MAGπ: Types for Failure-Prone Communication

Matthew Alan Le Brun and Ornela Dardha

University of Glasgow
m.le-brun.1@research.gla.ac.uk
ornela.dardha@glasgow.ac.uk

Abstract. *Multiparty Session Types* (MPST) are a typing discipline for communication-centric systems, guaranteeing communication safety, deadlock freedom and protocol compliance. Several works have emerged which model failures and introduce fault-tolerance techniques. However, such works often make assumptions on the underlying network, *e.g.*, assuming TCP-based communication where messages are guaranteed to be delivered; or adopting centralised reliable nodes and ad-hoc notions of reliability; or only addressing a single kind of failure, such as node crashes. In this work, we develop MAGπ—a Multiparty, Asynchronous and Generalised π-calculus, which is the *first language and type system* to accommodate in unison: (*i*) the widest range of non-Byzantine faults, including *message loss, delays* and *reordering*; *crash* and *link failures*; and *network partitioning*; (*ii*) a novel and most general notion of *reliability*, taking into account the viewpoint of *each* participant in the protocol; (*iii*) a spectrum of network assumptions from the lowest UDP-based network programming to the TCP-based application level. We prove subject reduction and session fidelity; process properties (deadlock freedom, termination, *etc.*); failure-handling safety and reliability adherence.

Keywords: Session types · Distributed protocols · Failures · Timeouts

1 Introduction

Despite large investments into fault-prevention techniques, failures still regularly occur in complex distributed applications. It is widely agreed that traditional methods of verification using software testing do not provide high levels of confidence in the correctness of distributed algorithms. This is mainly due to the *non-deterministic* behaviour inherent to these protocols, which makes it unfeasible to manually test for all edge cases. This problem is bypassed by using exhaustive techniques such as model checking [9,31], capable of exploring the entirety of the state space of a program to verify its correctness. However, building suitable models for complex distributed algorithms is arduous, expensive, and often intractable (due to the state explosion problem [10]). Furthermore, even if an algorithm is successfully encoded into a suitable model and checked, guarantees of correctness are on the *design* of the algorithm, and not on the software *implementation*; handwritten code is still prone to human error. Contrastively, types

and type systems [29] are lightweight forms of verification. Baked in programming languages, types provide guarantees directly on handwritten code and aid developers in implementing software which is correct by construction. Specific to concurrent and distributed computing, *session types* [14,35,15,36,33,16] have quickly grown in popularity since their initial conceptualisation [14], spanning from *binary*–two participants, to *multiparty*–many participants.

Session types enforce that processes communicate according to a protocol specification. Consequently, desirable properties about communication, *e.g.*, *type safety* (communication occurs error-free), *protocol compliance* (or session fidelity; processes behave according to their predefined protocol), and *deadlock freedom* (processes do not get stuck waiting), can be *statically* determined by a type checker. To this aim, session types have been implemented in various programming languages, including Java [18,11], Go [21], Haskell [17,27], Scala [32], Rust [19], Elixir [34].

To date, most session type theories are designed for *concurrent*, as opposed to *distributed* processes—*i.e.,* it is commonly assumed that communication failures do not occur. For the few (and rapidly increasing) works that do consider failures, heavy assumptions are made that impede their viability for realistic complex distributed applications. *E.g.*, *asynchronous* theories [24,16,33] use *message buffers* to model distributed communication under "TCP-like" assumptions: messages are guaranteed to be delivered and messages from a single sender do not get reordered. *Affine sessions* [25,12,6] only allow failure-handling of application level failures through *try-catch* blocks; there is no support for *arbitrary* failures that may stem from hardware faults, network inconsistencies *etc. Coordinator model* approaches [1,8,37] assume some degree of reliability, be it as a central resilient process, a reliable broadcast, or fixed synchronisation points.

The harsh reality is that many real-world distributed protocols (*e.g.*, consensus algorithms) cannot assume *any* of these conditions. Networks introduce many points of failure into a system: nodes may crash, messages can be dropped, delayed or duplicated, links between nodes may fail *etc.* Designers of distributed protocols have acknowledged that failure is inevitable, and so algorithms are designed to withstand a threshold of failure whilst still achieving their expected behaviour—known as *fault-tolerance* [22]. Examples of fault-tolerant protocols (extensively) used today include the Paxos [20] and Raft [26] consensus algorithms, which assume the possibility of *all non-Byzantine* faults—*i.e.,* node crashes, link failures, network partitions, and message inconsistencies.

Although the correctness of these algorithms has been heavily studied, many of them are developed with limited confidence in the correctness of the deployable artifact, due to the reasons previously outlined. To fill this gap, we need type-based verification, which can be made available to programming languages, thus supporting designers and developers in designing and implementing correct distributed algorithms. While (multiparty) session types have made great impact in modelling structured communication and guaranteeing relevant properties, their theory is not yet expressive to model these complex algorithms.

In this paper, we take steps towards filling this gap by presenting MAGπ—a Multiparty, Asynchronous and Generalised π-calculus—the first language and type system able to accommodate: (*i*) the widest range of non-Byzantine faults, including *message loss*, *delays* and *reordering*; *crash* and *link failures*; and *network partitioning*—all by using *timeouts*; (*ii*) a novel and most general notion of *reliability*, taking into account the viewpoint of each participant in the protocol; and (*iii*) a spectrum of network assumptions—from the lowest level of network programming based on UDP, to application level based on TCP.

Example 1 (Ping Pong: Types). We illustrate MAGπ with a simplified version of the ping utility from the Internet Control Message Protocol (ICMP[1]), which is our running example. The ping utility consists of a total of three *roles* communicating amongst each other: two roles, **p** and **r**, communicate *reliably* with each other, and both communicate *unreliably* with a third role **q**. Our definition of reliability (§ 3.2) takes into account the viewpoint of each role, thus allowing roles to have their own (possibly empty) *reliability set*. Following the assumptions above, the reliability set for **p** is $\{\mathsf{r}\}$, for **r** is $\{\mathsf{p}\}$, and for **q** is $\emptyset$.

Below we give the session types, denoted $\mathbb{S}_{\mathsf{r}}$, $\mathbb{S}_{\mathsf{p}}$ and $\mathbb{S}_{\mathsf{q}}$ for roles **r**, **p** and **q** respectively.

$$\mathbb{S}_{\mathsf{r}} = \&\{\mathsf{p}\,?\,\mathsf{ok}().\mathbf{end},\ \mathsf{p}\,?\,\mathsf{ko}().\mathbf{end}\}$$

$$\mathbb{S}_{\mathsf{p}} = \mathsf{q}\,!\,\mathsf{ping}().\,\&\begin{cases}\mathsf{q}\,?\,\mathsf{pong}().\mathsf{r}\,!\,\mathsf{ok}().\mathbf{end},\\ \circledcirc.\,\mathsf{q}\,!\,\mathsf{ping}().\,\&\begin{cases}\mathsf{q}\,?\,\mathsf{pong}().\mathsf{r}\,!\,\mathsf{ok}().\mathbf{end},\\ \circledcirc.\,\mathsf{q}\,!\,\mathsf{ping}().\,\&\begin{cases}\mathsf{q}\,?\,\mathsf{pong}().\mathsf{r}\,!\,\mathsf{ok}().\mathbf{end},\\ \circledcirc.\,\mathsf{r}\,!\,\mathsf{ko}().\mathbf{end}\end{cases}\end{cases}\end{cases}$$

$$\mathbb{S}_{\mathsf{q}} = \&\begin{cases}\mathsf{p}\,?\,\mathsf{ping}().\mathsf{p}\,!\,\mathsf{pong}().\mathbf{end},\\ \circledcirc.\,\&\begin{cases}\mathsf{p}\,?\,\mathsf{ping}().\mathsf{p}\,!\,\mathsf{pong}().\mathbf{end},\\ \circledcirc.\,\&\begin{cases}\mathsf{p}\,?\,\mathsf{ping}().\mathsf{p}\,!\,\mathsf{pong}().\mathbf{end},\\ \circledcirc.\,\mathbf{end}\end{cases}\end{cases}\end{cases}$$

Role **r** is the *receiver* (&–called *branching*), which waits on two options: it receives from **p** either the label ok or ko and then it terminates the protocol (**end**). Role **p** is the *sender* ($\oplus$[2]–called *selection*), and it tries to obtain information on the status of **q**. It begins by sending a ping message to **q** ($\mathsf{q}\,!\,\mathsf{ping}()$), then waits to receive from **q**. If a pong is received ($\mathsf{q}\,?\,\mathsf{pong}()$) in the top branch, then it concludes that the status of **q** is *reachable* and sends this information to **r** ($\mathsf{r}\,!\,\mathsf{ok}()$), after which it terminates. Alternatively, **p** enters a *timeout branch* ($\circledcirc$). For simplicity, we assume **p** will attempt to communicate with **q** three times (shown in the three-time indentation of the timeout branch) before assuming **q** is *unreachable*; after which the session will also terminate by sending ko to **r**, followed by **end**. In the same lines, the protocol for role **q** is given by the session type $\mathbb{S}_{\mathsf{q}}$, where its timeout branches match the timeouts from $\mathbb{S}_{\mathsf{p}}$.

---

[1] https://www.rfc-editor.org/rfc/rfc792

[2] For readability, we adopt a shorthand notation for sending towards a single role and for payloads of type unit, such that $\oplus\{\mathsf{s}\,!\,\mathsf{m}(\mathtt{unit}).\mathbb{S}\}$ is represented by $\mathsf{s}\,!\,\mathsf{m}().\mathbb{S}$.

1.1 Contributions

We now present our contributions w.r.t. our Multiparty, Asynchronous, and Generalised π-calculus (MAGπ).

1. **MAGπ** language (§ 2):
   - MAGπ is *the first language* to support the widest set of non-Byzantine faults, including **message loss**, **message delays** and **message reordering**; **crash failures** and **link failures**; and **network partitioning**.
   - MAGπ is *the first language* to introduce **timeouts** in receive branches (used for handling network failures), as well as support **undirected branching** in a *generalised* setting—the ability to simultaneously expect an incoming message from more than one sender.
2. **MAGπ** type system (§ 3):
   - is a *conservative extension* of a generalised asynchronous MPST theory [33], benefiting from: the ability to model *more protocols* than traditional syntactic theories (*e.g.* global types); and the flexibility of checking desired properties, such as deadlock freedom or termination, *a posteriori*—as opposed to during the design phase.
   - supports **undirected branching/selection** and is the first type system to introduce **timeout branches**.
   - supports a novel and most general *reliability* definition (§ 3.2), taking into account the viewpoint of *each* participating role, and is built on *optional role-dependant* reliability assumptions.
3. **Type properties** (§ 4): We prove subject reduction (theorem 1) and session fidelity (theorem 2). We show failure-handling safety (cor. 1) and its inverse, reliability adherence (cor. 2), which strictly connect timeouts and reliability. We prove process properties (theorem 3) *e.g.* deadlock-freedom, termination, liveness, in line with [33]. Finally, as our MAGπ type system is Turing-complete, we prove decidable type checking (theorem 4) and decidability of process properties for finite message buffers (theorem 5).
4. **TCP vs. UDP** (§ 5): MAGπ is expressive enough to capture a range of network assumptions: from low-level network programming operating over the User Datagram Protocol (UDP); to application-level software operating over the Transmission Control Protocol (TCP).
5. **Case study** (§ 6): we further demonstrate the use of timeouts and undirected branching to model a Domain Name System (DNS) distributed protocol with a cache and in-built load-balancer; we also show the properties it satisfies, including safety and deadlock freedom. Further examples are available in the technical report [23], including a prototype specification of a leader election algorithm used by consensus protocols.

