Timed, Distributed, Probabilistic, Typed Processes

Martin Berger and Nobuko Yoshida

Department of Computing, Imperial College London

Abstract. This paper studies types and probabilistic bisimulations for a timed π-calculus as an effective tool for a compositional analysis of probabilistic distributed behaviour. The types clarify the role of timers as interface between non-terminating and terminating communication for guaranteeing distributed liveness. We add message-loss probabilities to the calculus, and introduce a notion of approximate bisimulation that discards transitions below a certain specified probability threshold. We prove this bisimulation to be a congruence, and use it for deriving quantitative bounds for practical protocols in distributed systems, including timer-driven message-loss recovery and the Two-Phase Commit protocol.

1 Introduction

Designing formalisms for the development and verification of distributed systems (DS), understood as computation in the presence of partial failures and malevolent adversaries [26], is challenging: DS are often written in semantically rich high-level languages with expressive typing systems. Failures in DS (e.g. message-loss) are typically detected by timing. Hence programming environments for DS feature sophisticated timer mechanisms. State explosion, the key problem in verification, is worse in DS because timing and partial failures increase dramatically the space of possible system behaviour. Finally, although DS operate in adversarial environments, the behaviour of adversaries and the occurrences of failures is nevertheless structured: for example message-loss in the Internet can be assumed to occur only rarely, and cryptography is based on similar frequentist suppositions about attackers guessing passwords correctly. Such structural assumptions are usually phrased in probabilistic terms. We propose a type-based analysis for DS that addresses these points using a distributed, timed and probabilistic π-calculus: our formalism is not intended as an ultimate model for all issues pertaining to DS, but a good starting point for further investigations, capturing the above central elements in DS.

Following [3], we turn the π-calculus into a model for DS by adding locations and allowing messages travelling between locations (but not within a location) to be lost. This notion of failure exhibits a key feature of DS (as timers usually are provided to handle message-loss), and is a stepping stone towards more advanced forms of failure. As timed computation is not known to be reducible to untimed computation, we introduce (discrete) timing directly into our model via timers.

To allow compositional verification, to reduce the state space of checking distributed process behaviour and to aid programming DS, we use *types*. Types constrain processes and their contexts, leading to a marked reduction of possible behaviour. In this paper, we focus on the *linear type discipline* studied and extended for various purposes since

[18, 23]. One of the key concepts of this discipline is to handle a combination of terminating and (potentially) non-terminating communication channels [19] (often called deadlock-freedom [20, 22]). This distinction is highly useful in DS, where exchanges on (potentially) non-terminating channels correspond to remote communication, while termination is restricted to local interaction. For example loss of liveness due to failure is counteracted in DS by aborting distributed invocations that do not respond within some expected time. We model this situation with typing rules for timers such that a timer is a *converter* from (potential) non-terminating to terminating channels. Timers thus offer channel-based distributed liveness where an output at a certain channel can eventually happen, despite failures, thus guaranteeing transparency between remote and local invocations in the presence of message-loss. In addition, the linear type discipline has the capability to model DS with failures and timers where applications written in typed high-level languages run inside distributed locations, without losing expressivity and usability. For example it can be extended to type-based programming analyses such as secure information flow [19, 21], embeds various higher-order programming languages fully abstractly [6, 19, 32] and models distributed Java [1] and web services [8]. The main contribution of this paper is to show that the linear type discipline generalises smoothly to distributed computation: no new types have to be invented and all existing typing rules stay unchanged.

To demonstrate the stability and applicability of our types, we add message-loss probabilities to reason about fine-grained behaviours of distributed processes. We use probabilistic automata [28] so non-determinism and probabilities can coexist. We define a new form of approximate bisimulation that expresses that two processes are equivalent, except that some of their transitions are allowed not to be matched, as long the probability of those unmatched transitions is below some factor $\varepsilon \in [0, 1]$. Because the typing system works for non-deterministic message loss, it is also sound when message-loss is governed by probabilities, justifying our typing system. We prove our approximate bisimulation to be a *congruence*, allowing compositional reasoning for DS. We then use this approximate bisimulation to analyse message failure recovery mechanisms. An RPC protocol (similar to a concurrent alternating bit protocol in [2], but extended with timers and message forwarding), distributed leases [16] and the Two-Phase Commit protocol (2PCP) with probabilistic message-loss are specified in our timed π-calculus and reasoned about with the approximate bisimulation. Types and their liveness property reduce the number of transitions to be compared in proofs for DS. Omitted definitions and proofs are in [7]. We close with a summary of our results.

- We present the first typing for timers as a smooth generalisation of the linear type disciplines. This clarifies the behavioural status of timers as a converter from non-terminating to terminating behaviour. As a simple instance, we use the linear/affine typing system from [19] where linear and affine stand for termination and (potential) non-termination, respectively. This extension is achieved without additional types. All existing typing rules are unchanged on processes. The usefulness of the typing system is demonstrated by examples.
- We introduce approximate bisimulation for the distributed π-calculus and prove congruency. As far as we know, this is the first congruency result for probabilistic bisimulations of the π-calculus.

– We offer convenient verification of quantitative bounds for probabilistic processes. We reason about some distributed protocols, and demonstrate the use of types for reducing the burden of checking for bisimilarity of probabilistic processes.

2 Distributed Timed Processes

The syntax of *processes* $(P,Q,...)$ and *networks* $(M,N,...)$ is given by the following grammar where $a,b,x,y,v,...$ range over a countably infinite set of names.

$$P ::= 0 \mid \overline{a}\langle\tilde{v}\rangle \mid a(\tilde{x}).P \mid !a(\tilde{x}).P \mid P \mid Q \mid (\nu a)P \mid \mathsf{timer}^t\langle a(\tilde{x}).P,\ Q\rangle$$
$$M ::= 0 \mid [P] \mid M \mid N \mid (\nu a)M$$

In applications we often also communicate values in outputs, e.g. $\overline{x}\langle 3\rangle$. We often abbreviate $\overline{x}\langle\rangle$ to $\overline{x}$ and $x().P$ to $x.P$. We write $\overline{x}(\tilde{c})P$ for $(\nu\tilde{c})(\overline{x}\langle\tilde{c}\rangle|P)$ when $x \notin \tilde{c}$. Free names $\mathsf{fn}(\cdot)$ are defined as usual, except that $\mathsf{fn}(\mathsf{timer}^t\langle a(\tilde{x}).P,\ Q\rangle) = \mathsf{fn}(a(\tilde{x}).P) \cup \mathsf{fn}(Q)$ and $\mathsf{fn}([P]) \stackrel{\text{def}}{=} \mathsf{fn}(P)$. The structural congruence $\equiv$ is defined as usual, but over our extended syntax: for example, we have $[(\nu a)P] \equiv (\nu a)[P]$. The *timer* construct $\mathsf{timer}^t\langle a(\tilde{x}).P,\ Q\rangle$, with $t > 0$ being an integer, supports two operations: (1) *timeout* which means that after t steps it turns into Q, unless (2) it has been *stopped*, i.e. that a message has been received by the timer at a. It is easy to modify timers so they can be stopped at multiple channels.

$$\textsc{Stop}\ \frac{}{\mathsf{timer}^{t+1}\langle a(\tilde{x}).P,\ Q\rangle \mid \overline{a}\langle\tilde{v}\rangle \to P\{\tilde{v}/\tilde{x}\}}$$

The flow of time is communicated at each step in the computation by a *timestepper function* ϕ, which acts on processes. It models the implicit broadcast of time passing. The main two rules are:

$$\phi(\mathsf{timer}^{t+1}\langle P,\ Q\rangle) = \mathsf{timer}^t\langle P,\ Q\rangle \qquad \phi(\mathsf{timer}^1\langle P,\ Q\rangle) = Q$$

The others are: $\phi(P \mid Q) = \phi(P) \mid \phi(Q)$, $\phi((\nu a)P) = (\nu a)\phi(P)$ and otherwise $\phi(P) = P$. The remaining reduction rules for processes are given as follows:

$$\textsc{Par}\ \frac{P \to Q}{P|R \to Q|\phi(R)} \quad \textsc{Com}\ \frac{}{\overline{a}\langle\tilde{v}\rangle|a(\tilde{x}).P \to P\{\tilde{v}/\tilde{x}\}} \quad \textsc{Rep}\ \frac{}{\overline{a}\langle\tilde{v}\rangle|!a(\tilde{x}).P \to P\{\tilde{v}/\tilde{x}\}|!a(\tilde{x}).P}$$
$$\textsc{Res}\ \frac{P \to Q}{(\nu a)P \to (\nu a)Q} \quad \textsc{Idle}\ \frac{}{P \to \phi(P)} \quad \textsc{Cong}\ \frac{P \equiv P' \to Q' \equiv Q}{P \to Q}$$

