MAGπ!: The Role of Replication in Typing Failure-Prone Communication

Matthew Alan Le Brun(✉) and Ornela Dardha

University of Glasgow, Glasgow, UK
{matthewalan.lebrun,ornela.dardha}@glasgow.ac.uk

Abstract. MAGπ is a Multiparty, Asynchronous and Generalised π-calculus that introduces timeouts into session types as a means of reasoning about failure-prone communication. Its type system guarantees that all possible message-loss is handled by timeout branches. In this work, we argue that the previous is unnecessarily strict. We present MAGπ!, an extension serving as the first introduction of replication into Multiparty Session Types (MPST). Replication is a standard π-calculus construct used to model infinitely available servers. We lift this construct to type-level, and show that it simplifies specification of distributed client-server interactions. We prove properties relevant to generalised MPST: subject reduction, session fidelity and process property verification.

Keywords: Multiparty Session Types · Failure · Replication

1 The Tale of the MAG(pie/π)

The magpie is a bird with deep ties to British folklore. The first known mention of their counting for fortune telling dates back to 1780, where John Brand writes what is thought to be one of the original versions of the magpie rhyme [6]:

"*One for sorrow, Two for mirth, Three for a funeral, And four for a birth.*"

We can imagine that the natural reaction of a person who spots a solitary magpie is to scan the surrounding area for its companion. Alas, if no one is immediately visible, the person desperately waits—hoping a second magpie comes their way. But how long should one wait? The reality is that it is *impossible* to know the difference between *no magpie* and a magpie that has *not yet arrived*. To computer scientists, this is a well known *impossibility result* [2]. In the study of *distributed systems* and *fault tolerance*, mechanisms must be employed to approximate the impossibility result of determining whether a message has been *lost* or *delayed*—*e.g.* by using a *timeout*. Hence, the computer scientist who spots a lonely magpie knows to only wait some fixed amount of time before *assuming* that no other magpie is coming and accepting their sorrowful faith. This philosophy is the core principle of the process calculus MAGπ [17], a language designed to model communication failures (via *message loss*) with a generic type system aiming to provide configurable runtime guarantees.

© IFIP International Federation for Information Processing 2024
Published by Springer Nature Switzerland AG 2024
V. Castiglioni and A. Francalanza (Eds.): FORTE 2024, LNCS 14678, pp. 99–117, 2024.
https://doi.org/10.1007/978-3-031-62645-6_6

MAGπ is a Multiparty, Asynchronous and Generalised π-calculus, modelling distributed communication over n-participant *sessions*. Its key features include non-deterministic failure injection into the runtime of a program, asynchronous communication via *bag* buffers (allowing for total message reordering), and a generic type system capable of providing guarantees of runtime properties via *session types*. Session types [13,15,26] are *behavioural type systems* allowing for formal specification of communication protocols—their main benefit being that they provide correctness guarantees on both protocol design and implementation. *Multiparty session types* (MPST) [5,16,25] are a branch of session type theory that aims to support protocols involving *any* number of participants with interleaving communication. MAGπ builds upon a generalised form of MPST [4,25], where protocols are defined by a collection of *local types*—the communication patterns of individual participants' perspectives—which should be exhaustively checked (*e.g.* via model checking) to determine any properties they observe. Novelties of MAGπ stem from how it embraces the impossibility result of distinguishing between dropped or delayed messages; its language and type system use non-deterministic timeouts to model the *assumption* of failures. The type system guarantees that *all* failure-prone communication is handled by a timeout branch. In this work, we argue that the previous approach can, in some scenarios, be unnecessarily strict—resulting in needlessly more complex protocols. Some configurations may wish to leave the handling of failures up to senders, as opposed to recipients; these usually take the form of *client-server* interactions where servers are designed to remain infinitely available. For example, if a request to a web server were to drop, it is the client's responsibility to re-issue that request. We present an extension to MAGπ that better models infinitely available servers and simplifies failure-handling for client-server interactions.

In the π-calculus [24], a standard construct often used for representing infinite behaviour is that of *replication*. A replicated process is one which can be informally described as *infinitely available*. Naturally, the use of replicated processes lends itself well to the modelling of client-server interactions. We demonstrate how the use of replication in MAGπ can, not only better model infinitely available servers, but also simplify their protocols by relaxing the requirement of failure-handling branches from *every* receive to only *linear* receives.