2 MAGπ: Language

We present a multiparty session π-calculus, based on the theory of Scalas and Yoshida [33], extended to accurately model real-world distributed network environments. We assume the *lowest level* of abstraction—the only *failure detection mechanism* available to a process is an upper-bound wait limit, *i.e.,* a *timeout*.

$$
\begin{array}{rrll}
c & ::= & x \mid s[\mathbf{p}] & (\textit{variable, session w/ role}) \\
d & ::= & v \mid c & (\textit{basic value, variable, session w/ role}) \\
w & ::= & v \mid s[\mathbf{p}] & (\textit{basic value, session w/ role}) \\
P, Q & ::= & \mathbf{0} \mid (\nu s)\, P & (\textit{inaction, restriction}) \\
 & \mid & P \mid Q \mid P + Q & (\textit{composition, non-deterministic choice}) \\
 & \mid & c \oplus [\mathbf{q}]\,!\,\mathsf{m}\langle d \rangle.\, P & (\textit{selection towards role } \mathbf{q}) \\
 & \mid & c \,\&_{i \in I} \{[\mathbf{q}_i]\,?\,\mathsf{m}_i(x_i).P_i\} & (\textit{reliable branching from roles } \mathbf{q}_i) \\
 & \mid & c \,\&_{i \in I} \{[\mathbf{q}_i]\,?\,\mathsf{m}_i(x_i).P_i,\ \textcircled{1}.\, Q\} & (\textit{branching from roles } \mathbf{q}_i \textit{ w/ } \textbf{\textit{timeout}}\ Q) \\
 & \mid & \mathsf{def}\ D\ \mathsf{in}\ P \mid X\langle \tilde{d} \rangle & (\textit{process definition, process call}) \\
 & \mid & s : \sigma & (\textit{session buffer}) \\
D & ::= & X(\tilde{x}) = P & (\textit{process declaration}) \\
\sigma & ::= & (\mathbf{p} \triangleright \mathbf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma \mid \epsilon & (\textit{session queue, empty session queue})
\end{array}
$$

Fig. 1. Syntax for MAGπ

Our calculus presents three novel features: *(i)* the new *timeout* primitive; *(ii)* the capability of expecting a message from *different senders*; and *(iii)* operational semantics which can model *various non-Byzantine failures*. Timeouts can be attached to receive actions—henceforth referred to as *branches*—and are used to describe an alternative process to be executed in case failures are *assumed* to occur (akin to error handlers).

Failures are said to be *assumed*, as opposed to detected, since we model the impossibility result of distinguishing between a delayed *vs* lost message. Thus, it is possible for a processes to prematurely timeout without its corresponding message having been lost—just like the real-world!

The benefit of our approach is that the *failure detection mechanism is agnostic to the type of fault*, allowing us to model in unison **message loss**, **message delay**, **crash-stop failures**, **link failures**, and **network partitions**.

2.1 Syntax

Definition 1 (Language Syntax). *The Multiparty, Asynchronous and Generalised π-calculus syntax is defined by the grammar in fig. 1.*

Communication happens over sessions (s, s') between a number of roles ($\mathbf{p}$, $\mathbf{q}$) ranging over set $\boldsymbol{\rho}$. The primitives of the calculus are *sessions with roles* $s[\mathbf{p}]$, and basic values v, both of which can be abstracted using *variables* (x, y). Processes (P, Q), include the following standard constructs: *(i) inaction* $\mathbf{0}$ represents process termination; *(ii)* session *restriction* $(\nu s)\, P$ binds a new session s in P; *(iii)* parallel *composition* declares two concurrent processes; *(iv) selection* $c \oplus [\mathbf{q}]\,!\,\mathsf{m}\langle d \rangle.\, P$ uses channel c to send a message to $\mathbf{q}$ with label m and payload

$$[\textbf{R-}\oplus]\quad s[\mathsf{p}]\oplus[\mathsf{q}]\,!\,\mathsf{m}\langle w\rangle.\,P \;\mid\; s{:}\sigma \longrightarrow P \;\mid\; s{:}\sigma\cdot(\mathsf{p}\triangleright\mathsf{q}\triangleleft\mathsf{m}\langle w\rangle)\cdot\epsilon$$

$$[\textbf{R-}\&]\quad s[\mathsf{q}]\,\&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i\;[,\,\textcircled{t}.\,Q]\} \;\mid\; s{:}(\mathsf{p}_k\triangleright\mathsf{q}\triangleleft\mathsf{m}_k\langle w\rangle)\cdot\sigma \longrightarrow P_k[{}^{w}/_{x_k}] \;\mid\; s{:}\sigma \qquad \text{for } k\in I$$

$$[\textbf{R-}\textcircled{t}]\quad s[\mathsf{q}]\,\&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i,\,\textcircled{t}.\,Q\} \;\mid\; s{:}\sigma \longrightarrow Q \;\mid\; s{:}\sigma$$

$$[\textbf{R-}+]\quad P_1+P_2 \longrightarrow P_i \qquad \text{for } i\in\{1,2\}$$

$$[\textbf{R-}\boldsymbol{X}]\quad \mathsf{def}\;X(x_1,\ldots,x_n)=P\;\mathsf{in}\;(X\langle w_1,\ldots,w_n\rangle \mid Q) \longrightarrow \mathsf{def}\;X(x_1,\ldots,x_n)=P\;\mathsf{in}\;(P[{}^{w_1}/_{x_1}]\cdots[{}^{w_n}/_{x_n}] \mid Q)$$

$$[\textbf{R-}\mathbb{C}]\quad P\longrightarrow P' \implies \mathbb{C}[P]\longrightarrow\mathbb{C}[P']$$

$$[\textbf{R-}\downarrow]\quad s{:}h\cdot\sigma\longrightarrow s{:}\sigma$$

Fig. 2. Reduction rules for MAGπ

d—after sending, the process continues according to P; *(v) definition* and *declaration* allow processes to be assigned names, modelling recursion through the use of process *calls*. We now elaborate on the novelties in our language.

$\mathbf{c}\,\&_{i\in I}\{[\mathsf{q}_i]\,?\,\mathbf{m}_i(d).P\}$ Quantification over roles in a branch allows processes to receive from one in a range of other roles. This has practical applications in a multitude of distributed protocols, *e.g. load balancers*.

$\mathbf{c}\,\&_{i\in I}\{[\mathsf{q}_i]\,?\,\mathbf{m}_i(d).P,\ \textcircled{t}.\,Q\}$ Timeouts are used as a *failure detection mechanism* in receive branches. If a failure is assumed to have lost or prevented the incoming message, then process Q is initiated. It is key to note that timeouts are non-deterministic—they model an arbitrary and unknown duration of time a process waits before assuming a failure has occurred.

$P+Q$ Non-deterministic choice randomly picks between two possible process continuations. We use this construct to simplify examples for better presentation. Concretely, it replaces the need for *expressions* and *if-then-else* constructs, which are routine and orthogonal to our formulation.

$s:\sigma$ Message buffer for session s. An entry in the buffer $(\mathsf{p}\triangleright\mathsf{q}\triangleleft\mathsf{m}\langle w\rangle)$ models a message "in transit" from role p to q with label m and payload w. This is needed to accommodate asynchrony in our language.

2.2 Operational Semantics

We begin with definitions of a reduction context and buffer congruence.

Definition 2 (Reduction Context). *A* ***reduction context*** $\mathbb{C}$ *abstracts away an outer environment from a process, and is given by:*

$$\mathbb{C} \;::=\; \mathbb{C}\,|\,P \;\mid\; (\nu s)\,\mathbb{C} \;\mid\; \mathsf{def}\;D\;\mathsf{in}\;\mathbb{C} \;\mid\; [\,]$$

Hence, $\mathbb{C}[P]$ *refers to process* P *under some arbitrary context* $\mathbb{C}$.

Definition 3 (Buffer Congruence). *A process containing* ***only*** *a buffer under its restriction is congruent to inaction. Message buffers observe total reordering.*

$$(\nu s)\, s{:}\sigma \;\equiv\; \mathbf{0} \qquad\qquad s{:}\sigma_1 \cdot h_1 \cdot h_2 \cdot \sigma_2 \;\equiv\; s{:}\sigma_1 \cdot h_2 \cdot h_1 \cdot \sigma_2$$

Definition 4 (OS). *The operational semantics for MAGπ is given via a* reduction relation $\longrightarrow$ *inductively defined in fig. 2, together with standard* structural congruence rules *[33] and two* buffer congruence rules *defined in def. 3.*

Let us now comment on the reduction rules (fig. 2). Processes send messages using the **selection rule** [**R-**$\oplus$]; this adds the sent message as a new entry in the session buffer, and advances the process to its continuation. Sent messages are read from the buffer using the **branching rule** [**R-** $\&$]. If the receiver has a valid branch matching the sender and message label, then it advances to the specific continuation of said branch (a timeout branch for this rule is optional). The substitution $P_k[^{w}/_{x_k}]$ denotes the replacement of variable x_k with the payload value w in the continuation process P_k. The **timeout rule** [**R-**$\mathcal{O}$] advances processes to their timeout branch *without* changing the buffer. Non-deterministic choice is resolved using the **choice rule** [**R-**$+$], which advances the process to one of the two possible continuations. The **call rule** [**R-*X***] replaces a process call with its defined process, substituting each parameter. Processes can reduce under a context using the **context rule** [**R-**$\mathbb{C}$]. Lastly, messages can be lost from the buffer with the **drop rule** [**R-**$\downarrow$].

We now unpack how our semantics deals with failures. The reduction rules in fig. 2 allow various forms of failures to be modelled, stemming from the versatility and elegance of the drop rule [**R-**$\downarrow$]. The following elaborates on how this rule can be utilised to model different types of failure:

- **Message loss** is modelled directly through the reduction rule [**R-**$\downarrow$].
- **Crash-failure** is modelled through repeated applications of [**R-**$\downarrow$] for a particular role. *E.g.*, to model a crash of role p, the reduction step [**R-**$\downarrow$] should be applied to all messages that enter the buffer matching the pattern $(\mathsf{p} \triangleright \_ \triangleleft \_\langle \_ \rangle)$ ($\_$ symbolises a "don't care" value).
- **Link-failure** is modelled using a similar method; the difference being that messages between *two* specific recipients are dropped. *E.g.*, modelling a link-failure between roles p and q requires [**R-**$\downarrow$] to be applied to all messages entering the buffer with the patterns $(\mathsf{p} \triangleright \mathsf{q} \triangleleft \_\langle \_ \rangle)$ and $(\mathsf{q} \triangleright \mathsf{p} \triangleleft \_\langle \_ \rangle)$.
- **Message delay** is modelled by applying rule [**R-**$\mathcal{O}$] to a branch whilst a valid message resides in the buffer. *E.g.*:

$$\begin{aligned} &s[\mathsf{q}]\,\&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i,\ \mathcal{O}.\,Q\} \;\mid\; s{:}(\mathsf{p}_k \triangleright \mathsf{q} \triangleleft \mathsf{m}_k\langle w\rangle) \cdot \sigma \\ &\quad\longrightarrow\; Q \;\mid\; s{:}(\mathsf{p}_k \triangleright \mathsf{q} \triangleleft \mathsf{m}_k\langle w\rangle) \cdot \sigma \qquad\qquad \text{for } k \in I. \end{aligned}$$

- Total **message reordering** is modelled via *buffer congruence rules* (def. 3).
- **Network partitions** can be represented using multiple *link failures*.

The granularity at which we model failures allows for degrees of customisation. *E.g.*, benign fault-tolerant consensus algorithms typically assume the possibility of *all* non-Byzantine faults, therefore all the aforementioned failures are required. Alternatively, an application assumed to run over a trusted TCP network need not worry about single message drops, and hence **[R-↓]** should only be applied to model node crash and link failures.

Definition 5 (Well-formedness). *To ensure that communication is possible, we require that a well-formed process has a buffer for each session, i.e.,*

$$P = (\nu s)\, Q \implies Q \equiv (\nu \tilde{s'})\, (Q' \mid s{:}\sigma)$$

Def. 5 introduces a well-formedness condition to guarantee that a session always guards its buffer, hence ensuring that messages always have a queue to be placed in. From now on, we will only consider well-formed processes.

Before concluding this section, we recall our ping pong running example from the introduction, and present below the processes for roles p, q and r.