Note that we have $\phi(R)$ in the conclusion of [PAR]: this ensures that each *active timer*, that is any timer not under a prefix, is ticked one unit at each interaction. [IDLE] prevents the flow of time from ever being halted by deadlocked processes. The reduction relation of the networks, $\to$, is defined as:

$$\textsc{Intra}\ \frac{P \to Q}{[P] \to [Q]} \quad \textsc{Par}\ \frac{M \to N}{M|L \to N|L} \quad \textsc{Res}\ \frac{M \to N}{(\nu a)M \to (\nu a)N} \quad \textsc{Cong}\ \frac{M \equiv M' \to N' \equiv N}{M \to N}$$
$$\textsc{Loss}\ \frac{}{[P|\overline{a}\langle\tilde{v}\rangle] \mid [Q|a(\tilde{y}).R] \to [\phi(P)] \mid [Q|a(\tilde{y}).R]} \quad \textsc{Com}\ \frac{}{[P|\overline{a}\langle\tilde{v}\rangle]|[Q|a(\tilde{y}).R] \to [\phi(P)]|[\phi(Q)|R\{\tilde{v}/\tilde{y}\}]}$$

Rules for replicated input or timer input corresponding to [LOSS, COM] have been omitted. We define: $\twoheadrightarrow \stackrel{\text{def}}{=} (\equiv \cup \to)^*$. [PAR] at the network level prevents the synchronisation

of clocks between sites: this models that over the Internet, clock-synchronisation better than about 100 milliseconds (which is several orders of magnitude coarser than the temporal resolution available within CPU) is not currently achievable [25].

Example 1. The *remote invoker* $\overline{a}\langle\tilde{v}\rangle;(\tilde{x}).P \rhd^t Q$ sends a message on a to a remote site and awaits a reply for t ticks on a private channel.

$$\overline{a}\langle\tilde{v}\rangle;(\tilde{x}).P \rhd^t Q \stackrel{\text{def}}{=} (\nu r)(\overline{a}\langle\tilde{v}r\rangle \mid \mathsf{timer}^t\langle r(\tilde{x}).P,\ (r(\tilde{x}).0 \mid Q)\rangle)$$

(Note that $r(\tilde{x}).0$ in the timeout continuation is necessary only for the typing to be introduced in §3.) While the remote invoker can be used locally, its intended use can be seen in the network $M \stackrel{\text{def}}{=} [!a(\tilde{y}r).\overline{r}\langle\tilde{b}\rangle] \mid [\overline{a}\langle\tilde{v}\rangle;(\tilde{x}).P \rhd^t Q]$ which has two reductions:

$$M \twoheadrightarrow [!a(\tilde{y}r).\overline{r}\langle\tilde{b}\rangle] \mid [P\{\tilde{b}/\tilde{x}\}] \qquad M \twoheadrightarrow [!a(\tilde{y}r).\overline{r}\langle\tilde{b}\rangle] \mid [Q \mid (\nu r)r(\tilde{x}).0]$$

Often such invocations are iterated: if the server does not reply within time t, we assume message-loss to have occurred and resend the invocation a bounded number of times n before giving up. This recovery mechanism can be defined inductively.

$$\overline{a}\langle\tilde{v}\rangle^n;(\tilde{x}).P \rhd^t Q \stackrel{\text{def}}{=} \overline{a}\langle\tilde{v}\rangle;(\tilde{x}).P \rhd^t Q_{n-1} \text{ with } Q_0 \stackrel{\text{def}}{=} Q,\ Q_{k+1} \stackrel{\text{def}}{=} \overline{a}\langle\tilde{v}\rangle;(\tilde{x}).P \rhd^t Q_k$$

For example, we can model a remote arithmetic operation (using an extended syntax with numbers and their operations).

$$[!a(yr).\overline{r}\langle y \times 3\rangle] \mid [(\nu b)(\overline{a}\langle 10\rangle^{100};(\tilde{x}).\overline{b}\langle x+7\rangle \rhd^5 \overline{b}\langle 0\rangle \mid b(z).R)] \tag{1}$$

This remote invoker tries to get 3×10 evaluated remotely and if there is no reply within 5 timesteps, it re-sends a request again. This routine is repeated until success or until 100 tries have been fruitless, when the invoker gives up and emits an exception (here in form of a value 0): this is in essence what real DS do to deal with message failure.

3 A Typing System for Distribution and Timer

Our typing system is based on the *linear/affine typing* [19] but without a sequentiality constraint (i.e. multiple threads of control are permitted) and augmented for timers and message-loss. This system is a proper generalisation of [5, 6, 19] and a simple instance of various linearity-based typing systems which ensure that interactions at certain channels will eventually succeed. [32] is the most accessible introduction to the key ideas behind linear typing. Our aim is for the extension to DS to preserve the ability to embed high-level sequential and concurrent multi-threaded languages [6, 19, 32].

3.1 Linear/Affine Typing System: Basic Idea

The linear/affine typing system introduced in [19] assigns types to free channels, which constrain the direction of information flow, the kind of information that flows on a channel, and how often a channel may be used: linear (exactly once), affine (at most once) or replicated (an unlimited number of times, including not at all).

This is achieved with the following basic constraints: first, each channel is classified as either linear, affine, replicated-linear or replicated-affine; orthogonally, each channel is either input or output. *Affinity* denotes possibly diverging behaviour in which a question is given an answer at most once. *Linearity* means terminating behaviour in which a question is always given an answer precisely once. Replicated-linear channels stand for servers that are guaranteed to return an answer, while replicated-affine channels are for servers that may exhibit divergence. Syntactically, typing enforces the following constraints:

1. For each replicated name there is a unique replicator with zero or more outputs
2. For each linear or affine name there is a unique input with a unique output
3. Linear channels have no circular dependency
4. Affine input channels cannot prefix linear channels
5. Replicators cannot prefix linear or affine channels

Under (1) above, $P_1 \stackrel{\text{def}}{=} !\,b.\overline{a} \mid !\,b.\overline{c}$ is not typable because b is associated with two replicators, but $P_2 \stackrel{\text{def}}{=} !\,b.\overline{a} \mid \overline{b} \mid !\,c.\overline{b}$ is typable since, while the output at b appears twice, there is just one replicator at b. As an example for (2), $P_3 \stackrel{\text{def}}{=} b.\overline{a} \mid c.\overline{a}$ is untypable with a being linear, as a appears twice as output. $P_4 \stackrel{\text{def}}{=} b.\overline{a} \mid c.\overline{b} \mid a.(\overline{c} \mid \overline{e})$ is typable since each channel appears at most once as input and output. By (3), we can ensure *termination* behaviour over linear channels. For example, $P_5 \stackrel{\text{def}}{=} !\,b.\overline{a} \mid !\,a.\overline{b}$ is untypable under this constraint: if we compose it with $\overline{a}$, then the computation does not terminate. P_4 is also untypable under (3). As given above, "linearity" means more than just termination: it indicates *a process always returns an answer if it is asked for*, which we call *liveness*. (4) ensures liveness at linear channels in the presence of termination: $P_6 = a.\overline{b}$ where a is affine and $\overline{b}$ is linear should be untypable since we do not know the input a surely happens. (5) is a standard constraint for affinity/linearity, by which $!a.\overline{b}$ where b is either linear or affine output is untypable.

Linear and Affine Types. We outline the formal definitions of types and the typing system, introducing the minimum definitions needed. *Action modes* (ranged over $p, p', \ldots$) [5, 32] prescribe different modes of interaction at each channel. The L-modes correspond to linear [32] while the A-modes to affine [5]. The last line denotes the grammar of types.

$\downarrow_{\mathrm{L}}$ Linear input $\uparrow_{\mathrm{L}}$ Linear output $\downarrow_{\mathrm{A}}$ Affine input $\uparrow_{\mathrm{A}}$ Affine output
$!_{\mathrm{L}}$ Linear server $?_{\mathrm{L}}$ Client request to $!_{\mathrm{L}}$ $!_{\mathrm{A}}$ Affine server $?_{\mathrm{A}}$ Client request to $!_{\mathrm{A}}$
$\tau \ ::= \ (\tilde{\tau})^p \mid \updownarrow$

The modes in the first and third columns are *input* while the second and forth are *output*. The input and output modes in each row are *dual* to each other, writing $\overline{p}$ for the dual of p. In types, $\tilde{\tau}$ is a vector of types. $\updownarrow$ indicates that a channel is no longer available for *further* composition with the outside; for example, if $x.0$ has a $\downarrow_{\mathrm{L}}$-mode and $\overline{x}$ has a $\uparrow_{\mathrm{L}}$-mode, then $x.0 \mid \overline{x}$ has $\updownarrow$-mode at x. The $\updownarrow$-mode at x indicates that the process $x.0 \mid \overline{x}$ cannot be composed with any process that has x as a free name. $\updownarrow$'s mode is $\updownarrow$. We assume a replicated affine input does not carry linear output (and dually). This condition ensures an invocation at linear replication will eventually terminate, firing

an associated linear output [19]. We write $\mathsf{md}(\tau)$ for the outermost mode of τ; τ^p also indicates that p is the (outermost) mode of τ. The *dual of* τ, written $\overline{\tau}$, is the result of dualising all action modes. Then the least commutative partial operation, $\odot$, which controls the composition of channels is defined as:

$$\text{(a)}\ \ \tau\odot\tau=\tau \text{ and } \tau\odot\overline{\tau}=\overline{\tau} \text{ with } \mathsf{md}(\tau)=? \qquad \text{(b)}\ \ \tau\odot\overline{\tau}=\updownarrow \text{ with } \mathsf{md}(\tau)=\uparrow$$