Example 1 (Type-level replication). We evolve the motivating example presented in [17, Ex. 1], the *ping* protocol. Consider three participants: client c, server s, and result channel r. Communication between c and r is reliable; whereas with s is *unreliable*. The session types for a three-attempt ping in MAGπ! are:

$$
\begin{aligned}
S_{\mathsf{r}} &= \&\{\mathsf{c} : \mathsf{ok}\,.\,\mathbf{end},\ \mathsf{c} : \mathsf{ko}\,.\,\mathbf{end}\} \\
S_{\mathsf{c}} &= \oplus\,\mathsf{s} : \mathsf{ping}\,.\,\&\begin{cases} \mathsf{s} : \mathsf{pong}\,.\ \oplus\,\mathsf{r} : \mathsf{ok}\,.\,\mathbf{end}, \\ \circlearrowleft.\ \oplus\,\mathsf{s} : \mathsf{ping}\,.\,\&\begin{cases} \mathsf{s} : \mathsf{pong}\,.\ \oplus\,\mathsf{r} : \mathsf{ok}\,.\,\mathbf{end}, \\ \circlearrowleft.\ \oplus\,\mathsf{s} : \mathsf{ping}\,.\,\&\begin{cases} \mathsf{s} : \mathsf{pong}\,.\oplus\,\mathsf{r} : \mathsf{ok}\,.\,\mathbf{end}, \\ \circlearrowleft.\ \oplus\,\mathsf{r} : \mathsf{ko}\,.\,\mathbf{end} \end{cases} \end{cases} \end{cases} \\
S_{\mathsf{s}} &= !\mathsf{c} : \mathsf{ping}\,.\ \oplus\,\mathsf{c} : \mathsf{pong}\,.\,\mathbf{end}
\end{aligned}
$$

Client c sends a message with label ping to server s ($\oplus\,\mathsf{s} : \mathsf{ping}$) and waits for a pong response ($\&\mathsf{s} : \mathsf{pong}$). If successful, an ok message is sent to results role

r and the session is terminated for the client ($\mathbf{end}$). Since communication with the server is *unreliable*, receipt of the pong message is not guaranteed, and must be handled by a *timeout* branch $\circlearrowleft$. The client attempts to reach the server 3 times—if all attempts fail, it sends a ko message to r. The result role r waits for either of the reliable responses from c, thus no timeout is defined. Server s is defined as the replicated receive $!\mathsf{c} : \mathsf{ping} . \oplus \mathsf{c} : \mathsf{pong} . \mathbf{end}$, denoting its constant availability to receive a ping request and send a pong response. We highlight the absence of a failure-handling timeout branch in S_{s}; the server does not need to change its behaviour if a client request fails. Furthermore, if the pong reply fails, the server remains available to handle any number of retries from the client. Thus, the use of replication has offloaded the handling of failures entirely onto the client-side, has made the protocol more modular (since the type for s is now agnostic of a client's retry limit), and is simpler w.r.t. to the MAGπ specification.

Contributions. Concretely, our contributions are as follows:

1. **MAGπ! Language**: We present MAGπ! (Sect. 2), an extension of MAGπ that does away with recursion in favour of replication as a better means of modelling client-server interactions.
2. **MAGπ! Types**: We lift replication to type-level in Sect. 3. To the best of our knowledge, this work serves as the *first introduction* of replication into MPST. We improve upon the theory of MAGπ and show how three type contexts (*unrestricted*, *linear* and *affine*) can be used to type—and *simplify*—failure-prone communication in client-server interactions.
3. **MAGπ! Metatheory**: Sect. 4 expounds upon the metatheory of our type system. We prove *subjection reduction* and *session fidelity*, and demonstrate how they can be used for *property verification*. MAGπ! provides a *failure handling guarantee*, ensuring all failure-prone communication is handled by a timeout branch—a responsibility which servers offload to clients.

In Sect. 5 we conclude and give an account of related and future work. Details of proofs and additional examples can be found in our technical report [18].