Example 2 (Ping Pong: Processes).

$$P_{\mathsf{p}} = s[\mathsf{p}] \oplus [\mathsf{q}]\,!\,\mathsf{ping}\langle\rangle.\, s[\mathsf{p}]\,\&\begin{cases}[\mathsf{q}]\,?\,\mathsf{pong}().P_{\mathsf{p}}^{ok},\\ \textcircled{\scriptsize t}.\, s[\mathsf{p}] \oplus [\mathsf{q}]\,!\,\mathsf{ping}\langle\rangle.\, s[\mathsf{p}]\,\&\begin{cases}[\mathsf{q}]\,?\,\mathsf{pong}().P_{\mathsf{p}}^{ok},\\ \textcircled{\scriptsize t}.\, s[\mathsf{p}] \oplus [\mathsf{q}]\,!\,\mathsf{ping}\langle\rangle.\, P'_{\mathsf{p}}\end{cases}\end{cases}$$

$$P_{\mathsf{p}}^{ok} = s[\mathsf{p}] \oplus [\mathsf{r}]\,!\,\mathsf{ok}\langle\rangle.\,\mathbf{0} \qquad P'_{\mathsf{p}} = s[\mathsf{p}]\,\&\begin{cases}[\mathsf{q}]\,?\,\mathsf{pong}().P_{\mathsf{p}}^{ok},\\ \textcircled{\scriptsize t}.\, s[\mathsf{p}] \oplus [\mathsf{r}]\,!\,\mathsf{ko}\langle\rangle.\,\mathbf{0}\end{cases}$$

$$P_{\mathsf{q}} = s[\mathsf{q}]\,\&\begin{cases}[\mathsf{p}]\,?\,\mathsf{ping}().P_{\mathsf{q}}^{pong},\\ \textcircled{\scriptsize t}.\, s[\mathsf{q}]\,\&\begin{cases}[\mathsf{p}]\,?\,\mathsf{ping}().P_{\mathsf{q}}^{pong},\\ \textcircled{\scriptsize t}.\, s[\mathsf{q}]\,\&\{[\mathsf{p}]\,?\,\mathsf{ping}().P_{\mathsf{q}}^{pong},\ \textcircled{\scriptsize t}.\,\mathbf{0}\}\end{cases}\end{cases}$$

$$P_{\mathsf{q}}^{pong} = s[\mathsf{q}] \oplus [\mathsf{p}]\,!\,\mathsf{pong}\langle\rangle.\,\mathbf{0}$$

$$P_{\mathsf{r}} = s[\mathsf{r}]\,\&\{[\mathsf{p}]\,?\,\mathsf{ok}().\mathbf{0}, [\mathsf{p}]\,?\,\mathsf{ko}().\mathbf{0}\}$$

3 MAGπ: Type System

We introduce the type system for MAGπ, which is a conservative extension of the *generalised* asynchronous MPST theory [33, sec. 7]. *Generalised* MPST stray away from global protocol specifications (global types) and instead operate on user-defined localised specifications of each participating role (local types). The benefits of working with such theory include: *(i)* the ability to capture a larger set of viable protocols compared to traditional syntactic methods (*e.g.* global types) of enforcing consistent communication; *(ii)* the ability to model protocols of different requirements. In particular, instead of syntactically enforcing programmers to write, *e.g.*, deadlock-free code, a generalised theory allows programmers to unrestrictedly design protocols that are checked *a posteriori* against any number of required properties, such as deadlock-freedom, termination *etc.*

Basic and Session Types

$$\mathbb{T} ::= \mathbb{B} \mid \mathbb{S}$$
$$\mathbb{B} ::= \texttt{int} \mid \texttt{bool} \mid \texttt{real} \mid \texttt{unit} \mid \cdots$$
$$\mathbb{S} ::= \&_{i\in I}\{\mathsf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i[,\ \textcircled{\scriptsize t}.\,\mathbb{S}']\}$$
$$\mid \ \oplus_{i\in I}\{\mathsf{p}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$$
$$\mid \ \mu \mathrm{t}.\mathbb{S} \mid \mathrm{t} \mid \textbf{end}$$

Buffer Types

$$\mathbb{M} ::= \mathsf{p}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M} \mid \epsilon$$

Session-Buffer Types

$$\tau ::= \mathbb{M} \mid \mathbb{S} \mid (\mathbb{M}\,;\mathbb{S})$$

Fig. 3. Basic Types, Session Types, Buffer Types and Session-Buffer Types

$$\frac{}{\mathbb{T} \equiv \mathbb{T}} \qquad \frac{}{\mathbb{M}_1 \cdot \mathbb{M}_2 \equiv \mathbb{M}_2 \cdot \mathbb{M}_1} \qquad \frac{}{\epsilon \cdot \epsilon \equiv \epsilon} \qquad \frac{\mathbb{M} \equiv \mathbb{M}' \qquad \mathbb{S} \equiv \mathbb{S}'}{(\mathbb{M}\,;\,\mathbb{S}) \equiv (\mathbb{M}'\,;\,\mathbb{S}')}$$

Fig. 4. Type congruence rules

The novelties of our type system include: *(i) undirected branching/selection*; *(ii) timeout branches* (syntax in § 3.1); and *(iii) reliability sets*—sets of roles assumed to not fail, from the perspective of *each* role (§ 3.2). Reliability sets (possibly empty) enforce the use of *timeouts* for all failure-prone communication.

As in [33], our type system does *not* use global types, but solely relies on local types. Consequently, typing contexts must obey a safety property to ensure subject reduction (§ 3.3). Finally, we present the rules for our type system in § 3.4, and discuss its key properties in § 4.

3.1 Types

Our MPST theory is designed for the distributed computing setting. Concretely, our type system (def. 6) is *asynchronous*; it allows *branching* (resp. *selection*) from (resp. to) multiple roles; and supports *timeout* continuation types.

Definition 6 (Typing syntax). *The typing syntax is defined using the grammar in fig. 3. For undirected branching and selection, $I \neq \emptyset$ and role-label tuples $(\mathsf{p}_i, \mathsf{m}_i)$ must be pairwise distinct. Recursion variables cannot be free and must appear guarded under branching/selection types.*

Type $\mathbb{T}$ denotes either a *basic* type $\mathbb{B}$, or a *session type* $\mathbb{S}$, and is used to type variables. Session types describe how a channel should be used: (i) *undirected branching* (external choice) $\&_{i\in I}\{\mathsf{p}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i[,\ \textcircled{\scriptsize t}.\,\mathbb{S}']\}$ denotes *receiving* a message with label m_i and payload of type $\mathbb{T}_i$ from role p_i, then continuing according to $\mathbb{S}_i$. The (optional) timeout continuation type $\mathbb{S}'$ describes the protocol for handling failure on that branch; (ii) *undirected selection* (internal choice) $\oplus_{i\in I}\{\mathsf{p}_i\,!\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$ denotes *sending* a message with label m_i and payload $\mathbb{T}_i$ to role p_i, then continuing according to $\mathbb{S}_i$; (iii) type **end** marks a channel as closed, and terminates communication. A session buffer is typed using the *buffer*

type $\mathbb{M}$. Entries in the buffer must correspond to the type $\mathsf{p}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M}$, denoting a message sent to p with label m and payload of type $\mathbb{T}$. A session with role is typed using *session-buffer* types, combining a session type and a buffer type.

Type *congruence* $\equiv$ is defined in fig. 4. Notably, buffer types can be reordered, and two session-buffer types are congruent if their individual buffer and session types are congruent. Buffer type reordering is necessary to match the *total message reordering* supported by the language (def. 3).

3.2 Reliability

We go on a short detour and talk about reliability. Previous related work [4,1,38] have included the notion of *reliability* into their type systems. Generally, either one specific role, or a pre-defined set of roles, are assumed to be reliable—*i.e.*, no failures occur for communication involving the identified set of roles.

Our definition of reliability (def. 7) is the most general and the first to take into account the viewpoint of each role. We argue that this is necessary in a distributed setting since reliability in networks is dependant on the physical topology of processes. Recalling the ping utility (example 1), we could imagine the processes representing roles p and r reside on the same physical hardware, thus their communication cannot be affected by network faults; and the process for q resides on geographically separated hardware, therefore its communication with both p and r is vulnerable to failure.

Definition 7 (Reliability). *The* ***reliability set*** $\mathfrak{R}$ *for a role* $\mathsf{p} \in \boldsymbol{\rho}$ *is defined as* $\mathfrak{R} \subseteq \boldsymbol{\rho} \setminus \{\mathsf{p}\}$*, capturing the viewpoint of* p*.* ***Reliability*** $\mathcal{R}$ *is defined as a function mapping roles to their reliability set, i.e.,* $\mathcal{R} : \boldsymbol{\rho} \to \mathfrak{R}$*.*

To better model real distributed environments, our definition of reliability allows each role to have its own (possibly empty) reliability set.

Example 3 (Ping Pong: Reliability Sets). W.r.t. example 1, as the three roles have different viewpoints on each other, then the reliability set for each of them is different. In particular, we have $\mathcal{R}(\mathsf{p}) = \{\mathsf{r}\},\ \mathcal{R}(\mathsf{r}) = \{\mathsf{p}\},\ \mathcal{R}(\mathsf{q}) = \emptyset$.

Investigating the extremes, we have: for a set of roles $\boldsymbol{\rho}$, if for all $\mathsf{p} \in \boldsymbol{\rho} \cdot \mathcal{R}(\mathsf{p}) = \emptyset$, then *no* communication is reliable; conversely, if for all $\mathsf{p} \in \boldsymbol{\rho} \cdot \mathcal{R}(\mathsf{p}) = \boldsymbol{\rho} \setminus \{\mathsf{p}\}$, then *all* communication is reliable—referred to as a *reliable network*. This work only considers static configurations for $\mathcal{R}$, thus reliability sets cannot change at runtime. We find that even with this restriction, our definition is the most general compared to related work.

3.3 Contexts

Definition 8 (Type contexts). *Context* Θ *is a partial mapping from process variables to n-tuples of types and context* Γ *is a partial mapping from variables to types, and sessions with roles to session-buffer types, both defined below:*

$$\Theta ::= \emptyset \mid \Theta, X : \mathbb{T}_1, \ldots, \mathbb{T}_n \qquad \Gamma ::= \emptyset \mid \Gamma, x : \mathbb{T} \mid \Gamma, s[\mathsf{p}] : \tau$$

*The **composition** of contexts (Γ_1, Γ_2) is defined iff:*

$$\forall c \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) : \quad \Gamma_i(c) = \mathbb{M} \ \wedge \ \Gamma_j(c) = \mathbb{S}$$

For such c, $(\Gamma_1,\Gamma_2)(c) = (\mathbb{M};\mathbb{S})$.
*Contexts are **congruent** $\Gamma_1 \equiv \Gamma_2$ iff:*

$$\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2) \ \wedge \ \forall c \in \mathsf{dom}(\Gamma_1) : \Gamma_1(c) \equiv \Gamma_2(c)$$

Context Θ is *non-linear* and types *process variables* by tracking the types of their parameters. Context Γ is *linear* and allows *variables* to have *basic* or *session types*, and *sessions with roles* to have *session-buffer types*; as a program progresses, a role may simultaneously have both an active session type and messages queued in the message buffer.

Context *composition* allows two contexts to coexist as long as their common channels map to buffer types in one context, and session types in the other.

Context *congruence* holds if two contexts have the same domain and the types of their channels are congruent. It is key to note that by the definitions of context composition and congruence we have $s[\mathbf{p}] : (\mathbb{M}; \mathbb{S}) \equiv s[\mathbf{p}] : \mathbb{M}, s[\mathbf{p}] : \mathbb{S}$. Buffer types (resp. session-buffer types) are only used internally by the type system; end-users are not expected to explicitly define these types.

Definition 9 (Context reduction). *An **action** α is given as:*

$$\alpha \ ::= \ s[\mathbf{p}]\,!\,\mathbf{q} : \mathsf{m}(\mathbb{T}) \ \mid \ s[\mathbf{p}][\mathbf{q}] : \mathsf{m} \ \mid \ s[\mathbf{p}]\,\textcircled{t}$$

*From left to right, this reads as (i) a **sent message**; (ii) **communication** of a message; and (iii) the **timeout** of a channel. **Context transition** $\xrightarrow{\alpha}_{(\Sigma;\mathcal{R})}$ is defined in fig. 5. We write $\Gamma \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})}$ iff $\exists\, \Gamma' : \Gamma \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'$. We define two **context reductions** $\rightarrow_{(\Sigma;\mathcal{R})}$ and $\rightarrow_{\Sigma}$ as:*

$$\Gamma \rightarrow_{(\Sigma;\mathcal{R})} \Gamma' \text{ holds iff } \Gamma \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'$$

$$\Gamma \rightarrow_{\Sigma} \Gamma' \text{ holds iff } \Gamma \xrightarrow{\alpha}_{\Sigma} \Gamma' \text{ for } \alpha \in \{s[\mathbf{p}]\,!\,\mathbf{q} : \mathsf{m}(\mathbb{T}),\ s[\mathbf{p}][\mathbf{q}] : \mathsf{m}\}$$

We write $\rightarrow^{+}_{(\Sigma;\mathcal{R})}$ (resp. $\rightarrow^{+}_{\Sigma}$) and $\rightarrow^{*}_{(\Sigma;\mathcal{R})}$ (resp. $\rightarrow^{*}_{\Sigma}$) for their transitive and reflexive/transitive closures respectively.

A context Γ keeps track of open buffers using a *buffer-tracker* Σ. Whenever a new session is initialised, it is added to Σ, details in § 3.4 item [T-ν]. For now it suffices to know that buffer trackers restrict communication to occur only over restricted sessions, thus by def. 5 (well-formedness), it guarantees that a session buffer exists for all sessions in Σ.