(a) and (b) ensure the two constraints (1) and (2) in the list of § 3.1.
An *action type*, denoted $A,B,\ldots$, is a finite directed graph with nodes of the form $x:\tau$, such that no names occur twice; and *causality edge* $x:\tau\to y:\tau'$ is of the form: from a linear input $\downarrow_{\text{L}}$ to a linear output $\uparrow_{\text{L}}$; or from a linear replication $!_{\text{L}}$ to a linear client $?_{\text{L}}$. $\overline{A}$ dualises all types in A. We write $A(x)$ for the channel type assigned to x occurring in A. The partial operator $A\odot B$ is defined iff channel types with common names compose and the adjoined graph does not have a cycle. This avoids divergence on linear channels. For example, $a:\tau_1\to b:\tau_2$ and $b:\overline{\tau_2}\to a:\overline{\tau_1}$ are not composable, hence a process such as $P_5\stackrel{\text{def}}{=}!a.\overline{b}\mid !b.\overline{a}$ is untypable. We write $A_1\asymp A_2$ when such composition is possible, while the result of composition is written $A_1\odot A_2$ (see [7] for details). By this operation, we can guarantee the condition (3) in § 3.1 for the linear channels. Non-circular causality between linear channels guarantees liveness at linear output channels, resulting in the distributed liveness theorem below.

3.2 Typing Systems for Message-loss and Timers

The typing judgements for processes forms $P\vdash A$ (a process P has an action type A) and networks forms $N\vdash A$ (a network N has an action type A). We list the selected rules below.

$$\textsc{Loc}\ \frac{P\vdash A \quad A \text{ distributable}}{[P]\vdash A} \qquad \textsc{Timer}\ \frac{x(\tilde{y}).P\vdash A \quad Q\vdash A}{\mathsf{timer}^t\langle x(\tilde{y}).P,\ Q\rangle\vdash A} \qquad \textsc{Timer}_c\ \frac{x(\tilde{y}).P\vdash (x:(\tilde{\tau})^{\downarrow_{\text{L}}}\to A),B \quad Q\vdash x:(\tilde{\tau})^{\downarrow_{\text{A}}},A,B}{\mathsf{timer}^t\langle x(\tilde{y}).P,\ Q\rangle\vdash x:(\tilde{\tau})^{\downarrow_{\text{A}}},A,B}$$

It is worth pointing out that the types and the rules for non-distributed processes are identical with those in [19].

Typing Networks. The key difficulty in typing networks is that messages can get lost, but we must ensure that linear inputs and outputs do not get lost, i.e. linear names must never be sent to remote sites. We achieve this by imposing that free names observable at the network level cannot be linear input or linear output, and cannot carry such names recursively: we say type τ is *distributable* if τ does not contain either linear input or linear output in its subexpressions. For example, neither $()^{\uparrow_{\text{L}}}$ nor $(()^{\downarrow_{\text{L}}})^{?_{\text{L}}}$ is distributable. We say A is *distributable* if $A(x)$ is distributable for all x. Then by [Loc], we transfer a process P which has a distributable type to the network level as $[P]$. Note that linear names can also exist hidden by a restriction or bound by prefixes. But because by construction distributable types cannot carry linear names, no hidden linear name can escape to the network, thus preserving linearity.

Typing Timers. Timers can be seen as a form of mixed choice (ignoring timing, $\mathsf{timer}^t\langle x(\tilde{v}).P,\ Q\rangle$ can be translated as $x(\tilde{v}).P + \tau.Q$) where one branch is chosen by the environment while the other can be triggered internally. Hence, if both branches can be given the same type A, then the timer also has type A, thus explaining [TIMER].

While natural, this rule is not expressive enough: the most important use of timers, detection of message-loss is not typable. To see why, consider

$$[!x(vr).P] \mid [(\nu r)(\overline{x}\langle ar\rangle \mid \mathsf{timer}^t\langle r(\tilde{v}).Q,\ R\rangle|...)]$$

a simplified version of the remote invoker from Example 1 where the timer triggers some recovery action in R if the remote action does not reply in time. For this process to be typable, x must not carry linear names, hence r must be affine. (4) in §3.1 does not allow to suppress linear names under an affine input, hence $r(\tilde{v}).Q$ would not have any linear (liveness) behaviour. Timers are used to make liveness guarantees so that a computation does not hang forever, even if messages get lost or servers do not reply.

To overcome this lack of expressivity, we introduce the second rule [TIMER_c] that allows to suppress linear names under an affine output without breaking the typing system. We can use timers to ensure distributed liveness, i.e. "the result of an invocation will always eventually be returned", but refining the concept of liveness to "*either* a result *or* an indication of error will be returned". Formally, [TIMER_c] first types $x(\tilde{v}).P$ assuming x is a linear input $x:(\tilde{\tau})^{\downarrow_L}$, hence permitting to suppress linear outputs A (here $x:(\tilde{\tau})^{\downarrow_L} \to A$ is a type which is obtained by adding edges from $x:(\tilde{\tau})^{\downarrow_L}$ to A). Notice that we *cannot* apply the same method for Q since after t-ticks, when Q is launched, it may wait forever on x (see Example 2 below). Hence, the second premise assumes that x is an affine input, and no linear names can depend causally on x, linear names must be available without external interaction (and are hence guaranteed to fire eventually). This means Q will be of the form $(\nu\tilde{z})(R|x(\tilde{v}).Q'), x \notin \tilde{z}$, with all the free linear names being in R. The conclusion of [TIMER_c] then gives the timer the type of its timeout continuation. Note that the typing rules for timers do not depend on the concrete details of the timing model (e.g. discrete or continuous), but rather apply to all.

Theorem 1. *If $P \vdash A$ and $P \to Q$ then $Q \vdash A$, and likewise for networks.*

Next we formulate a distributed liveness property which states that linear local outputs always fire (under the usual implicit fairness assumptions that each process that is not deadlocked or terminated will eventually be scheduled). Write $P \Downarrow_a$ if there exist $\tilde{b}, R, \tilde{v}$ s.t. $P \twoheadrightarrow (\nu\tilde{b})(R \mid \overline{a}\langle\tilde{v}\rangle)$ with $a \notin \{\tilde{b}\}$.

Definition 1. *1. (local liveness) We say A is* closed *if* $\mathsf{md}(A) \in \{!_A, !_L, \updownarrow\}$. *Suppose $P \vdash A, a:\tau$ with A closed and* $\mathsf{md}(\tau) = \uparrow_L$. *Then we say P satisfies* liveness at *a if whenever $P \twoheadrightarrow P'$, $P' \Downarrow_a$.*

2. (distributed liveness) We say network N satisfies distributed liveness*, if, for all P such that $N \equiv (\nu\tilde{a})([P \mid Q] \mid M)$ which is derived from $P \vdash A, a:\tau$ with A closed and* $\mathsf{md}(\tau) = \uparrow_L$*, P has a local liveness at a.*

The following is proved from Theorem 1 as stated in [19].

Theorem 2. *For all N such that $N \vdash A$, N satisfies distributed liveness.*

Example 2. Suppose that a,b are linear names and c,u are affine.

