}$.
    - Case $\mathbf{servDFR} \xrightarrow{req(y,z)@n_i} T_2$. Similar to above.
  - Case τ.
    - Consider the transition

$$\mathbf{servDFR} \xrightarrow{\tau}$$
$$T_2 \equiv \nu\, n_b, n_r, b, s, retry, n_c, c.\Delta \triangleright$$
$$[\,K\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel$$
$$[\,\overline{c} \mid !c.(\mathsf{create}\ n_r.(R \mid \mathsf{spawn}\ n_i.\overline{retry}) \mid \overline{c})\,]^{n_c}$$

      Then, the move can be matched by the other system by doing nothing, indeed $(\mathbf{servD}, T_2) \in \mathcal{S}_1 \subseteq \mathcal{R}$
    - Consider the transition

$$\mathbf{servDFR} \xrightarrow{\tau} T_2 \equiv \nu\, n_r, n_b, a, b, s, retry, n_c.\Delta \ominus n_r \triangleright$$
$$[\,K\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c}$$

      Then, the move can be matched by the other system by doing nothing, indeed $(\mathbf{servD}, T_2) \in \mathcal{S}_1 \subseteq \mathcal{R}$
- Pairs $(\mathbf{servD}, R_0) \in \mathcal{S}_0$ are handled similarly as in the case above.
- 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

$$\nu\,\tilde{u}.\Delta \triangleright [\,\overline{z}\langle w\rangle\,]^{n_1}$$

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

$$R_1 \equiv \nu\, n_r, n_b, n_c, r_1, r_2, b, c, retry.\Delta'' \triangleright \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,retry.\mathsf{spawn}\ n_r.\overline{r_1}\langle y,c\rangle\,]^{n_i} \parallel \\ [\,\overline{retry} \mid R\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c} \end{pmatrix}$$

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.

$$R_1 \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright \begin{matrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,retry.\mathsf{spawn}\ n_r.\overline{r_1}\langle y,c\rangle\,]^{n_i} \parallel [\,\overline{retry}\,]^{n_r} \parallel [\,R\,]^{n_r} \parallel \\ [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c} \end{matrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\mathsf{spawn}\ n_r.\overline{r_1}\langle y,c\rangle\,]^{n_i} \parallel [\,R\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\overline{r_i}\langle y,c\rangle\,]^{n_r} \parallel [\,R\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \vartriangle \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\overline{r_1}\langle y,c\rangle\,]^{n_r} \parallel [\,r_1(y,z).\mathsf{spawn}\ n_b.\overline{b}\langle y,z\rangle\,]^{n_r} \parallel \\ [\,r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c} \end{pmatrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\mathsf{spawn}\ n_b.\overline{b}\langle y,c\rangle\,]^{n_r} \parallel [\,r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle\,]^{n_r} \parallel \\ [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c} \end{pmatrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\overline{b}\langle y,c\rangle\,]^{n_b} \parallel [\,r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle\,]^{n_r} \parallel \\ [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c} \end{pmatrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\overline{b}\langle y,c\rangle\,]^{n_b} \parallel [\,r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle\,]^{n_r} \parallel \\ [\,\mathsf{kill}\,]^{n_r} \parallel [\,b(y,z).(!B \mid \mathsf{spawn}\ n_r.\overline{r_2}\langle z,w_y\rangle)\,]^{n_b} \parallel [\,C\,]^{n_c} \end{pmatrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel \\ [\,!B \mid \mathsf{spawn}\ n_r.\overline{r_2}\langle c,w_y\rangle)\,]^{n_b} \parallel [\,C\,]^{n_c} \end{pmatrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel \\ [\,!B\,]^{n_b} \parallel [\,\mathsf{spawn}\ n_r.\overline{r_2}\langle c,w_y\rangle\,]^{n_b} \parallel [\,C\,]^{n_c} \end{pmatrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright \begin{pmatrix} [\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,r_2(y,z).\mathsf{spawn}\ n_i.\overline{z}\langle y\rangle\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel \\ [\,!B\,]^{n_b} \parallel [\,\overline{r_2}\langle c,w_y\rangle\,]^{n_r} \parallel [\,C\,]^{n_c} \end{pmatrix} \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright ([\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\mathsf{spawn}\ n_i.\overline{c}\langle w_y\rangle\,]^{n_r} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c}) \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright ([\,c(w).\overline{z}\langle w\rangle\,]^{n_i} \parallel [\,\overline{c}\langle w_y\rangle\,]^{n_i} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c}) \xrightarrow{\tau}$$

$$\nu\,\tilde{u}.\Delta'' \triangleright ([\,\overline{z}\langle w_y\rangle\,]^{n_i} \parallel [\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c}) \xrightarrow{\overline{z}\langle w_y\rangle@n_i}$$

$$T_2 \equiv \nu\,\tilde{u}.\Delta'' \triangleright ([\,\mathsf{kill}\,]^{n_r} \parallel [\,!B\,]^{n_b} \parallel [\,C\,]^{n_c})$$

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.

□