Context reduction (def. 9) models communication at the *type-level*. Context Γ can reduce by sending, communicating, or timing out. By [Γ-$\textcircled{t}$], $\Gamma = s[\mathbf{p}] : \&_{i\in I}\{\mathbf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ \textcircled{t}.\,\mathbb{S}'\}$ can reduce to a *timeout* branch continuation type $\mathbb{S}'$ if s is in the buffer-tracker (*i.e.,* a buffer exists for session s), and *at least one*

$$[\Gamma\text{-}\textcircled{1}]\quad \frac{s \in \Sigma \qquad \exists\, k \in I : \mathsf{q}_k \notin \mathcal{R}(\mathsf{p})}{s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ \textcircled{1}.\mathbb{S}'\} \xrightarrow{s[\mathsf{p}]\,\textcircled{1}}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : \mathbb{S}'}$$

$$[\Gamma\text{-Snd}_1]\quad \frac{s \in \Sigma \qquad k \in I}{s[\mathsf{p}] : \oplus_{i\in I}\{\mathsf{q}_i \,!\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\} \xrightarrow{s[\mathsf{p}]\,!\,\mathsf{q}_k:\mathsf{m}_k(\mathbb{T}_k)}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : (\mathsf{q}_k \,!\, \mathsf{m}_k(\mathbb{T}_k)\cdot\epsilon\,;\,\mathbb{S}_k)}$$

$$[\Gamma\text{-Snd}_2]\quad \frac{s \in \Sigma \qquad k \in I}{s[\mathsf{p}] : (\mathbb{M}\,;\,\oplus_{i\in I}\{\mathsf{q}_i \,!\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}) \xrightarrow{s[\mathsf{p}]\,!\,\mathsf{q}_k:\mathsf{m}_k(\mathbb{T}_k)}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : (\mathbb{M}\cdot\mathsf{q}_k \,!\, \mathsf{m}_k(\mathbb{T}_k)\cdot\epsilon\,;\,\mathbb{S}_k)}$$

$$[\Gamma\text{-Com}]\quad \frac{s \in \Sigma \qquad \exists\, k \in I : (\mathsf{p},\mathsf{m},\mathbb{T}) = (\mathsf{p}_k,\mathsf{m}_k,\mathbb{T}_k)}{s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M},\ \ s[\mathsf{q}] : \&_{i\in I}\{\mathsf{p}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\ [,\textcircled{1}.\mathbb{S}']\} \xrightarrow{s[\mathsf{p}][\mathsf{q}]:\mathsf{m}}_{(\Sigma;\mathcal{R})} s[\mathsf{p}] : \mathbb{M},\ \ s[\mathsf{q}] : \mathbb{S}_k}$$

$$[\Gamma\text{-}\mu]\quad \frac{s[\mathsf{p}] : \mathbb{S}[^{\mu t.\mathbb{S}}/_t] \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'}{s[\mathsf{p}] : \mu t.\mathbb{S} \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma'} \qquad [\Gamma\text{-Cong}]\quad \frac{\Gamma_1 \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma_2}{\Gamma,\Gamma_1 \xrightarrow{\alpha}_{(\Sigma;\mathcal{R})} \Gamma,\Gamma_2}$$

Fig. 5. Context reduction rules

of the roles in the branch is *unreliable*. The latter prevents taking a timeout for communication that is sure to be delivered. Reductions [Γ-Snd$_1$] and [Γ-Snd$_2$] simulate sending a message by reducing the selection type $\oplus_{i\in I}\{\mathsf{q}_i \,!\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$ to one of its continuations $\mathbb{S}_i$, and by inserting the sent message into the buffer type. The difference is that [Γ-Snd$_1$] creates the buffer type if it was previously not specified, whereas [Γ-Snd$_2$] appends the message to an already existing buffer type. Communication between two roles is simulated through [Γ-Com], where a branch type $s[\mathsf{q}] : \&_{i\in I}\{\mathsf{p}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\ [,\textcircled{1}.\mathbb{S}']\}$ consumes the message from a buffer type $s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M}$, reducing to the continuations $s[\mathsf{p}] : \mathbb{M},\ s[\mathsf{q}] : \mathbb{S}_k$. Lastly, [$\Gamma$-$\mu$] allows reduction through recursion and [Γ-Cong] reduces substructures of compatibly composed contexts.

Definition 10. *Property φ_{s} is a $(\Sigma;\mathcal{R})$-**safety** property on typing contexts iff:*

[S-$\mathcal{R}_1$] $\varphi_{\mathsf{s}}(\Gamma,\ s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}) \implies \forall i \in I : \mathsf{q}_i \in \mathcal{R}(\mathsf{p})$

[S-$\mathcal{R}_2$] $\varphi_{\mathsf{s}}(\Gamma,\ s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ \textcircled{1}.\mathbb{S}'\}) \implies \exists i \in I : \mathsf{q}_i \notin \mathcal{R}(\mathsf{p})$

[S-Com] $\varphi_{\mathsf{s}}(\Gamma,\ s[\mathsf{p}] : \&_{i\in I}\{\mathsf{q}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\ [,\textcircled{1}.\mathbb{S}']\},\ s[\mathsf{q}] : \mathbb{M})$
and $\mathbb{M} \equiv \mathsf{p}\,!\,\mathsf{m}(\mathbb{T})\cdot\mathbb{M}'$
and $\exists\, k \in I : \mathsf{q}_k = \mathsf{q} \wedge \mathsf{m}_k = \mathsf{m} \implies \mathbb{T}_k = \mathbb{T}$

[S-μ] $\varphi_{\mathsf{s}}(\Gamma,\ s[\mathsf{p}] : \mu t.\mathbb{S}) \implies \varphi_{\mathsf{s}}(\Gamma,\ s[\mathsf{p}] : \mathbb{S}[^{\mu t.\mathbb{S}}/_t])$

[S-$\to$] $\varphi_{\mathsf{s}}(\Gamma)$ and $\Gamma \to_{(\Sigma;\mathcal{R})} \Gamma' \implies \varphi_{\mathsf{s}}(\Gamma')$

As previously mentioned, our type system is a *generic* one that does not use syntactic methods of enforcing consistent communication. Therefore, we define a *safety* property in def. 10 on type contexts that is used to guarantee subject reduction and other theorems (presented in § 4).

We say $\varphi_{\mathtt{s}}$ is the *largest* safety property required to guarantee subject reduction. The property can be re-instantiated with more specific conditions (as demonstrated in § 5) as per the requirements of the implementation. Concretely, [S-$\mathcal{R}_1$] and [S-$\mathcal{R}_2$] ensure that timeouts are only not defined if communication is reliable and that timeouts are defined if communication is unreliable respectively. Condition [S-Com] ensures that communicating messages have matching payload types. Lastly, [S-μ] preserves $\varphi_{\mathtt{s}}$ through recursion unfolding and [S-$\rightarrow$] requires safety to hold after context reduction.

3.4 Typing Rules

Our type system is defined by the typing rules in fig. 6. Below we explain them in detail. Typing judgements are of the form: $\Theta \cdot \Gamma \vdash P$ reading "process P is well typed under type contexts Θ and Γ"; and $\Gamma \vdash d : \mathbb{T}$ reading "value (or variable, or channel) d is of type $\mathbb{T}$ under type context Γ".

[T-**0**] The inaction process **0** is typed by a context that is "`end` typed", determined by the predicate $\mathtt{end}(\Gamma)$—defined in fig. 7. The predicate holds: (*i*) if $\Gamma = \emptyset$; (*ii*) if Γ consists of variables, then it holds if all the variables are either of a `basic` type, or can be typed by **end**; and (*iii*) if Γ consists of sessions with roles, then it holds if all the channels can be typed by **end**.

[T-Var] A variable or session with role c has type $\mathbb{T}$ in a context only containing the mapping of c to $\mathbb{T}$.

[T-Val] A value v is typed by a `basic` type $\mathbb{B}$ if v is contained in the set of that `basic` type. *E.g.*, $42 : \mathbb{N}$ is typed under an empty context $\emptyset$ since $42 \in \mathbb{N}$.

[T-X] A process variable X is typed to an n-tuple of types $\mathbb{T}_1, \ldots, \mathbb{T}_n$ under context Θ, if Θ maps the process variable to the same n-tuple of types.

[T-$\oplus$] The *selection* process $c \oplus [\mathbf{q}_k] \,!\, \mathsf{m}_k\langle d \rangle . P$ is typed under a context which maps the sending channel c to a selection session type $\oplus_{i \in I}\{\mathbf{q}_i \,!\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$, where a selection option matches the send process, *i.e.*, $k \in I$. The context should match the payload d to the type indicated in the selection ($\mathbb{T}_k$), and continuation process P should be typed under the continuation type $\mathbb{S}_k$.

[T-&] The *branching* process $c \,\&_{i \in I}\{[\mathbf{p}_i] \,?\, \mathsf{m}_i(x_i).P_i\}$ is typed under a context which maps the receiving channel c to a branch type $\&_{i \in I}\{\mathbf{p}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\}$, where all roles and message labels of each branch match. Every continuation process P_i must be typed under the continuation type $\mathbb{S}_i$ and payload typed by $\mathbb{T}_i$. If the process is a *timeout* branch $c \,\&_{i \in I}\{[\mathbf{p}_i] \,?\, \mathsf{m}_i(x_i).P_i,\ (t).Q\}$, then it should be typed by a session type also containing a timeout continuation $\&_{i \in I}\{\mathbf{p}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i,\ (t).\mathbb{S}'\}$, and the timeout process Q must be typed by $\mathbb{S}'$.

[T-Call] A process *call* $X\langle d_1, \ldots, d_n \rangle$ is correctly typed if Θ types the process variable to a n-tuple of types $\mathbb{T}_1, \ldots, \mathbb{T}_n$ and Γ maps each parameter d_i to the corresponding $\mathbb{T}_i$ (for $i \in 1..n$). Any remaining channels in Γ cannot be open, and hence must be `end` typed.

[T-**0**]
$$\frac{\mathsf{end}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}}$$

[T-Var]
$$\frac{}{c : \mathbb{T} \vdash c : \mathbb{T}}$$

[T-Val]
$$\frac{v \in \mathbb{B}}{\emptyset \vdash v : \mathbb{B}}$$

[T-X]
$$\frac{\Theta(X) = \mathbb{T}_1, \ldots, \mathbb{T}_n}{\Theta \vdash X : \mathbb{T}_1, \ldots, \mathbb{T}_n}$$

[T-$\oplus$]
$$\frac{\Gamma_1 \vdash c : \oplus_{i \in I}\{\mathbf{q}_i \,!\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\} \qquad k \in I \qquad \Gamma_2 \vdash d : \mathbb{T}_k \qquad \Theta \cdot \Gamma, c : \mathbb{S}_k \vdash P}{\Theta \cdot \Gamma, \Gamma_1, \Gamma_2 \vdash c \oplus [\mathbf{q}_k] \,!\, \mathsf{m}_k\langle d \rangle . P}$$

[T- &]
$$\frac{\Gamma' \vdash c : \&_{i \in I}\{\mathbf{p}_i \,?\, \mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i \,[,\ \textcircled{t}.\mathbb{S}']\} \qquad [\Theta \cdot \Gamma, c : \mathbb{S}' \vdash Q] \qquad \forall i \in I \cdot \Theta \cdot \Gamma, x_i : \mathbb{T}_i, c : \mathbb{S}_i \vdash P_i}{\Theta \cdot \Gamma, \Gamma' \vdash c \,\&_{i \in I}\{[\mathbf{p}_i] \,?\, \mathsf{m}_i(x_i).P_i \,[,\ \textcircled{t}.Q]\}}$$

[T-Call]
$$\frac{\Theta \vdash X : \mathbb{T}_1, \ldots, \mathbb{T}_n \qquad \mathsf{end}(\Gamma') \qquad \forall i \in 1..n \cdot \Gamma_i \vdash d_i : \mathbb{T}_i}{\Theta \cdot \Gamma_1, \ldots, \Gamma_n, \Gamma' \vdash X\langle d_1, \ldots, d_n \rangle}$$

[T-Def]
$$\frac{\Theta, X : \mathbb{T}_1, \ldots, \mathbb{T}_n \cdot x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n \vdash P \qquad \Theta, X : \mathbb{T}_1, \ldots, \mathbb{T}_n \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \mathsf{def}\ X(x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n) = P\ \mathsf{in}\ Q}$$

[T-+]
$$\frac{\Theta \cdot \Gamma \vdash P_1 \qquad \Theta \cdot \Gamma \vdash P_2}{\Theta \cdot \Gamma \vdash P_1 + P_2}$$

[T-Lift]
$$\frac{\Theta \cdot \Gamma \vdash P}{\Theta \cdot \Gamma \vdash_{\emptyset} P}$$

[T-ϵ]
$$\frac{\mathsf{gc}(\Gamma)}{\Theta \cdot \Gamma \vdash_{\{s\}} s : \epsilon}$$

[T-σ_1]
$$\frac{\Theta \cdot \Gamma' \vdash_{\{s\}} s : \sigma \qquad \Gamma \vdash w : \mathbb{T}}{\Theta \cdot \Gamma, \Gamma', s[\mathbf{p}] : \mathbf{q} \,!\, \mathsf{m}(\mathbb{T}) \cdot \epsilon \vdash_{\{s\}} s : (\mathbf{p} \triangleright \mathbf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma}$$