1. $a.\overline{b}$, $b.\overline{c}$ and $c.0 \mid \overline{b}$ are typable as $a.\overline{b} \vdash a:()^{\downarrow_{\mathrm{L}}} \to b:()^{\uparrow_{\mathrm{L}}}$, $b.\overline{c} \vdash b:()^{\downarrow_{\mathrm{L}}}, c:()^{\uparrow_{\mathrm{A}}}$ and $c.0 \mid \overline{b} \vdash c:()^{\downarrow_{\mathrm{A}}}, b:()^{\uparrow_{\mathrm{L}}}$, respectively. But $c.\overline{b}$ is not typable.
2. Let $\Omega_u \stackrel{\text{def}}{=} (\nu y)(\mathsf{fw}_{uy}|\mathsf{fw}_{yu})$ with $\mathsf{fw}_{xy} \stackrel{\text{def}}{=} !x(z).\overline{y}\langle z\rangle$. fw_{xy} is called a *forwarder*, while Ω_u is an *omega* which diverges with a message as: $\Omega_u|\overline{u}\langle e\rangle \to \Omega_u|\overline{u}\langle e\rangle \to \ldots$. It is typed as $\Omega_u \vdash u : ()^{\uparrow_{\mathrm{A}}})^{?_{\mathrm{A}}}$.
3. $\mathsf{timer}^5\langle a.\overline{b},\ a.\overline{b}\rangle$, $\mathsf{timer}^5\langle c.0,\ c.0\rangle$ and $P \stackrel{\text{def}}{=} \mathsf{timer}^5\langle c.\overline{b},\ (c.0 \mid \overline{b})\rangle$ are typable but $Q \stackrel{\text{def}}{=} \mathsf{timer}^5\langle c.\overline{b},\ c.\overline{b}\rangle$ is not typable. The first two are by [TIMER], and P is by [TIMER_c]. We shall see how the liveness at b is guaranteed in P by dividing into the two cases. The first case is when c is returned within the time limit 5 because the call between the first and second processes terminates.

$$!e(y).\overline{y} \mid (\nu x)(\overline{e}\langle x\rangle \mid x.\overline{c}) \mid P \to\!\!\to !e(y).\overline{y} \mid \overline{c} \mid \phi^2(P) \to\!\!\to !e(y).\overline{y} \mid \overline{b}$$

The second case is a time-out due to non-termination of the call.

$$\Omega_u \mid (\nu x)(\overline{e}\langle x\rangle \mid x.\overline{c}) \mid P \to^5 \Omega_u \mid (\nu x)(\overline{e}\langle x\rangle \mid x.\overline{c}) \mid \phi^5(P) \equiv \Omega_u \mid (\nu x)(\overline{e}\langle x\rangle \mid x.\overline{c}) \mid c.0 \mid \overline{b}$$

This shows if we replace P by Q, we cannot guarantee the liveness at b.
4. We show typing of the remote invoker from (1) in Example 1. First we type the server as: $[!a(yr).\overline{r}\langle y\times 3\rangle] \vdash a:\tau$ with $\tau = (\mathtt{nat}(\mathtt{nat})^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$ Then the remote invoker has type $\overline{a}\langle 10\rangle^{100}; (\tilde{x}).\overline{b}\langle x+7\rangle \rhd^5 \overline{b}\langle 0\rangle \vdash a:\overline{\tau}, b:(\mathtt{nat})^{\uparrow_{\mathrm{L}}}$. Here we assume `nat` is the type of natural numbers. Note that b has a linear output, hence ensuring liveness. By hiding b, the client location can also have a distributable type.

4 Probabilistic Distributed Timed Processes and Bisimulation

We refine our model by replacing non-deterministic message-loss with a probability $r \in [0,1]$ which structures all message-loss globally and independently (i.e. the message-loss probability does not change throughout the course of a computation, and the events that two different messages get lost are independent). More general forms of probability like having different message-loss probabilities for different channels, or having probabilities change over time, are easily expressible in our framework, but have been omitted for brevity. Our approach to adding probabilities, inspired by [11, 12], is classical in that we use probabilistic automata [28].

Definition 2. Let A be a discrete set (i.e. finite or countably infinite). A *formal quantity* over A is a relation $\mathcal{R} \subseteq A \times \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the non-negative real numbers. The *support* of $\mathcal{R}$ is $\mathsf{support}(\mathcal{R}) \stackrel{\text{def}}{=} \{a \in A \mid (a,r) \in \mathcal{R}, r > 0\}$. We often omit A. We can add formal quantities, and multiply them with scalars: if $r \in \mathbb{R}^+$ then $r \cdot \mathcal{R} \stackrel{\text{def}}{=} \{(a, r \cdot a) \mid (a,r) \in \mathcal{R}\}$. Likewise $\Sigma_{i\in I}\mathcal{R}_i \stackrel{\text{def}}{=} \bigcup_{i\in I}\mathcal{R}_i$. We often write $\Sigma_{(a,r)\in\mathcal{R}} r\cdot\overline{a}$ for $\mathcal{R}$. If $\mathcal{R}$ is a function, we call $\mathcal{R}$ a *quantity* over A. An important example of a quantity over A is $\overline{a} \stackrel{\text{def}}{=} \{(a,1)\} \cup \{(a',0) \mid a' \neq a\}$. We call $\overline{a}$ the *Dirac-distribution for* a (in A), and often write simple a for $\overline{a}$. Many formal quantities $\mathcal{R}$ can be *flattened* into a quantity $\flat(\mathcal{R})$ which is the map $a \mapsto \Sigma_{(a,r)\in\mathcal{R}} r$, assuming that the sum in this expression converges

for every a. Hence $\flat(\cdot)$ is a partial operation. Scalar multiplication of quantities is that of formal quantities, whereas summation on quantities is given by $\Sigma_{i\in I} f_i \stackrel{\text{def}}{=} \flat(\bigcup_{i\in I} f_i)$, which is only defined where $\flat(\cdot)$ is defined.

A *subprobability distribution*, ranged over by $\Delta, ...$, over a discrete set A is a quantity over A such $\Delta : A \to [0,1]$ such that $\sum_{a\in A} \Delta(a) \leq 1$. We often call subprobability distributions just distributions. A subprobability distribution Δ is a *probability distribution* if $\sum_{a\in A} \Delta(a) = 1$. We write $\Delta_1|\Delta_2$ for the distribution Δ such that $\Delta(M) = \Delta_1(N) \cdot \Delta_2(L)$, provided that $M = N|L$, and $\Delta(M) = 0$ otherwise. $\Delta|M$ is short for $\Delta|\overline{M}$. Similarly, $(\nu x)\Delta$ is the distribution Δ' such that $\Delta'(N)$ is $\Delta(M)$ if $N = (\nu x)M$ and 0 otherwise. If $\mathsf{support}(\Delta) = \{M_1, ..., M_n\}$ and $\Delta(M_i) = p_i$, we also write $(M_1 : p_1, ..., M_n : p_n)$ or just $\tilde{M} : \tilde{p}$ for Δ. We write $()$ for the subprobability distribution that is 0 everywhere, and often do not specify 0 probabilities. We call a formal quantity $\mathcal{R}$ such that $\flat(\mathcal{R})$ is a subprobability distribution a *formal subprobability distribution*. We write $\Delta_1 \equiv \Delta_2$ provided $\Delta_i = (M_1^i : p_1, ..., M_n^i : p_n)$ for $i = 1,2$ and for all j: $M_j^1 \equiv M_j^2$. If $\mathcal{R}$ is a relation on networks, its *lifting* to subprobability distributions is: $\flat(\tilde{M} : \tilde{r})\ \mathcal{R}\ \flat(\tilde{N} : \tilde{r})$ iff for all i: $M_i\ \mathcal{R}\ N_i$. By $\sim$ we denote the usual *probabilistic strong bisimilarity* and also its lifiting.

Reductions and Transitions. We define probabilistic reductions on networks by the rules below. Reductions are of the form $M \to \Delta$ where Δ is a probability distribution over networks. $\Delta(N)$ expresses the probability that M can evolve into N in one step. Probabilistic choices are made only about loosing messages in remote communication. All other choices relating are resolved non-deterministically.

$$\textsc{Intra}\ \frac{P \to Q}{[P] \to_1 [Q]} \quad \textsc{Par}\ \frac{M \to \Delta}{M|N \to \Delta|N} \quad \textsc{Res}\ \frac{M \to \Delta}{(\nu a)M \to (\nu a)\Delta} \quad \textsc{Cong}\ \frac{M \equiv M' \to \Delta' \equiv \Delta}{M \to \Delta}$$

$$\textsc{Com}\ \frac{}{[P|\overline{x}\langle\tilde{y}\rangle] \mid [Q|x(\tilde{v}).R] \to \left\{ \begin{array}{ll} [\phi(P)]|[Q|x(\tilde{v}).R] & : r \\ [\phi(P)]|[\phi(Q)|R\{\tilde{y}/\tilde{v}\}] & : 1-r \end{array} \right\}}$$

$$\textsc{Rep}\ \frac{}{[P|\overline{x}\langle\tilde{y}\rangle] \mid [Q|!x(\tilde{v}).R] \to \left\{ \begin{array}{ll} [\phi(P)]|[Q|!x(\tilde{v}).R] & : r \\ [\phi(P)]|[\phi(Q)|R\{\tilde{y}/\tilde{v}\}|!x(\tilde{v}).R] & : 1-r \end{array} \right\}}$$