[T-σ_2]
$$\frac{\Theta \cdot \Gamma', s[\mathbf{p}] : \mathbb{M} \vdash_{\{s\}} s : \sigma \qquad \Gamma \vdash w : \mathbb{T}}{\Theta \cdot \Gamma, \Gamma', s[\mathbf{p}] : \mathbf{q} \,!\, \mathsf{m}(\mathbb{T}) \cdot \mathbb{M} \vdash_{\{s\}} s : (\mathbf{p} \triangleright \mathbf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma}$$

[T-σ_w]
$$\frac{\Gamma = (\Gamma_0 \rightsquigarrow \Gamma_1), \Gamma_2 \qquad \Theta \cdot \Gamma_1 \vdash_{\Sigma} s : \sigma \qquad \mathsf{gc}(\Gamma_0, \Gamma_2)}{\Theta \cdot \Gamma \vdash_{\Sigma} s : \sigma}$$

[T-|]
$$\frac{\Theta \cdot \Gamma_1 \vdash_{\Sigma_1} P_1 \qquad \Theta \cdot \Gamma_2 \vdash_{\Sigma_2} P_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Theta \cdot \Gamma_1, \Gamma_2 \vdash_{\Sigma_1 \cup \Sigma_2} P_1 \mid P_2}$$

[T-ν]
$$\frac{\Gamma' = \{s[\mathbf{p}] : \tau_{\mathbf{p}}\}_{\mathbf{p} \in \boldsymbol{\rho}} \qquad s \notin \Gamma \qquad (\{s\}; \mathcal{R})\text{-}\varphi_{\mathbf{s}}(\Gamma') \qquad \Theta \cdot \Gamma, \Gamma' \vdash_{\Sigma} P}{\Theta \cdot \Gamma \vdash_{\Sigma \setminus \{s\}} (\nu s : \Gamma')\, P}$$

Fig. 6. Typing rules

$$\frac{}{\mathtt{end}(\emptyset)} \qquad \frac{\forall i \in 1..n \;\cdot\; \mathtt{basic}(\mathbb{T}_i) \;\;\vee\;\; x_i : \mathbb{T}_i \;\vdash\; x_i : \mathbf{end}}{\mathtt{end}(x : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n)}$$

$$\frac{\mathtt{end}(\Gamma_1) \qquad \mathtt{end}(\Gamma_2)}{\mathtt{end}(\Gamma_1, \Gamma_2)} \qquad \frac{\forall i \in 1..n, \mathsf{p} \in \boldsymbol{\rho} \;\cdot\; s_i[\mathsf{p}] : \tau_i \;\vdash\; s_i[\mathsf{p}] : \mathbf{end}}{\mathtt{end}(s_i[\mathsf{p}] : \tau_1, \ldots, s_i[\mathsf{p}] : \tau_n)}$$

Fig. 7. Predicate $\mathtt{end}(\Gamma)$

$$\frac{}{\mathtt{gc}(\emptyset)} \qquad \frac{\mathtt{gc}(\Gamma)}{\mathtt{gc}(\Gamma, s[\mathsf{p}] : \epsilon)} \qquad \frac{\mathtt{basic}(\mathbb{T}) \qquad \mathtt{gc}(\Gamma, s[\mathsf{p}] : \mathbb{M})}{\mathtt{gc}(\Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M})}$$

$$\frac{\Gamma = \Gamma', s'[\mathsf{p}'] : \mathbb{T} \qquad \mathtt{gc}(\Gamma', s[\mathsf{p}] : \mathbb{M})}{\mathtt{gc}(\Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M})}$$

Fig. 8. The garbage collector predicate $\mathtt{gc}(\Gamma)$

$$s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \epsilon \rightsquigarrow \Gamma, s[\mathsf{p}] : \mathbb{M} = \Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M}$$
$$s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \epsilon \rightsquigarrow \Gamma \;\; \textit{when} \;\; s[\mathsf{p}] : \mathbb{M} \notin \Gamma = \Gamma, s[\mathsf{p}] : \mathsf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \epsilon$$

Fig. 9. Message insertion function $\Gamma' \rightsquigarrow \Gamma$

[T-Def] Process *declaration* $X(x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n) = P$ is well typed if P is self-contained, *i.e.,* contexts containing the types of the declaration parameters (along with any previous Θ) should type P. Process *definition* $\mathsf{def}\ X(x_1 : \mathbb{T}_1, \ldots, x_n : \mathbb{T}_n) = P\ \mathsf{in}\ Q$ is typed under $\Theta \cdot \Gamma$ if its declaration is well typed and Q is typed under Γ and Θ composed with the new process variable.

[T-+] Non-deterministic choice is well typed if processes are typed by $\Theta \cdot \Gamma$ in isolation. This is in line with how *case* or *if-then-else* processes are typed.

[T-Lift] We annotate the typing judgement $\Theta \cdot \Gamma \vdash P$ with the buffer-tracker to obtain $\Theta \cdot \Gamma \;\vdash_\Sigma P$, denoting that the sessions in Σ occur in P. The lifting rule annotates the typing judgement with an empty buffer-tracker if the buffer-less judgement ($\vdash$) types P (using the rules mentioned thus far).

[T-ϵ] In standard asynchronous MPST theory, the empty buffer $s : \epsilon$ is typed under the empty context $\emptyset$, ensuring a one-to-one correlation between buffer types in the context and messages in a session buffer. However, since our calculus *models message loss*, it is possible that a context contains buffer types for messages that have been dropped from the process buffer. Thus, our theory types $s : \epsilon$ under a *garbage collected* Γ. The predicate $\mathtt{gc}$ is defined in fig. 8, and states that valid leftover types $\mathtt{gc}(\Gamma)$ are: (*i*) empty; (*ii*) empty buffer types; (*iii*) message buffer types with basic-type payloads; or (*iv*) message buffer types with channel payloads that are typed under Γ.

[T-σ_1] [T-σ_2] An entry in a session buffer $s : (\mathbf{p} \triangleright \mathbf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma$ is typed under a context containing a mapping from $s[\mathbf{p}]$ to a buffer type $\mathbf{q}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M}$, matching the recipient and message label. The message payload w must be of type $\mathbb{T}$, indicated by the buffer type, and buffer continuation $s : \sigma$ should be typed under the buffer continuation type $\mathbb{M}$ in the case that it is not empty ([T-σ_2]).

[T-σ_w] Weakening allows a session buffer to be typed under a larger context if the addition can be garbage collected and inserted into the original context using the message insertion function (fig. 9). This is partial function that either appends a message to an existing buffer type, or inserts it as the head of a new buffer type. Put differently, weakening allows a buffer to be typed under a larger context containing irrelevant types that can be garbage collected.

[T-|] If a process P_1 is typed by Γ_1, and process P_2 is typed by Γ_2, then the composition Γ_1,Γ_2 types the *parallel* composition $P_1 \mid P_2$. It is also required that parallel processes *cannot* each contain a buffer for the same session s. This guarantees the uniqueness of one session buffer per restricted session.

[T-ν] Session restriction $(\nu s : \Gamma')\, P$ requires session s to be instantiated with a Γ' mapping each session with role to its session-buffer type. $\varphi_{\mathbf{s}}(\Gamma')$ must hold to ensure subject reduction, as discussed in § 3.3. Session s should not be present in a previous context Γ, and process P should be typed under the composition of the previous and newly instantiated context with the updated buffer-tracker $\Theta \cdot \Gamma, \Gamma' \vdash_{\Sigma} P$ (since the buffer for s is contained in P).

Example 4 (Ping Pong: Type Context). Recalling the ping pong example, the whole system can then be described by a parallel composition of the three processes representing each role $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$ together with an empty buffer, which is closed under a type context Γ with the following typing assumptions.

$$\Gamma = \{s[\mathbf{p}] : \mathbb{S}_{\mathbf{p}},\ s[\mathbf{q}] : \mathbb{S}_{\mathbf{q}}, s[\mathbf{r}] : \mathbb{S}_{\mathbf{r}}\}$$

$$P_{ping} = (\nu s : \Gamma)\ P_{\mathbf{p}} \mid P_{\mathbf{q}} \mid P_{\mathbf{r}} \mid s : \epsilon$$

4 Type Properties

The main results of our MPST system for MAGπ processes are *subject reduction* (theorem 1) and *session fidelity* (theorem 2). It is key to note that our results are parametric on the reliability function $\mathcal{R}$. Thus, the theorems we present hold for *any* configuration of reliability, *i.e.,* from *no* reliable communication all the way to *completely* reliable networks.

In order to synchronise reliability assumptions between types and processes, we define the *reliable process reduction* $\longrightarrow_{\mathcal{R}}$, such that $\longrightarrow_{\mathcal{R}} \subseteq \longrightarrow$.

Definition 11 (Reliable process reduction). *The reliable process reduction $\longrightarrow_{\mathcal{R}}$ is inductively defined by the same reduction rules for $\longrightarrow$ (in fig. 2), with the following changes* [3]*:*

$$\textbf{[R-}\clockface\textbf{]} \qquad s[\mathbf{q}]\,\&_{i\in I}\{[\mathbf{p}_i]\,?\,\mathsf{m}_i(x_i).P_i, \clockface.\,Q\} \mid s : \sigma \longrightarrow_{\mathcal{R}} Q \mid s : \sigma \qquad \text{if } \exists k \in I : \mathbf{p}_k \notin \mathcal{R}(\mathbf{q})$$

$$\textbf{[R-}\downarrow\textbf{]} \qquad s : (\mathbf{p} \triangleright \mathbf{q} \triangleleft \mathsf{m}\langle w \rangle) \cdot \sigma \longrightarrow_{\mathcal{R}} s : \sigma \qquad \text{for } \mathbf{q} \notin \mathcal{R}(\mathbf{p})$$

[3] For a fully unreliable network, *i.e.,* $\forall \mathbf{p} \in \boldsymbol{\rho} \cdot \mathcal{R}(\mathbf{p}) = \emptyset$, $\longrightarrow_{\mathcal{R}}$ is equivalent to $\longrightarrow$.

Intuitively, the reliable process reduction disregards network faults for reliable communication. Concretely, a timeout reduction [**R-**$\textcircled{\scriptsize t}$] is only possible if *at least one role* in the branch is unreliable; and message loss [**R-**$\downarrow$] can only occur for messages that are *not* reliable from the viewpoint of the sender. This ensures that no messages are ignored or lost for reliable communication. Proofs of our theorems, along with any auxiliary results, are given in the technical report [23].

4.1 Subject Reduction

Using $\longrightarrow_{\mathcal{R}}$, we now present our result of *subject reduction*. Intuitively, subject reduction states that, if a process P is typed under a safe context, and P reliably reduces to some process P', then the context also reduces (in 0 or 1 steps) to a safe context, which types the new process P'.

Theorem 1 (Subject Reduction).

$$\Theta \cdot \Gamma \vdash_{\Sigma} P \;\; \textit{and} \;\; (\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma) \;\; \textit{and} \;\; P \rightarrow_{\mathcal{R}} P' \implies \exists \Gamma' : \Gamma \rightarrow^{\{0,1\}}_{(\Sigma;\mathcal{R})} \Gamma' \;\; \textit{and} \;\; (\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma') \;\; \textit{and} \;\; \Theta \cdot \Gamma' \vdash_{\Sigma} P'$$

A key novel result of our type system is that no unexpected network failures can occur at runtime, *i.e.,* a process always has a failure-handling subprotocol defined for unreliable communication. This follows from the definition of our safety property φ_{s} (def. 10) and holds through subject reduction. We state the result in cor. 1. More precisely, this corollary states that timeout branches are guaranteed to be defined for unreliable communication. The inverse is stated in cor. 2, *i.e.,* timeouts are not defined for branches containing *only* reliable sources.

Corollary 1 (Failure handling safety). *Given a reliability function* $\mathcal{R}$: $\mathsf{p} \notin \mathcal{R}(\mathsf{q})$ *and* $\Theta \cdot \Gamma \vdash_{\Sigma} P$ *with* $(\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma)$ *and* $P \longrightarrow^{*}_{\mathcal{R}} P' \equiv \mathbb{C}[Q]$ *implies* $Q \neq s[\mathsf{q}]\,\&_{i\in I}\{\ldots, [\mathsf{p}]\,?\,\mathsf{m}(x).Q'\}$. *I.e., Q cannot be a branch at* q *receiving from* p *and not define a timeout.*

Corollary 2 (Reliability adherence). *Given a reliability function* $\mathcal{R}$: $\mathcal{R}(\mathsf{q}) = \mathfrak{R}_{\mathsf{q}}$ *and* $\Theta \cdot \Gamma \vdash_{\Sigma} P$ *with* $(\Sigma;\mathcal{R})\text{-}\varphi_{\mathsf{s}}(\Gamma)$ *and* $P \longrightarrow^{*}_{\mathcal{R}} P' \equiv \mathbb{C}[Q]$ *implies* $Q \neq s[\mathsf{q}]\,\&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(x_i).Q_i,\ \textcircled{\scriptsize t}.Q'\}$ st: $\forall i \in I : \mathsf{p}_i \in \mathfrak{R}_{\mathsf{q}}$. *I.e., Q cannot be a branch at* q *only receiving from reliable roles* p_i *and define a timeout.*

4.2 Session Fidelity

Session fidelity states the opposite implication of subject reduction, *i.e.,* if Γ types a process P, and Γ can reduce, then P can match at least one of the context reductions.

Consequently, relevant properties of process P can be deduced from the behaviour of its type context Γ (as we will see in theorem 3). However, as shown by Scalas and Yoshida [33, sec. 5.2], the result does *not* hold for all well-typed

processes. Concretely, session fidelity is violated by: (i) processes that recurse infinitely without being productive (*e.g.* $\mathsf{def}\ X(x) = X\langle x\rangle\ \mathsf{in}\ X\langle s[\mathsf{p}]\rangle$); and (ii) processes that deadlock by interleaving communication across multiparty sessions. Hence, we assume the necessary conditions on processes to restrict the aforementioned violations, by adapting [33, def. 5.3].

Definition 12 (Conditions for session fidelity). *Assuming* $\emptyset \cdot \Gamma \vdash_{\{s\}} P$. *We say that* P*:*

1. ***has guarded definitions*** *iff each process definition in* P *of the form*

$$\mathsf{def}\ X(x_1 : \mathbb{T}, \ldots, x_n : \mathbb{T}) = Q\ \mathsf{in}\ P'$$

$\forall j \in 1..n$: if $\mathbb{T}_j$ is a session type, then a process call $Y\langle \ldots, x_j, \ldots\rangle$ can only occur in Q as a subterm of

$$x_j\,\&_{i\in I}\{[\mathsf{p}_i]\,?\,\mathsf{m}_i(y_i).P_i[,\ \mathrm{t}.\,P_t]\}\quad\text{or}\quad x_j \oplus [\mathsf{p}]\,!\,\mathsf{m}\langle y\rangle.\,P'',$$

i.e., after x_j is used for input or output.

2. ***only plays role*** p ***in*** s***, by*** Γ *iff:* ***(i)*** P *has guarded definitions (from 1);* ***(ii)*** $\mathtt{fv}(P) = \emptyset$*;* ***(iii)*** $\Gamma = \Gamma_0, s[\mathsf{p}] : \tau$ *with* $\tau \neq \mathbf{end}$ *and* $\mathtt{end}(\Gamma_0)$*; and* ***(iv)*** *for all* $(\nu s' : \Gamma')\,P'$ *subterm of* P, $\mathtt{end}(\Gamma')$.

We say "P only plays role p in s" iff $\exists \Gamma : \emptyset \cdot \Gamma \vdash_{\{s\}} P$ *and condition 2 holds.*

Def. 12 formalises guarded recursion in condition 1, and the notion of only playing a single role for a given session in condition 2. Together, these conditions ensure that session fidelity, stated in theorem 2, holds for all well-typed processes.

Theorem 2 (Session Fidelity). *Assuming* $\emptyset \cdot \Gamma \vdash_\Sigma P$ *with* $(\Sigma;\mathcal{R})$-$\varphi_{\mathsf{s}}(\Gamma)$, $P \equiv (\Pi_{\mathsf{p}\in I}\, P_{\mathsf{p}}) \,|\, s : \sigma$ *and* $\Gamma = \bigcup_{\mathsf{p}\in I} \Gamma_{\mathsf{p}}$*, and for each* P_{p}*: (i)* $\emptyset \cdot \Gamma_{\mathsf{p}} \vdash_\Sigma P_{\mathsf{p}}$*, and (ii)* P_{p} *being* $\mathbf{0}$ *(up-to-*$\equiv$*)* or *only plays role* p *in* s*, by* Γ_{p}*. Then,*

$\Gamma \longrightarrow_{(\Sigma;\mathcal{R})}$ *implies* $\exists \Gamma', P'$*: (i)* $\Gamma \longrightarrow_{(\Sigma;\mathcal{R})} \Gamma'$*, (ii)* $P \longrightarrow_{\mathcal{R}}^{+} P'$*, (iii)* $\emptyset \cdot \Gamma' \vdash_\Sigma P'$ *with* $(\Sigma;\mathcal{R})$-$\varphi_{\mathsf{s}}(\Gamma')$*, (iv)* $P' = (\Pi_{\mathsf{p}\in I}\, P'_{\mathsf{p}}) \,|\, s : \sigma'$ *and* $\Gamma' = \bigcup_{\mathsf{p}\in I} \Gamma'_{\mathsf{p}}$*, and (v) for each* P'_{p}*:* $\emptyset \cdot \Gamma'_{\mathsf{p}} \vdash_\Sigma P'_{\mathsf{p}}$*, and* P'_{p} *is* $\mathbf{0}$ *(up-to-*$\equiv$*)* or *only plays role* p *in* s*, by* Γ'_{p}.

4.3 Process Properties

Our result of session fidelity (§ 4.2) allows us to infer runtime properties about programs in MAGπ from their types. We proceed by defining desirable runtime properties on processes (def. 13); expressing the equivalence of these properties at type-level (def. 14); and presenting our result of *process properties verification* (theorem 3), linking process properties to their type-level equivalences.

From def. 13 below, a process is: **(i)** $\mathcal{R}_F$*-communication-safe* (new w.r.t.[33]) if it reaches the end of communication over reliable reductions and has *no leftover messages* in the buffer; **(ii)** *deadlock-free* if it either reduces or it is inaction; **(iii)** *terminating* if it is deadlock free and can reach inaction in a finite number

of steps; **(iv)** *never-terminating* if it can always infinitely reduce; and **(v)** *live* if, for every reliable branch it can reduce to, it can eventually reduce to some branch continuation. We need not consider branches with timeouts since these are trivially live, given that a process can always reduce over the timeout.

Definition 13 (Process properties, adapted from [33]**).** *For some reliability function $\mathcal{R}$, and full reliability function $\mathcal{R}_F$, a process P is said to be:*

(i) ***$\mathcal{R}_F$-communication-safe*** *iff*

$$P \longrightarrow^*_{\mathcal{R}_F} P' \not\longrightarrow_{\mathcal{R}_F} \quad \textit{and} \quad P' = \mathbb{C}[s : \sigma] \quad \textit{implies} \quad \sigma = \epsilon;$$

(ii) ***deadlock-free*** *iff* $P \longrightarrow^*_{\mathcal{R}} P' \not\longrightarrow_{\mathcal{R}}$ *implies* $P' \equiv \mathbf{0}$;

(iii) ***terminating*** *iff it is deadlock free, and*

$$\exists\, i \textit{ finite}\ \text{st:}\ \forall n \geq i : P = P_0 \longrightarrow_{\mathcal{R}} P_1 \longrightarrow_{\mathcal{R}} \cdots \longrightarrow_{\mathcal{R}} P_n \quad \textit{implies} \quad P_n \not\longrightarrow_{\mathcal{R}};$$

(iv) ***never-terminating*** *iff* $P \longrightarrow^*_{\mathcal{R}} P'$ *implies* $P' \longrightarrow_{\mathcal{R}}$;

(v) ***live*** *iff* $P \longrightarrow^*_{\mathcal{R}} P' \equiv \mathbb{C}[Q]$ *implies*

$$\textit{if} \quad Q = c\,\&_{i\in I}\{[\mathbf{q}_i]\,?\,\mathsf{m}_i(x_i).Q'_i\}, \quad \textit{then} \quad \exists\, \mathbb{C}', k \in I, w : P' \longrightarrow^*_{\mathcal{R}} \mathbb{C}'[\,Q'_k[{}^{w}/_{x_k}]\,].$$

Note that, differently from other works [4,33], our definition of liveness only speaks about receiving processes, and not sending. Typically, liveness also requires that a sent message—in the case of MAGπ, any message in a session buffer—is always eventually consumed. However, because of the failures that our calculus models, it is possible that a process is live and still have unconsumed messages in the buffer (*e.g.*, as a result of timing out due to a message delay). Additionally, for a *$\mathcal{R}_F$-communication-safe* process it follows that all sent messages are consumed in the reliable case. Hence, the traditional definition of liveness still holds for reliable network configurations, and our new definition provides the largest guarantees possible given the failure assumptions.

We now present the type-level equivalences of the above process properties. For liveness, we generalise to the largest liveness property, as done with safety in def. 10, allowing users to define more fine-grained notions of liveness, if required.

From def. 14 below, a type context is: **(i)** *$\mathcal{R}_F$-communication-safe* if it has no populated buffer types when it can no longer reliably reduce; **(ii)** *deadlock-free* if the reason why it can no longer reduce is because it is `end` typed (and possibly, as a result of network failures, has some leftover types that can be garbage collected); **(iii)** *terminating* if it is deadlock free and can reach the end of the protocol in a finite number of steps; **(iv)** *never-terminating* if it can always infinitely reduce; and **(v)** *live* if, for every reliable branch it can reduce to, there is a series of steps that can reduce to a continuation of that branch.

Definition 14 (Type context properties). *For some reliability function $\mathcal{R}$, a full reliability function $\mathcal{R}_F$, and a set of sessions Σ, we say context Γ is:*

(i) $(\Sigma;\mathcal{R}_F)$***-communication-safe*** *iff*

$$\Gamma \longrightarrow^*_{(\Sigma;\mathcal{R}_F)} \Gamma' \not\longrightarrow_{(\Sigma;\mathcal{R}_F)} \text{ and } s[\mathtt{p}] : \mathbb{M} \in \Gamma' \text{ implies } \mathbb{M} = \epsilon;$$

(ii) $(\Sigma;\mathcal{R})$***-deadlock-free*** *iff*

$$\Gamma \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma' \not\longrightarrow_{(\Sigma;\mathcal{R})} \quad \text{implies} \quad \Gamma' = \Gamma'_0, \Gamma'' \text{ st: } \mathtt{end}(\Gamma'_0) \text{ and } \mathtt{gc}(\Gamma'');$$

(iii) $(\Sigma;\mathcal{R})$***-terminating*** *iff it is* $(\Sigma;\mathcal{R})$*-deadlock-free, and* $\exists\, i$ *finite* st:

$$\forall n \geq i : \Gamma = \Gamma_0 \longrightarrow_{(\Sigma;\mathcal{R})} \Gamma_1 \longrightarrow_{(\Sigma;\mathcal{R})} \cdots \longrightarrow_{(\Sigma;\mathcal{R})} \Gamma_n \text{ implies } \Gamma_n \not\longrightarrow_{(\Sigma;\mathcal{R})};$$

(iv) $(\Sigma;\mathcal{R})$***-never-terminating*** *iff* $\Gamma \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma' \quad \text{implies} \quad \Gamma' \longrightarrow_{(\Sigma;\mathcal{R})}$;

(v) $(\Sigma;\mathcal{R})$***-live*** *iff it obeys some liveness property* $(\Sigma;\mathcal{R})$-$\varphi_{\mathtt{L}}$ st:

$$\begin{array}{l}
(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma,\ s[\mathtt{p}] : \mathbb{S}) \text{ and } \mathbb{S} = \&_{i\in I}\{\mathtt{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\} \\
\qquad\qquad \Longrightarrow\ \ \exists \Gamma', k \in I : \Gamma, s[\mathtt{p}] : \mathbb{S} \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma', s[\mathtt{p}] : \mathbb{S}_k \\
(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma,\ s[\mathtt{p}] : \mu \mathrm{t}.\mathbb{S}) \implies (\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma,\ s[\mathtt{p}] : \mathbb{S}[^{\mu \mathrm{t}.\mathbb{S}}/_{\mathrm{t}}]) \\
(\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma) \text{ and } \Gamma \rightarrow_{(\Sigma;\mathcal{R})} \Gamma' \implies (\Sigma;\mathcal{R})\text{-}\varphi_{\mathtt{L}}(\Gamma)
\end{array}$$

We are now ready to use these type-level equivalent properties to infer behaviours of the processes they type. We present our result in theorem 3 which formally states that, under the same assumptions given in session fidelity (theorem 2), if a process is typed under some type context, and a property holds on that context, then the same property holds for the process itself.