Again, the corresponding reductions for replicated input and timed input are omitted. Reductions on processes are unchanged. We write $M \to_1 N$ to mean $M \to \overline{N}$. *Network transitions* are of the form $P \xrightarrow{l} \Delta$ where Δ is a probability distribution, and l is a *label* generated as usual by $l ::= \tau \mid x(\tilde{y}) \mid \overline{x}\langle(\nu\tilde{y})\tilde{z}\rangle$. The process transitions are standard [4], non-probabilistic and listed in [7]. We write $M \xrightarrow{l}_1 N$ for $M \xrightarrow{l} \overline{N}$. The transition system is given by the rules:

$$\frac{P \xrightarrow{l} Q}{[P] \xrightarrow{l}_1 [Q]}\ \textsc{Loc} \quad \frac{M \xrightarrow{l} \Delta \quad \mathsf{fn}(N) \cap \mathsf{bn}(l) = \emptyset}{M|N \xrightarrow{l} \Delta|N}\ \textsc{Par} \quad \frac{M \xrightarrow{l} \Delta \quad x \notin \mathsf{n}(l)}{(\nu x)M \xrightarrow{l} (\nu x)\Delta}\ \textsc{Res}$$

$$\frac{M \xrightarrow{\overline{x}\langle(\nu\tilde{y})\tilde{z}\rangle} \Delta_1 \quad N \xrightarrow{x(\tilde{z})} \Delta_2 \quad \tilde{y}\cap \mathsf{fn}(N)=\emptyset}{M|N \xrightarrow{\tau} r\cdot(\nu\tilde{y})(\Delta_1|\overline{N}) + (1-r)(\nu\tilde{y})(\Delta_1|\Delta_2)}\ \textsc{Com}$$

$$\frac{M \xrightarrow{\overline{x}\langle(\nu\tilde{y})\tilde{z}\rangle} \Delta \quad a\in\tilde{z}\setminus(\tilde{y}\cup\{x\})}{(\nu a)M \xrightarrow{\overline{x}\langle(\nu a\tilde{y})\tilde{z}\rangle} \Delta}\ \textsc{Open} \qquad \frac{M \equiv_\alpha M' \quad M' \xrightarrow{l} \Delta}{M \xrightarrow{l} \Delta}\ \textsc{Alpha}$$

It is easy to show that $M \to \Delta$ iff $M' \equiv M \xrightarrow{\tau} \Delta' \equiv \Delta$. Clearly, whenever $M \xrightarrow{l} \Delta$ then $|\mathsf{support}(\Delta)| < 3$ and Δ is a probability distribution. Next we prepare for defining our notion of bisimilarity. For this we need to abstract from τ-transitions with transitions $M \xrightarrow{\hat{l}} \Delta$ and $\Delta \stackrel{\hat{l}}{\Longrightarrow} \Delta'$ where Δ and Δ' are *sub*probability distributions. The need for subprobability distributions is explained below.

Definition 3. *The* auxiliary transitions $\stackrel{l}{\rightsquigarrow}$ *are defined by: (1)* $M \xrightarrow{l} \Delta$ *implies* $M \stackrel{l}{\rightsquigarrow} \Delta$ *and (2)* $M \stackrel{l}{\rightsquigarrow} ()$. *Now* weak transitions $M \xrightarrow{\hat{l}} \Delta$ *are defined if (1)* $M \stackrel{l}{\rightsquigarrow} \Delta$ *or (2)* $l = \tau$ *and* $\Delta = \overline{M}$. *This is extended to distributions as follows. We write* $\Delta \xrightarrow{\hat{l}} \Delta'$ *provided: (1)* $\Delta = \Sigma_{i\in I} p_i \cdot \overline{M_i}$, *(2) for all* $i \in I$ *with* $p_i > 0$: $M_i \xrightarrow{\hat{l}} \Delta_i$, *(3)* $\Delta' = \Sigma_{i\in I} p_i \cdot \Delta_i$. *Now we set* $\Delta \stackrel{\hat{\tau}}{\Longrightarrow} \Delta'$ *whenever* $\Delta(\xrightarrow{\hat{\tau}})^*\Delta'$. *We define* $\Delta \stackrel{\hat{l}}{\Longrightarrow} \Delta'$ *when* $\Delta \stackrel{\hat{\tau}}{\Longrightarrow} \xrightarrow{l} \stackrel{\hat{\tau}}{\Longrightarrow} \Delta'$. *Similarly, for* $l \neq \tau$, *we also write* $\Delta \stackrel{\hat{l}}{\Longrightarrow} \Delta'$ *for* $\Delta \stackrel{l}{\Longrightarrow} \Delta'$. $M \stackrel{\hat{l}}{\Longrightarrow} \Delta$ *stands for* $\overline{M} \stackrel{\hat{l}}{\Longrightarrow} \Delta$.

Example 3. Let $M \stackrel{\text{def}}{=} [\overline{x}\langle y\rangle]|[x(v).\overline{v}], \Delta = ([0]|[x(v).\overline{v}] : r, [0]|[\overline{y}] : 1-r), \Delta' \stackrel{\text{def}}{=} ([0]|[x(v).\overline{v}] : 0, [0]|[0] : 1-r)$. Note that Δ' is a subprobability distribution. Then M's (auxiliary/weak) transitions and reductions include:

$$M \to \Delta \qquad M \stackrel{\tau}{\rightsquigarrow} \Delta \qquad M \stackrel{\hat{\tau}}{\rightsquigarrow} \overline{M} \qquad M \stackrel{\hat{\tau}}{\rightsquigarrow} \Delta \qquad M \stackrel{\hat{\tilde{y}}}{\Longrightarrow} \Delta'$$

Weak transitions have a proper subprobability distribution as target. It is inferred as $M \stackrel{\hat{\tau}}{\rightsquigarrow} \Delta \stackrel{\hat{\tilde{y}}}{\Longrightarrow} \Delta'$. Then $\Delta \stackrel{\hat{\tilde{y}}}{\Longrightarrow} \Delta'$ is inferred as the combination of $[0]|[\overline{y}] \stackrel{\hat{\tilde{y}}}{\rightsquigarrow} \overline{[0]|[0]}$ and $[0]|[x(v).\overline{v}] \stackrel{\hat{\tilde{y}}}{\rightsquigarrow} ()$. Note that this last transition uses $()$ only to specify that $[0]|[x(v).\overline{v}]$ does not in fact have a transition labelled $\overline{y}$. Presenting the absence of a transition this way simplifies defining bisimulations.

4.1 Approximate Bisimulation

We introduce probabilistic approximate bisimulations. They are useful in the study of DS where we often want to give erroneous behaviour a different status from normal operation. To explain the issue, consider:

$$M \stackrel{\text{def}}{=} (\nu x)([\overline{x}\langle y\rangle] \mid [x(v).P]) \qquad\qquad N \stackrel{\text{def}}{=} [P\{y/v\}]$$

Assuming $x \notin \mathsf{fn}(P)$ and no message-loss, we expect $M \approx N$, where $\approx$ is a chosen notion of weak equivalence. However, if $r > 0$ is the global message-loss probability, such an equality can no longer hold, *irrespective of how negligible r may be*. This often stands

in the way of reasoning, because we want to abstract away from negligible probabilities. Now consider

$$M^n \stackrel{\text{def}}{=} (\nu x)([\Pi_{i=1}^n \overline{x}\langle y\rangle] \mid [x(v).P]) \qquad (n>0)$$

In general, with $x \notin \mathsf{fn}(P)$, M^n can only be distinguished from N if that all n outputs $\overline{x}\langle y\rangle$ get lost. The probability of this happening is r^n. We would like to have a notion of equality $\approx^\varepsilon$ such that $M^n \approx^{r^n} N$, i.e. $\varepsilon \in [0,1]$ gives a quantitative bound on how much the processes compared by $\approx^\varepsilon$ are allowed to mismatch. Approximate notions of equality are of prime importance in cryptography where one usually demonstrates the safety of a cryptographic protocol by showing that the probability that it can be broken vanishes exponentially quickly in a chosen system parameter (like password length). Approximate bisimulations are intended to generalise this style of verification and connect it with standard methods in concurrency theory.

Definition 4. *An* approximate bisimulation *is a family* $\{\mathcal{R}^\varepsilon\}_{\varepsilon\in B}$ *where* $B \subseteq [0,1]$ *of relations on networks such that* $M\,\mathcal{R}^\varepsilon\,N$ *implies: whenever* $M \xrightarrow{l} \Delta$ *then also* $N \stackrel{\hat{l}}{\Longrightarrow} \Delta'$ *for some* Δ' *with* $\Delta\,\mathcal{R}^\varepsilon\,\Delta'$*, and vice versa. Here* $\Delta\,\mathcal{R}^\varepsilon\,\Delta'$ *means that we can find two formal subprobability distributions* $\tilde{M} : \tilde{r}, \tilde{M}' : \tilde{s}$ *and* $\tilde{N} : \tilde{r}, \tilde{N}' : \tilde{t}$ *such that: (1)* $\Delta = \flat(\tilde{M} : \tilde{r}, \tilde{M}' : \tilde{s})$*; (2)* $\Delta' = \flat(\tilde{N} : \tilde{r}, \tilde{N}' : \tilde{t})$*; (3)* $1 - \Sigma_i r_i \leq \varepsilon$*; (4) for all i:* $M_i\,\mathcal{R}^{\varepsilon'_i}\,N_i$ *for some* $\varepsilon'_i \leq \frac{\varepsilon}{r_i}$*. We call* ε *the* discount *of* $\mathcal{R}^\varepsilon$ *and* $\mathcal{R}^\varepsilon$ *on distributions the* ε-lift. Strong approx. bisimulations *are defined similarly.*

This definition refines [15] by weighting discounts through clause (4). Similar techniques can be used to produce approx. forms of other equivalence (e.g. traces). Without refinement, one cannot prove, e.g. $M^n \approx^{r^n} N$, only the weaker $M^n \approx^r N$.