Theorem 3 (Process properties verification). *Assuming:* $\emptyset \cdot \Gamma \vdash_\Sigma P$ *with* $(\Sigma;\mathcal{R})$-$\varphi_{\mathtt{s}}(\Gamma)$, $P \equiv (\Pi_{\mathtt{p}\in I} P_{\mathtt{p}}) \,|\, s : \sigma$ *and* $\Gamma = \bigcup_{\mathtt{p}\in I} \Gamma_{\mathtt{p}}$. *Further, for each* $P_{\mathtt{p}}$*: (i)* $\emptyset \cdot \Gamma_{\mathtt{p}} \vdash_\Sigma P_{\mathtt{p}}$*, and (ii)* $P_{\mathtt{p}} \equiv \mathbf{0}$ or $P_{\mathtt{p}}$ *only plays role* $\mathtt{p}$ *in* s*, by* $\Gamma_{\mathtt{p}}$*. Then,* $\forall \phi \in \{\mathcal{R}_F$*-communication-safe, deadlock-free, terminating, never-terminating, live*$\}$*, if* $(\Sigma;\mathcal{R})$-$\phi(\Gamma)$*, then* P *is* ϕ*.*

4.4 Decidability

Since MAGπ is Turing-complete, determining the properties listed in def. 13 from *processes* is *undecidable* [5]. A benefit of our generalised theory is that undecidable process properties can be inferred from *decidable* type-level properties.

Theorem 4 (Decidability). *If* $(\Sigma;\mathcal{R})$-$\phi(\Gamma)$ *is decidable, then "*$\Theta \cdot \Gamma \vdash_\Sigma P$ *with* $(\Sigma;\mathcal{R})$-$\phi(\Gamma)$*" is decidable.*

Our decidability result (theorem 4) states that for any decidable type-level property, type-checking with that property is decidable. However, since MAGπ is *asynchronous*, we have *no* results on decidability of ϕ. On the contrary, as discussed in [33, sec. 7], type-level properties for *asynchronous* type theories are, in some cases, *undecidable*. This is a result of pairing buffer types with session types—which makes the type system Turing-powerful [3, thm. 2.5]. Scalas and Yoshida [33] address this issue through two methods: (*i*) standard global types

produce type contexts that can be captured through a *decidable* ***consistency*** property; and (*ii*) restricting the size of the message buffer to make properties decidable. The former ensures decidability by restricting communication to match the expressivity of global types. For the latter, they show that any type context that remains bound within a finite-sized buffer is decidable (since the type has a finite state transition system representation). In line with their results, we lift their definition of *boundedness*, *i.e.,* a restriction on the size of a buffer, to MAGπ's type system.

Definition 15 (Boundedness, from [33]**).** *We say* Γ *is* $(\Sigma;\mathcal{R})$-***bound***$_k$ *iff* $\exists k \in \mathbb{N} : \Gamma \longrightarrow^*_{(\Sigma;\mathcal{R})} \Gamma', s[\mathsf{p}] : \mathbb{M}$ *implies* $|\mathbb{M}| < k$.
We say Γ *is* $(\Sigma;\mathcal{R})$-***bounded*** *iff* $\exists k$ *finite* : $(\Sigma;\mathcal{R})$-***bound***$_k(\Gamma)$.

Using def. 15, we present our result of decidable bounded properties in theorem 5.

Theorem 5 (Decidable bounded properties). $(\Sigma;\mathcal{R})$-$bound_k(\Gamma)$ *is decidable for all* $\Sigma, \mathcal{R}$, *and* k. *Furthermore, if* $(\Sigma;\mathcal{R})$-$bounded(\Gamma)$, *then* $\forall \phi \in \{\mathcal{R}_F$-*communication-safe, deadlock-free, terminating, never-terminating, live*$\}$, *it holds that* $(\Sigma;\mathcal{R})$-$\phi(\Gamma)$ *is decidable.*

Thus, decidability is guaranteed for all protocols expressible through standard *asynchronous* global type theory, and all protocols that use finite message buffers—now with the benefit of reasoning about and handling network errors!

Example 5 (Ping Pong: Properties). Inspecting the types in example 1 and example 4, we can conclude that $\Gamma = \{s[\mathsf{p}] : \mathbb{S}_{\mathsf{p}}, s[\mathsf{q}] : \mathbb{S}_{\mathsf{q}}, s[\mathsf{r}] : \mathbb{S}_{\mathsf{r}}\}$ is bound_4. By theorem 5, Γ is decidable to check for type-level properties. On doing so, we determine that Γ is: (*i*) *safe*, it satisfies the safety property (def. 10) required for subject reduction; (*ii*) $\mathcal{R}_F$*-communication-safe*, since if we only consider reliable reductions, no buffer types remain populated; (*iii*) *terminating*, since we can count the number of steps taken to reach the end of the protocol; and (*iv*) *live*, as reliable communication $\mathbb{S}_{\mathsf{r}}$ always reduces—*i.e.,* a result is always obtained.

5 Generalising Network Assumptions

The work presented thus far covers worst-case network assumptions for communication. As beneficial as this may be for low-level networks programming, and for complex distributed applications with minimal assumptions (*e.g.* consensus protocols), not all applications are built on these pessimistic conditions. *E.g.* many distributed applications operate over the Transmission Control Protocol (TCP), and thus assume that if consecutive messages are received from the same source, then they are guaranteed to arrive in the order in which they were sent.

We now showcase the few changes to MAGπ required to alter its network assumptions. It is key to note that these changes produce a *subset* of MAGπ, thus all relevant properties continue to be valid for its TCP-compliant version.

5.1 From Total to Partial Reordering

In a *reliable* network configuration designed to run over TCP, message reordering for communication between *two parties* is guaranteed to *not occur*. Therefore, we can adjust the message reordering of MAGπ to model this environment, and strengthen our safety property $\varphi_{\mathtt{s}}$ to *TCP-safe* communication. MAGπ models message reordering through buffer congruence rules. Therefore, strengthening congruence suffices to restrict communication to the TCP-safe assumptions.

Definition 16 (TCP process-congruence). *The process congruence for the TCP-compliant subset of MAGπ, $\equiv_{TCP}$, is inductively defined using the same rules defining $\equiv$ (in def. 3), but with the following change:*

$$s{:}\sigma_1 \cdot h_1 \cdot h_2 \cdot \sigma_2 \;\equiv\; s{:}\sigma_1 \cdot h_2 \cdot h_1 \cdot \sigma_2$$

replaced by

$$\frac{\mathbf{p}_1 \neq \mathbf{p}_2 \text{ or } \mathbf{q}_1 \neq \mathbf{q}_2}{\begin{array}{l} s{:}\sigma_1 \cdot (\mathbf{p}_1 \triangleright \mathbf{q}_1 \triangleleft \mathsf{m}_1\langle w_1\rangle) \cdot (\mathbf{p}_2 \triangleright \mathbf{q}_2 \triangleleft \mathsf{m}_2\langle w_2\rangle) \cdot \sigma_2 \\ \equiv_{\mathtt{TCP}}\; s{:}\sigma_1 \cdot (\mathbf{p}_2 \triangleright \mathbf{q}_2 \triangleleft \mathsf{m}_2\langle w_2\rangle) \cdot (\mathbf{p}_1 \triangleright \mathbf{q}_1 \triangleleft \mathsf{m}_1\langle w_1\rangle) \cdot \sigma_2 \end{array}}$$

To obtain the TCP-compliant subset of MAGπ, we assume reductions over fully reliable networks and adopt TCP process congruence from def. 16, which no longer allows reordering of messages for each role couple. We now reflect this definition of TCP congruence at the type-level in def. 17, and use this to define a TCP-safety property on type contexts in def. 18.

Definition 17 (TCP type-congruence). *The type congruence for the TCP-compliant subset of MAGπ, $\equiv_{TCP}$, is inductively defined using the same rules as $\equiv$ (fig. 4), but with the following change:*

$$\frac{}{\mathbb{M}_1 \cdot \mathbb{M}_2 \equiv \mathbb{M}_2 \cdot \mathbb{M}_1} \quad \textit{replaced by} \quad \frac{\mathbf{p} \neq \mathbf{q}}{\begin{array}{l} \mathbf{p}\,!\,\mathsf{m}_1(\mathbb{T}_1) \cdot \mathbf{q}\,!\,\mathsf{m}_2(\mathbb{T}_2) \cdot \mathbb{M} \\ \equiv_{\mathtt{TCP}}\; \mathbf{q}\,!\,\mathsf{m}_2(\mathbb{T}_2) \cdot \mathbf{p}\,!\,\mathsf{m}_1(\mathbb{T}_1) \cdot \mathbb{M} \end{array}}$$

Definition 18 (TCP safety). *Predicate $\varphi_{\mathtt{TCP}}$ is a Σ-**TCP-safety** property on typing contexts iff:*

$$\begin{array}{lcl} \varphi_{\mathtt{TCP}}(\Gamma,\ s[\mathbf{p}] : \&_{i\in I}\{\mathbf{q}_i\,?\,\mathsf{m}_i(\mathbb{T}_i).\mathbb{S}_i\},\ s[\mathbf{q}]{:}\mathbb{M}) & & \\ \quad \text{and } \mathbb{M} \equiv_{\mathtt{TCP}} \mathbf{p}\,!\,\mathsf{m}(\mathbb{T}) \cdot \mathbb{M}' & & \\ \quad \text{and } \exists\, k \in I : \mathbf{q}_k = \mathbf{q} & \Longrightarrow & \mathsf{m}_k = \mathsf{m} \ \wedge\ \mathbb{T}_k = \mathbb{T} \\ \varphi_{\mathtt{TCP}}(\Gamma,\ s[\mathbf{p}] : \mu \mathtt{t}.\mathbb{S}) & \Longrightarrow & \varphi_{\mathtt{TCP}}(\Gamma,\ s[\mathbf{p}] : \mathbb{S}[{}^{\mu \mathtt{t}.\mathbb{S}}/_{\mathtt{t}}]) \\ \varphi_{\mathtt{TCP}}(\Gamma) \text{ and } \Gamma \to_{\Sigma} \Gamma' & \Longrightarrow & \varphi_{\mathtt{TCP}}(\Gamma') \end{array}$$

Similar to our previous definition of safety in def. 10, TCP safety ensures that payload types of communicating entities match. In addition, it also requires correct ordering of messages (up to $\equiv_{\mathtt{TCP}}$) by checking message labels—this is possible since messages between two parties do not get reordered, and so they must be received in the same order they are sent. In order to benefit from the session theorems proved in § 4, all that is required is to show that $\varphi_{\mathtt{TCP}} \subseteq \varphi_{\mathtt{s}}$, *i.e.,* any context that is TCP-safe is also safe. This is the only requirement since all theorems in § 4 (*i*) are parametric on the reliability function $\mathcal{R}$, including fully reliable networks; and (*ii*) are proven for $(\Sigma;\mathcal{R})$-$\varphi_{\mathtt{s}}(\Gamma)$.

Proposition 1 (Containment of $\varphi_{\mathtt{TCP}}$ in $\varphi_{\mathtt{s}}$). $\forall \Gamma \in \varphi_{\mathtt{TCP}} : \Gamma \in \varphi_{\mathtt{s}}$.

Proof. $\varphi_{\mathtt{TCP}}$ uses a fully reliable configuration of MAGπ—*i.e.,* is void of failure-handling timeouts—and thus trivially abides by [S-$\mathcal{R}_1$] and [S-$\mathcal{R}_2$]. [S-μ] is reflected directly in $\varphi_{\mathtt{TCP}}$. [S-$\rightarrow$] is reflected for $\mathcal{R} = \mathcal{R}_F$, *i.e.,* for a fully reliable configuration. [S-Com] is never violated by $\Gamma \in \varphi_{\mathtt{TCP}}$ since $\equiv_{\mathtt{TCP}} \subset \equiv$. □

6 Case Study

This work presents the Ping (examples 1–5) and Domain Name System (§ 6.1) examples as they are widely known, and between them cover the full range of our contributions. Previous related works are *not* expressive enough to model either protocol with our range of failure assumptions. Thus Ping and DNS are suitable to illustrate how MAGπ pushes the boundaries of MPST. Additional examples are provided in the technical report [23].

6.1 DNS

We now demonstrate the key features of MAGπ through a case study. We present a multiparty example of a Domain Name System (DNS) with a cache and inbuilt load-balancer. This example: (*i*) reasons about failures in its unreliable connections that are specified using our novel *viewpoint-specific* reliability sets; (*ii*) defines *failure-handling* protocols for these possible failures; (*iii*) is *bounded* (def. 15), and thus has decidable type-level properties; and (*iv*) is *safe, $\mathcal{R}_F$-communication-safe, deadlock-free, terminating,* and *live*. Typically, DNS is implemented over TCP, however the distributed components can still suffer hardware failures. To cater for this, and for better demonstration of our contributions, we describe the protocol in our failure-prone setting.

Specification We consider a specification of a client-DNS interaction, where the client consults a cache, and the DNS delegates requests to workers.

The client, represented by role c, wishes to retrieve a web-address for a particular URL, and can do so by issuing a request to the DNS. As an optimisation, the client also stores recently retrieved addresses in a local and reliable cache—thus before issuing new requests to the DNS, it first consults this cache. Upon receiving a request, the DNS offloads processing work to one of two workers, represented by roles w_1 and w_2. After retrieving the appropriate address, the worker sends the response to the client.

The reliability configuration of this application is as such: the client and cache have reliable connections, formally $\mathcal{R}(\mathsf{c}) = \{\mathsf{cache}\}$ and $\mathcal{R}(\mathsf{cache}) = \{\mathsf{c}\}$; the DNS and workers have reliable connections, formally $\mathcal{R}(\mathsf{DNS}) = \{\mathsf{w}_1, \mathsf{w}_2\}$ and $\mathcal{R}(\mathsf{w}_1) = \mathcal{R}(\mathsf{w}_2) = \{\mathsf{DNS}\}$; all other communications are unreliable.

We now present the session types specifying the communication protocol for this distributed application. We adopt shorthand notion for singleton selections, and omit payload types for simplicity, as with the ping example.

Example 6 (DNS protocol).

$$\mathbb{S}_{\mathsf{c}} = \mathsf{cache}\,!\,\mathsf{req}().\,\&\begin{cases}\mathsf{cache}\,?\,\mathsf{ans}().\mathbf{end},\\ \mathsf{cache}\,?\,404().\mathsf{DNS}\,!\,\mathsf{req}().\,\&\begin{cases}\mathsf{w}_1\,?\,\mathsf{ans}().\mathsf{cache}\,!\,\mathsf{new}().\mathbf{end},\\ \mathsf{w}_2\,?\,\mathsf{ans}().\mathsf{cache}\,!\,\mathsf{new}().\mathbf{end},\\ \clock.\,\mathsf{cache}\,!\,\mathsf{ko}().\mathbf{end}\end{cases}\end{cases}$$

$$\mathbb{S}_{\mathsf{cache}} = \&\begin{cases}\mathsf{c}\,?\,\mathsf{req}().\,\oplus\begin{cases}\mathsf{c}\,!\,\mathsf{ans}().\mathbf{end},\\ \mathsf{c}\,!\,404().\,\&\begin{cases}\mathsf{c}\,?\,\mathsf{new}().\mathbf{end},\\ \mathsf{c}\,?\,\mathsf{ko}().\mathbf{end}\end{cases}\end{cases}\end{cases}$$

$$\mathbb{S}_{\mathsf{DNS}} = \&\begin{cases}\mathsf{c}\,?\,\mathsf{req}().\,\oplus\begin{cases}\mathsf{w}_1\,!\,\mathsf{req}().\mathsf{w}_2\,!\,\mathsf{ko}().\mathbf{end}\\ \mathsf{w}_2\,!\,\mathsf{req}().\mathsf{w}_1\,!\,\mathsf{ko}().\mathbf{end}\end{cases}\\ \clock.\,\mathsf{w}_1\,!\,\mathsf{ko}().\mathsf{w}_2\,!\,\mathsf{ko}().\mathbf{end}\end{cases}$$

$$\mathbb{S}_{\mathsf{w}_i} = \&\begin{cases}\mathsf{DNS}\,?\,\mathsf{req}().\mathsf{c}\,!\,\mathsf{ans}().\mathbf{end},\\ \mathsf{DNS}\,?\,\mathsf{ko}().\mathbf{end}\end{cases}$$

Our viewpoint-specific definition of reliability is necessary to specify the reliable connections with the DNS and workers whilst maintaining unreliable connections with the client. Additionally, the client type $\mathbb{S}_{\mathsf{c}}$ (resp. the DNS type $\mathbb{S}_{\mathsf{DNS}}$) is dependant on using undirected branching (resp. selection). Hence this example is not expressible using previous theory [4,33].

7 Related Work, Conclusions and Future Work

Modelling failures has become a relevant and widely researched topic in recent years. We elaborate on how our generic type system and modular language differs from, and in some cases may possibly subsume, related work.

Majumdar *et al.* [24] introduce undirected branching as a means of catering for the non-deterministic *partial* reordering of messages that is possible in networks using the Transmission Control Protocol (TCP). As shown in § 5, the modularity of our type system allows MAGπ to be adapted to support this network configuration, as well as other settings with lower levels of abstraction.

Affine type systems define types that can be used *at most once*. Affine session types [25,12,6] use affine typing metatheory to allow sessions to be prematurely cancelled in the event of failure. These works only model application-level failure (using try/catch blocks) and do not necessarily describe *how* a failure is handled, but only allow the initial protocol to be abandoned if failure occurs.

Viering *et al.* [38] present a MPST theory for event-driven distributed systems, where processes are restarted by monitors if they crash. This approach *requires* a centralised reliable node, a notion that is subsumed by our *view-point specific* definition of *reliability*, def. 7.

Chen *et al.* [8] remove the need for a centralised reliable node. They equip their type system with *synchronisation points* capable of detecting and handling failures raised by the nodes that experience them. Similarly, Adameit *et al.* [1] consider an environment free from a centralised reliable node where unstable *links* between participants can fail. They introduce the concept of *optional blocks*,

allowing *default values* to substitute data not received due to communication failure. Viering *et al.* [37], motivated by consensus algorithms, delegate a group of processes as a permanently available recovery system capable of monitoring processes and informing them of failures. Thus, they no longer rely on *one* centralised robust node, but instead assume that at least some of the processes that make up the coordinator are alive at any given time. The drawback in these approaches is their reliance on coordination to handle faults. This may not be suitable with certain network configurations and failure-models. Since our type system handles failure through low-level techniques, it remains agnostic to the types of failures, and is suitable for any non-Byzantine network configuration.

Recent work by Peters *et al.* [28] extends global type theory with *failure annotations*—marking communication susceptible to failures and the kind of failure (specifically either process crashes or message loss). They handle failure by defining *default values and branches*. Since the theory is an extension of global types, it suffers from the same problems that are addressed through generalised MPST. Additionally, the work is not agnostic to failure-models, and so it is uncertain if the theory is capable of model failures other than the two considered.

Most similar to MAGπ is work by Barwell *et al.* [4], where generalised session type theory is extended to reason about crash-stop failures. They reserve the crash message label, which can be used in receive branches to detect node failure and specify failure-handling subprotocols. In line with our research, their type system is generic, thus improving its expressiveness. However, unlike MAGπ, their theory is not asynchronous, does not support undirected branching/selection, and assumes crash-stops to be the only possible faults—we address and capture a range of failures such as crash failures, link failures, message loss, delays and reordering and network partitioning.

Distributed variations of the π-calculus [2,30,7,13] introduce process *locations*—representations of real-world physical hardware. Processes are assigned to locations to form a topology, and locations can be crashed to model failures. None of these calculi model the range of failures that are supported by MAGπ, nor do they have type systems to ensure communication-safe failure handling.

To conclude the paper, we presented MAGπ—a Multiparty, Asynchronous and Generalised π-calculus which addresses the widest set of non-Byzantine faults by using timeouts and the most general reliability definition. Our language builds on the *generalised* and *asynchronous* MPST, which is the most flexible for distributed programming. We prove subject reduction and session fidelity; a series of process properties, as well as fault-handling safety and reliability adherence. As future work, we aim to investigate linear logic for Curry-Howard correspondences in order to understand the foundational and canonical meaning of faults and reliability. We aim to investigate Byzantine faults in combination with the non-Byzantine faults addressed here. Lastly, we will explore the use of model checking to streamline the verification of process properties.

Acknowledgements. We thank the anonymous reviewers and give a special thanks to Simon Fowler for his invaluable support and feedback.

References

1. Adameit, M., Peters, K., Nestmann, U.: Session types for link failures. In: Bouajjani, A., Silva, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10321, pp. 1–16. Springer (2017). https://doi.org/10.1007/978-3-319-60225-7_1
2. Amadio, R.M.: An asynchronous model of locality, failure and process mobility. In: Garlan, D., Métayer, D.L. (eds.) Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1282, pp. 374–391. Springer (1997). https://doi.org/10.1007/3-540-63383-9_92
3. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. Log. Methods Comput. Sci. **12**(4) (2016). https://doi.org/10.2168/LMCS-12(4:7)2016
4. Barwell, A.D., Scalas, A., Yoshida, N., Zhou, F.: Generalised multiparty session types with crash-stop failures. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.35
5. Busi, N., Gabbrielli, M., Zavattaro, G.: On the expressive power of recursion, replication and iteration in process calculi. Math. Struct. Comput. Sci. **19**(6), 1191–1222 (2009). https://doi.org/10.1017/S096012950999017X
6. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty sessions. Math. Struct. Comput. Sci. **26**(2), 156–205 (2016). https://doi.org/10.1017/S0960129514000164
7. Castellani, I.: Process algebras with localities. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 945–1045. North-Holland / Elsevier (2001). https://doi.org/10.1016/b978-044482830-9/50033-3
8. Chen, T., Viering, M., Bejleri, A., Ziarek, L., Eugster, P.: A type theory for robust failure handling in distributed systems. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9688, pp. 96–113. Springer (2016). https://doi.org/10.1007/978-3-319-39570-8_7
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
10. Clarke, E.M., Klieber, W., Novácek, M., Zuliani, P.: Model checking and the state explosion problem. In: LASER Summer School. Lecture Notes in Computer Science, vol. 7682, pp. 1–30. Springer (2011). https://doi.org/10.1007/978-3-642-35746-6_1
11. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: ECOOP 2006. vol. 4067, pp. 328–352. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/11785477_20, http://link.springer.com/10.1007/11785477_20, lecture Notes in Computer Science
12. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. Proc. ACM Program. Lang. **3**(POPL), 28:1–28:29 (2019). https://doi.org/10.1145/3290341

13. Hennessy, M.: A distributed Pi-calculus. Cambridge University Press (2007)
14. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
17. Kokke, W., Dardha, O.: Deadlock-free session types in linear haskell. In: Hage, J. (ed.) Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021. pp. 1–13. ACM (2021). https://doi.org/10.1145/3471874.3472979
18. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with mungo and stmungo. In: Cheney, J., Vidal, G. (eds.) Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016. pp. 146–159. ACM (2016). https://doi.org/10.1145/2967973.2968595
19. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine rust programming with multiparty session types. In: 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.4
20. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229
21. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: liveness and safety for channel-based programming. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 748–761. ACM (2017). https://doi.org/10.1145/3009837.3009847
22. Laprie, J.C.: Dependable computing and fault-tolerance. Digest of Papers FTCS-15 **10**(2), 124 (1985)
23. Le Brun, M.A., Dardha, O.: Magπ: Types for failure-prone communication (2023). https://doi.org/10.48550/ARXIV.2301.10827, https://arxiv.org/abs/2301.10827
24. Majumdar, R., Mukund, M., Stutz, F., Zufferey, D.: Generalising projection in asynchronous multiparty session types. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.35
25. Mostrous, D., Vasconcelos, V.T.: Affine sessions. Log. Methods Comput. Sci. **14**(4) (2018). https://doi.org/10.23638/LMCS-14(4:14)2018
26. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference. pp. 305–319. USENIX Association (2014), https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
27. Orchard, D., Yoshida, N.: Session types with linearity in haskell. Behavioural Types: from Theory to Tools p. 219 (2017)

28. Peters, K., Nestmann, U., Wagner, C.: Fault-tolerant multiparty session types. In: Mousavi, M.R., Philippou, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 42nd IFIP WG 6.1 International Conference, FORTE 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13273, pp. 93–113. Springer (2022). https://doi.org/10.1007/978-3-031-08679-3_7
29. Pierce, B.C.: Types and programming languages. MIT Press (2002)
30. Riely, J., Hennessy, M.: Distributed processes and location failures. Theor. Comput. Sci. **266**(1-2), 693–735 (2001). https://doi.org/10.1016/S0304-3975(00)00326-1
31. Rossi, M.: Modeling and analysis of communicating systems. Formal Aspects Comput. **33**(2), 297–298 (2021). https://doi.org/10.1007/s00165-021-00533-8
32. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
33. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019). https://doi.org/10.1145/3290343
34. Tabone, G., Francalanza, A.: Session types in elixir. In: Castegren, E., Koster, J.D., Fowler, S. (eds.) AGERE 2021: Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, Virtual Event / Chicago, IL, USA, 17 October 2021. pp. 12–23. ACM (2021). https://doi.org/10.1145/3486601.3486708
35. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE '94. LNCS, vol. 817, pp. 398–413. Springer (1994)
36. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
37. Viering, M., Chen, T., Eugster, P., Hu, R., Ziarek, L.: A typing discipline for statically verified crash failure handling in distributed systems. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801, pp. 799–826. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_28
38. Viering, M., Hu, R., Eugster, P., Ziarek, L.: A multiparty session typing discipline for fault-tolerant event-driven distributed programming. Proc. ACM Program. Lang. **5**(OOPSLA), 1–30 (2021). https://doi.org/10.1145/3485501