Lemma 1. *1.* $\{\equiv^0\}$ *is an approx. bisimulation.*
2. $\{\mathcal{R}^1\}$ *is an approx. bisimulation for every* $\mathcal{R}$*.*
3. *If* $\{\mathcal{R}^{\varepsilon_i}\}_{i\in I}$ *is an approx. bisimulation and* $\varepsilon_i \leq \varepsilon'_i$ *for all i then* $\{\mathcal{R}^{\varepsilon'_i}\}_{i\in I}$ *is an approx. bisimulation.*
4. *If* $\{\mathcal{R}_j^\varepsilon\}_{\varepsilon\in B_j}$ *is an approx. bisimulation for each j then so is* $\{\mathcal{S}^\varepsilon \mid \varepsilon \in \bigcup_j B_j\}$ *where* $\mathcal{S}^\varepsilon \stackrel{def}{=} \{(M,N) \mid \exists j.(M,N) \in \mathcal{R}_j^\varepsilon\}$*.*
5. $\{\mathcal{R}^0\}$ *is an approx. bisimulation, where* $\mathcal{R} \stackrel{def}{=} \{([P],[Q]) \mid P \approx Q\}$ *with* $\approx$ *being the usual bisimulation on processes [4, 19].*

Lemma 1.5 transfers the chosen equivalence on processes to networks.

Definition 5. *M and N are* ε-bisimilar *if* $M\,\mathcal{R}^\varepsilon\,N$ *for some approx. bisimulation* $\{\mathcal{R}^{\varepsilon_i}\}_i$*. In this case we also write* $M \approx^\varepsilon N$*.*

To aid reasoning about approx. bisimulations one can use up-to techniques.

Definition 6. $\{\mathcal{R}^{\varepsilon_i}\}_i$ *is an approx. bisimulation* up to $\sim$ (resp. up to restriction) *if* $M\,\mathcal{R}^\varepsilon\,N$ *implies that whenever* $M \xrightarrow{l} \Delta$ *there is* $N \stackrel{\hat{l}}{\Longrightarrow} \Delta'$ *such that* $\Delta\ (\sim \circ \mathcal{R}^\varepsilon \circ \sim)\ \Delta'$ *(resp.* $\Delta_0\ \mathcal{R}^\varepsilon\ \Delta'_0$ *with* $\Delta = (\nu\tilde{x})\Delta_0, \Delta' = (\nu\tilde{x})\Delta'$*), and vice versa.*

The main result follows.

Theorem 3 (congruency). $\approx^{\varepsilon}$ *is a congruence.*

The following theorem offers compositional and tractable verification tools for approx. bisimulations.

Theorem 4. *1. If* $\{\mathcal{R}^{\varepsilon}\}_{\varepsilon\in B}$ *is an* ε*-bisimulation up to* $\sim$ *or up to restriction then* $\mathcal{R}^{\varepsilon} \subseteq \approx^{\varepsilon}$ *for all* $\varepsilon \in B$.
2. If $M \approx^{r} N$ *and* $N \approx^{s} L$ *then* $M \approx^{\min(r+s,1)} L$.
3. With r being the global message-loss probability, $(\nu\tilde{x})([P|Q]) \approx^{r} (\nu\tilde{x})([P]|[Q])$ *for all* $\tilde{x}$ *and all timer-free* P, Q.

The restriction to timer-free processes in (3) is vital because the relative timing between P and Q is very different if these processes run in a single location rather than in two.

We can now motivate subprobability distributions and the shape of auxiliary transitions: consider $(\nu x)M$ with M as in Example 3. If we want to show that $(\nu x)M \approx^{r} [\bar{y}]$, we need to match $[\bar{y}] \xrightarrow{\bar{y}} [0]$. But $(\nu x)M$ can do an output on y only if the internal message on x is not lost. This is expressed by the subprobability appearing in the matching weak transition $(\nu x)M \xrightarrow{\hat{\tau}} (\nu x)\Delta \xRightarrow{\hat{\bar{y}}} (\nu x)\Delta'$. Without this definition of weak transitions, the definition of approx. bisimulation would be more complicated.

Typed Approximate Bisimulations. We now show how types lead to more efficient approximate reasoning. The key point [5] is that some transitions cannot be observed in a typed setting because no well-typed observer can interact with it. E.g. $P \stackrel{\text{def}}{=} \bar{x}\langle v\rangle | x(y).Q$ has transitions at x, but if $P \vdash x : \updownarrow, A$ then x is not available for further composition. Hence we need not consider transitions at x when comparing P with another process of the same type. Similarly from $\bar{x}\langle v\rangle | !x(y).Q$, we cannot observe the output at x since it should be consumed in the unique replicator. This intuition is formalised as follows: let A be an action type and l an action. The predicate $A \vdash l$ is defined if (1) $l = \bar{x}\langle(\nu\tilde{y})\tilde{z}\rangle$ implies $\mathsf{md}(A(x)) \in \{\uparrow_{\mathrm{L}}, \uparrow_{\mathrm{A}}, ?_{\mathrm{L}}, ?_{\mathrm{A}}\}$; or (2) $l = x(\tilde{y})$ implies $A(x) \in \{\downarrow_{\mathrm{L}}, \downarrow_{\mathrm{A}}, !_{\mathrm{L}}, !_{\mathrm{A}}\}$; or (3) $l = \tau$. A (sub)probability distribution has *type* A if $\Delta(M) > 0$ implies $M \vdash A$. *Typed labelled transitions* $P \xrightarrow{l} Q \vdash A$ are defined if $P \vdash A$, $A \vdash l$ and $P \xrightarrow{l} Q$. For networks we have $M \xrightarrow{l} \Delta \vdash A$ provided $M \vdash A$, $A \vdash l$ and $M \xrightarrow{l} \Delta$.

Definition 7. *A* typed approximate bisimulation *is a family* $\{\mathcal{R}^{\varepsilon}\}_{\varepsilon\in B}$ *where* $B \subseteq [0,1]$ *of binary relations on typed networks, relating only terms of the same type, such that* $M \mathcal{R}^{\varepsilon} N$ *implies that whenever* $M \xrightarrow{l} \Delta \vdash A$ *then also* $N \xRightarrow{\hat{l}} \Delta'$ *for some* Δ' *with* $\Delta \mathcal{R}^{\varepsilon} \Delta'$*, and vice versa. The definition of* $\Delta \mathcal{R}^{\varepsilon} \Delta'$ *is similarly adapted.*

We note that Theorems 3 and 4 also hold for the typed bisimilarity.

4.2 Examples of Distributed Protocols

Verifying an RPC Protocol with Message Recovery. We show how to use approx. bisimulations to reason about remote procedure calls (RPCs), an important distributed algorithm that uses timers to increase the reliability of remote communication. In this subsection, we assume that [IDLE] is only used when no other rules apply. This standard

assumption is called *maximal progress* in the literature and prevents the timer aborting by itself even when communication is possible. Consider:

$$P \stackrel{\text{def}}{=} !x(mv).\overline{v}\langle m+2\rangle \qquad\qquad Q \stackrel{\text{def}}{=} (\nu y)(\overline{x}\langle 5y\rangle | y(m).\overline{a}\langle m+1\rangle)$$

We type: $P|Q \vdash A$ with $A \stackrel{\text{def}}{=} x : (\mathtt{nat}(\mathtt{nat})^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}, a : (\mathtt{nat})^{\uparrow_{\mathrm{L}}}$. It is easy to show that $P|Q \approx P|\overline{a}\langle 8\rangle$ (where $\approx$ is the typed bisimilarity on processes [19]), hence by Theorem 4 and Lemma 1, we have:

$$[P|\overline{a}\langle 8\rangle|a(z).R] \quad \approx^0 \quad [P|Q|a(z).R] \quad \approx^r \quad [P] \mid [Q|a(z).R]$$

Theorem 4 is thus useful because it gives a straightforward upper bound on how different (untimed) processes can be when distributed. But since it does so without assumptions on P and Q, the bounds are weak. To improve on the right bound, we can use the remote invoker from Example 1:

$$Q^n \stackrel{\text{def}}{=} \overline{x}\langle 5\rangle^n; (m).\overline{a}\langle m+1\rangle \rhd^t \overline{a}\langle 0\rangle$$

The timer is used to amplify the reliability of communication over an unreliable channel x. It uses a hidden name, generated at the client, to get the acknowledgement. The timer re-sends the invocation repeatedly, if the acknowledgement is not received within time t. The receiver $a(z).R$ at the server side knows whether it was correctly delivered a datum or whether a timeout happened, because $\overline{a}\langle 0\rangle$ signals n-ary timeout, which can be taken as indicating failure. Assuming that r is the global message-loss probability, we want to establish that

$$[P] \mid [Q^n \mid a(z).R] \quad \approx^{r^n} \quad [P] \mid [\overline{a}\langle 8\rangle \mid a(z).R] \tag{2}$$

provided $t > 1$ and $x \notin \mathsf{fn}(R)$. Note that a in (2) has type $x : \updownarrow$. Establishing (2) is straightforward by induction. The base case is trivial. For the inductive step note that $[P]|[\overline{a}\langle 8\rangle \mid a(z).R] \stackrel{\hat{\tau}}{\longrightarrow}_1 [P]|[0 \mid R\{8/z\}]$. This can be matched in several ways: either the first invocation attempt already succeeds with probability $1-r$, or the first fails but the second succeeds (probability $r\cdot(1-r)$) and so on, giving the weak transition $[P]|[Q^n \mid a(z).R] \stackrel{\hat{\tau}}{\Longrightarrow} ([P]|[0 \mid R\{8/z\}] : 1-r^n)$. The other remaining transitions can be matched exactly since linear liveness at a guarantees that the interaction with $a(z).R$ will always happen. Due to typing, we do not need to consider output transitions of Q^n because no typable observer can interact with them. This way, typing reduces the number of transitions to be matched in approx. bisimulations. Overall we easily establish (2) which means that n-ary remote invokers are an effective error recovery technique, reducing the possibility of message failures exponentially quickly in the parameter n.

Next we show that conversely, the more a DS relies on networked communication, the more error-prone it becomes. To this end, define

$$M^{n+1} \stackrel{\text{def}}{=} (\nu \tilde{x})([\overline{x_1}\langle v\rangle] \mid \Pi_{i=1}^{n-1}[x_i(y).\overline{x_{i+1}}\langle y\rangle] \mid [x_n(y).\overline{y}]) \qquad M^0 \stackrel{\text{def}}{=} [\overline{v}]$$

As in the previous examples one can then show that $M^n \approx^{1-(1-r)^n} [\overline{v}]$. This says that the chance of M^n behaving like $[\overline{v}]$ diminishes exponentially quickly.

Leases. Another important example for fault-tolerant DS are *leases* [16] which allow clients to access a remote service for a limited amount of time. Once that time has expired without the client renewing the lease, access is denied to the client, and the server holding the service is free to close it, to make it available to others, or to withdraw it completely. In our setting, leases are naturally expressed using *typed, timed forwarders*. A timed forwarder $\mathsf{tfw}_{xy}^{t\cdot P}$ does the same, but only for t units of time, and after expiry of the lease executes the 'clean-up process' P.

$$\mathsf{tfw}_{xy}^{0\cdot P} \stackrel{\text{def}}{=} P|x(\tilde{v}).0 \qquad \mathsf{tfw}_{xy}^{t+1\cdot P} \stackrel{\text{def}}{=} \mathsf{timer}^1\langle x(\tilde{v}).(\overline{y}\langle v\rangle \mid \mathsf{tfw}_{xy}^{t\cdot 0}),\ \mathsf{tfw}_{xy}^{t\cdot P}\rangle$$

where x, y are affine input and output. This is vital for consistent usage of the clean-up process P. Timed forwarders can be typed in two easy ways: (1) with the tailor-made rules below, or (2) in the system that replaces replication with general recursion, given in [7].

$$\textsc{Ign}\ \frac{a(\tilde{x}).P \vdash a:(\tilde{\tau})^{\downarrow_{\mathrm{A}}}, A^{\text{-}a}}{a(\tilde{x}).(P \mid\ !a(\tilde{x}).0) \vdash a:(\tilde{\tau})^{!_{\mathrm{A}}}, A} \qquad \mathrm{TFW}_0\ \frac{P \vdash A^{\text{-}xy}}{\mathsf{tfw}_{xy}^{0\cdot P} \vdash x:(\tilde{\tau})^{\downarrow_{\mathrm{A}}}, y:(\tilde{\overline{\tau}})^{\uparrow_{\mathrm{A}}}, A} \qquad \mathrm{TFW}_t\ \frac{\mathsf{tfw}_{xy}^{t+1\cdot P} \vdash A}{\mathsf{tfw}_{xy}^{t+2\cdot P} \vdash A}$$

Now $\mathsf{tfw}_{xy}^{t\cdot P}$ is typable as $\mathsf{tfw}_{xy}^{t\cdot P} \vdash x:\tau, y:\overline{\tau}, A$ assuming τ is an distributable affine output, A is a type of P, and x, y do not occur in A. We now consider a simple use of leases. Let the resource in question be $y.\overline{a}$: all we can do with it is to close it by sending a message to the affine name y. Clearly $(\nu y)[y.\overline{a} \mid \overline{y}] \approx^0 [\overline{a}]$. When we access the resource over the net, message loss may leave the resource unclosed: $(\nu y)([y.\overline{a}] \mid [\overline{y}]) \to (\nu y)[y.\overline{a}]$. Now we employ a lease $P \stackrel{\text{def}}{=} !b(r).\overline{r}(x)\mathsf{tfw}_{xy}^{t\cdot\overline{y}}$ to ensure that the resource gets closed automatically if the leaseholder does not close it explicitly.

$$M \stackrel{\text{def}}{=} (\nu y b)([P|y.\overline{a}] \mid [\overline{b}(r)r(x).\overline{x}])$$

Then we show that message-loss does not affect closing the resource: $M \approx^0 [\overline{a}]$.

Verifying the 2PCP. The Two-Phase Commit protocol (2PCP) is a ubiquitous distributed algorithm [3]. It is a network of the form

$$\mathsf{2PCP} \stackrel{\text{def}}{=} (\nu \tilde{d}\tilde{v})([P_1] \mid ... \mid [P_n] \mid [C])$$

It has a coordinator C and n participants P_i, all of which can decide to commit or abort. If all processes decide to commit and the coordinator receives all the votes towards commitment in time, then every participant will commit, otherwise they will all abort. We shall verify this property using our formalism. The protocol is described as follows:

$$P_i \stackrel{\text{def}}{=} P_i^a \oplus P_i^c \quad P_i^a \stackrel{\text{def}}{=} \overline{v_i}^k\langle\mathsf{true}\rangle|\overline{a_i}|!d_i(b).0 \qquad P_i^c \stackrel{\text{def}}{=} \overline{v_i}^k\langle\mathsf{false}\rangle \mid\ !d_i(b).\mathsf{if}\ b\ \mathsf{then}\ \overline{a_i}\ \mathsf{else}\ \overline{c_i}$$

$$C \stackrel{\text{def}}{=} (\nu a\tilde{c})(C_{\mathsf{wait}}|C_{\mathsf{and}}|C_{\mathsf{ab}}) \qquad C_{\mathsf{wait}} \stackrel{\text{def}}{=} \Pi_i v_i(b).\mathsf{if}\ b\ \mathsf{then}\ \overline{e_i}\langle\tilde{d}\rangle\ \mathsf{else}\ \overline{a}\langle\tilde{d}\rangle$$

$$C_{\mathsf{and}} \stackrel{\text{def}}{=} e_1(\tilde{d}).\ ...\ e_n(\tilde{d}).\Pi_i\overline{d_i}^k\langle\mathsf{false}\rangle \qquad C_{\mathsf{ab}} \stackrel{\text{def}}{=} a(\tilde{d}).(\Pi_i\overline{d_i}^k\langle\mathsf{true}\rangle \mid\ !a(\tilde{d}).0)$$

Here we use standard extended syntax $P \oplus Q$ (internal choice) and if-branch (see [7] for their straightforward typing rules). P_i makes a non-deterministic choice between aborting (P_i^a) and committing (P_i^c). P_i^a does two things: it signals its decision to abort

to the outside world by sending an a_i. At the same time, the coordinator is informed of its choice, by sending a vote $\overline{v_i}\langle \mathsf{true} \rangle$ on the internal voting channel v_i. P_i^c is similar in that it sends its vote to the coordinator, but it externalises its decision only after having received back the overall decision from the coordinator. The coordinator has a subprocess C_{wait} that awaits the votes from all participants. Communication between participants and the coordinator happens on v_i, where the votes are cast, and d_i where the coordinator returns its decision back to the participants. This protocol guarantees that either all participants will commit or all participants will abort, assuming that eventually all sent messages will be delivered with high probability. In P_i, $\overline{x}^k\langle \tilde{y} \rangle$ with $k > 0$ is a simpler version of the remote invoker in Example 1 with the following semantics: $\phi(\overline{x}^k\langle \tilde{y} \rangle) = \overline{x}^k\langle \tilde{y} \rangle$ and, writing $\overline{x}^0\langle \tilde{y} \rangle$ for 0,

$$[P|\overline{x}^k\langle \tilde{y} \rangle] \mid [x(\tilde{v}).Q|R] \rightarrow \left\{ \begin{array}{ll} [\phi(P)|\overline{x}^{k-1}\langle \tilde{y} \rangle] \mid [x(\tilde{v}).Q|R] & : r \\ [\phi(P)] \mid [Q\{\tilde{y}/\tilde{v}\}|R] & : 1-r \end{array} \right\}.$$

Here x is of affine type $(\tilde{\tau})^{\uparrow_{\mathrm{A}}}$ with y_i typed by τ_i. The detailed types of this protocol, including the choice and if-branch are given in [7].

Next we state the main result of this subsection that the probability this protocol fails vanishes exponentially quickly in the parameter k.

Theorem 5. *Recall that r is the global message-loss probability. Then:*

$$\mathsf{2PCP} \approx^{e(k)} (\Pi_i[\overline{a_i}]) \oplus (\Pi_i[\overline{c_i}]) \qquad \textit{with } e(k) \stackrel{def}{=} 1-(1-r^k \cdot n)^n$$

Here $\approx^{e(k)}$ is the typed approx. bisimulation.

This theorem states that the probability that the protocol does not reach a consensus is negligible in k. This result improves on [3] in that precise quantities for failure of the protocol are derived. The types are also useful in compositional and quick reasoning about this 2PCP. First the protocol is typed as $\mathsf{2PCP} \vdash \tilde{a} : ()^{?_{\mathrm{A}}}, \tilde{c} : ()^{?_{\mathrm{A}}}$ where $\tilde{c} : \tau$ means $c_1 : \tau, .., c_n : \tau$. Then, for example when considering C we use the fact that $C \vdash \tilde{v} : (\mathtt{bool})^{\downarrow_{\mathrm{A}}}, \tilde{d} : (\mathtt{bool})^{?_{\mathrm{A}}}, \tilde{e} : \updownarrow$. This means we do not have to consider input or output actions happening on a or e_i in the external environment. Likewise, we do not observe non-τ actions from $[P_1] \mid ... \mid [P_n] \mid [C]$ by types. This significantly reduces a number of transitions needed to be considered in reasoning.

5 Conclusion and Related Work

We introduced a convenient typing system for a timed, distributed π-calculus that generalises the existing linear/affine typing discipline [19] for the asynchronous π-calculus. We refined some of the non-determinism in our calculus into probabilities and proposed a notion of typed approximate bisimulation to discard behaviour under a probability threshold. The timed calculus was originally introduced in [3, 4]; [10, 27] propose different timed π-calculi without distribution. Neither alternative considers typing or probabilities. Probabilistic π-calculi are investigated in [9, 11, 12, 17, 30]. None of

these works considers types, timing or distribution, except that [30] uses the affine typing system [5] to prove a correspondence between probabilistic automata, confusion-free event structures and a typed probabilistic π-calculus. The notions of equivalence studied in these works are not directly applicable to our setting because they are not approximate (i.e. they do not allow to quantify parts of the computation that is discarded when considering equality), hence it is not possible to verify the examples in § 4.2.

Our work can be extended in several dimensions. One topic is to investigate our approximate bisimulation, for example by asking how to axiomatise it. We believe the techniques developed for axiomatising weak bisimilarity in [32] to be applicable to the present probabilistic extension, leading to a tractable transformation for reasoning about liveness at each location. It would also be fruitful to use probabilities for constraining other forms of non-determinism. A starting point would be the [IDLE] rule: rather than requiring maximal progress, we could have probabilistic idling. More ambitiously, timer behaviour could be guided by a probability distribution: the key technical challenge here is to find a tractable way of expressing the correlations between *different* probabilistic timers running in parallel.

The paper proposes a general way to integrate the timer with linearity, offering extensibility of various type-based analyses of processes [1, 5, 6, 8, 20, 22, 32] to timing and distribution. First, the secure information flow analysis (SIF) from [19, 21] can be adapted to study timing attacks [24] in distribution. Following [20, 22] an extension of types that accounts of usage numbers of linear channels can lead to more precise type-based SIF analysis in the presence of timers. Secondly, more complicated forms of failure can be considered, like message duplication, correlations between message failures (e.g. if a channel looses a message, the probability of subsequent message-losses increases), site failure [3] or byzantine message corruption [31]. It would also be fruitful to use probabilities for constraining other forms of non-determinism such as the permissible amount of idling. Finally, there is much recent work on (pseudo-)metrics for probabilistic automata [13, 14, 29]. These works do not feature distribution or types, but forging connections with the present approach would be very interesting.

Acknowledgements We thank anonymous reviewers for their helpful comments. This work is partially supported by EPSRC GR/T04724, GR/T03208 and IST-2005-015905 MOBIUS.

References

1. A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. In *OOPSLA '05*, pages 403–422. ACM Press, 2005. A full version will appear in *TCS*.
2. S. Andova, J. C. M. Baeten, and T. A. C. Willemse. A complete axiomatisation of branching bisimulation for probabilistic systems with an application in protocol verification. In *Prc. CONCUR*, volume 4137 of *LNCS*, pages 327–342, 2006.
3. M. Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, London, 2002.
4. M. Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous π-Calculi. In *Proc. CONCUR*, volume 3170 of *LNCS*, pages 115–130, 2004.
5. M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π-calculus. In *Proc. TLCA'01*, volume 2044 of *LNCS*, pages 29–45. Springer-Verlag, 2001.

6. M. Berger, K. Honda, and N. Yoshida. Genericity and the π-calculus. *Acta Inf.*, 42(2-3):83–141, 2005.
7. M. Berger and N. Yoshida. Timed, distributed, probabilistic, typed processes. Long version of the present paper, draft, 2007.
8. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *Proc. ESOP*, LNCS. Springer, 2007.
9. K. Chatzikokolakis and C. Palamidessi. A Framework to Analyze Probabilistic Protocols and its Application to the Partial Secrets Exchange. *TCS*, To appear.
10. J. Chen. A timed mobile calculus. In *Proc. Nordic Workshop on Programming Theory*, pages 65–67, 2004.
11. Y. Deng and W. Du. Probabilistic Barbed Congruence. In *Proc. QAPL*, 2007. To appear.
12. Y. Deng and C. Palamidessi. Axiomatizations for probabilistic finite-state behaviors. *TCS*, 373(1-2):92–114, 2007.
13. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Metrics for Labelled Markov Processes. *TCS*, 318(3):413–354, 2004.
14. J. Desharnais, R. Jagadeesan, V. Gupta, and P. Panangaden. The metric analogue of weak bisimulation for probabilistic processes. In *Proc. LICS*, pages 413–422, 2002.
15. A. Giacalone, C.-C. Jou, and S. A. Smolka. Algebraic reasoning for probabilistic concurrent systems. In *Proc. Working Conf. on Programming Concepts and Methods*, pages 443–458, 1990.
16. C. G. Gray and D. R. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. Technical Report CS-TR-90-1298, Stanford University, 1990.
17. O. M. Herescu and C. Palamidessi. Probabilistic asynchronous π-calculus. In *Proceedings of 3rd FoSSaCS*, volume 1784 of *LNCS*, pages 146–160. Springer, 2000.
18. K. Honda. Composing Processes. In *POPL'96*, pages 344–357. ACM Press, 1996.
19. K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *POPL'02*, pages 81–92. ACM Press, 2002. Full version to appear in ACM TOPLAS.
20. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.
21. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.
22. N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247, 2006.
23. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM TOPLAS*, 21(5):914–947, Sept. 1999.
24. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. CRYPTO '96*, LNCS, pages 104–113, 1996.
25. D. L. Mills. The network computer as precision timekeeper. In *Proc. Precision Time and Time Interval (PTTI)*, pages 96–108, 1996.
26. S. Mullender, editor. *Distributed Systems*. Addison-Wesley, 1993.
27. W. C. Rounds and H. Song. The Phi-Calculus: A Language for Distributed Control of Reconfigurable Embedded Systems. In *Proc. HSCC*, pages 435–449, 2003.
28. R. Segala. Probability and Nondeterminism in Operational Models of Concurrency. In *Proc. CONCUR*, LNCS, pages 64–78. Springer, 2006.
29. F. van Breugel and J. Worrell. A Behavioural Pseudometric for Probabilistic Transition Systems. *TCS*, 331(1):115–142, 2005.
30. D. Varacca and N. Yoshida. Probabilistic π-Calculus and Event Structures. In *Proc. QAPL*, ENTCS, 2007. To appear.
31. M. Ying. π-calculus with noisy channels. *Acta Informatica*, 41(9):525–593, 2005.
32. N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π-Calculus. *Inf. & Comp.*, 191(2004):145–202, 2004.