On Delegation and Language Simplification. This work builds upon a *subset* of MAGπ [17] as our language only considers communication over a *single session*. Reasons for this are: *(i)* to simplify notation for better readability due to limited space; and *(ii)* to remove session fidelity assumptions. On the latter, generalised MPST theory assumes communication over a single session to prove *session fidelity* (a.k.a. protocol compliance) [25, Def. 5.3]. This is to remove deadlocks that can occur due to incorrect interleaving of multiple sessions. Effectively, the language subset we consider syntactically abides by the assumptions of session fidelity by assuming all communication happens over a single session and by removing delegation. We foresee no issues with extending MAGπ! to multiple sessions, although this will only improve the number of *safe* protocols that can be expressed and has *no effect* on verification of other properties. Lastly, replication in MAGπ! is a *top-level* construct only. This simplifies our type system at the cost of sacrificing expressivity of nested replication. The type system can still express meaningful examples (*e.g.* load balancers), and we intend to explore guarded and nested replication in future work.

2 Bird Songs

We present MAGπ!, an extension of MAGπ that replaces recursion with replicated processes as its preferred means of reasoning about infinite behaviour. Programs in MAGπ! represent distributed networks, consisting of concurrent and parallel processes running on machines connected over some *failure-prone* medium. We discuss how networks of various topologies are defined in Sect. 2.1. Section 2.2 details the syntax and semantics of processes.

2.1 Topology

Distributed protocols typically consist of a number of participants (or *roles*) representing physically separated devices, communicating over a *failure-prone* network. We model such a setting by associating processes to uniquely identifiable roles, which communicate asynchronously through a *bag buffer* allowing for *total message reordering.* Roles are related through a notion of *reliability*, modelling physical locations of processes—*i.e.,* reliable roles are ones that live on the same physical device and thus are not susceptible to communication errors. A formal account of networks, buffers and reliability is given below.

Networks. A program in MAGπ! models some distributed network $\mathcal{N}$. These networks consist of a parallel composition of processes, each representing specific *roles* in the network. The formal description of a network is given by Definition 1.

Definition 1 (Networks). *A network $\mathcal{N}$ is given by the following grammar:*

$$\mathcal{N} ::= \mathsf{p} \triangleleft \mathcal{P} \mid \mathcal{N} \mathbin{||} \mathcal{N} \mid \mathcal{B}$$

where $\mathcal{B}$ is a message buffer; $\mathcal{P}$ is the process instruction; and p is a role name.

A *process* $\mathsf{p} \triangleleft \mathcal{P}$ consists of a uniquely identifying role name p, and process instructions $\mathcal{P}$. It is key to note that all processes, *i.e.,* participants, of a network are syntactically defined—thus, MAGπ! assumes a finite network size where all participants are *statically* known. The $||$ constructor denotes *parallel composition* of processes within a network, and $\mathcal{B}$ is its message buffer.

Buffers. MAGπ! models asynchrony through a *bag buffer* (semantics discussed in Sect. 2.2). The buffer, Definition 2, serves two purposes. Firstly, it allows for non-blocking (*fire and forget*) sends by acting as an intermediary where messages wait until recipients are ready to consume them. Second, and important to distributed communication, is that it models messages *in transit* over the network and is thus the point-of-failure in our system.

Definition 2 (Buffers). *A message $\mathcal{M}$ is defined as $\mathcal{M} ::= \langle \mathbf{p} \to \mathbf{q}, \mathsf{m}\langle\tilde{v}\rangle\rangle$, i.e., a tuple identifying the source and destination of the message ($\mathbf{p} \to \mathbf{q}$), along with a message label and payload contents ($\mathsf{m}\langle\tilde{v}\rangle$). A buffer $\mathcal{B}$ is a multiset of messages $\mathcal{M}$. Concatenating a message $\mathcal{M}$ with a buffer $\mathcal{B}$, written $\mathcal{B} \cdot \mathcal{M}$ corresponds to the multiset sum of $\mathcal{B} + \{\mathcal{M}\}$.*

Reliability. A network is initialised with a reliability relation $\mathcal{R}$ (Definition 3), defining roles which may communicate sans failure. All communication outwith the reliability relation is considered failure-prone; this may be used to simulate physical topologies, or to study a protocol at various degrees of reliability.

Definition 3 (Reliability). *Given a network $\mathcal{N}$, and set of roles ρ acting in $\mathcal{N}$, the reliability relation $\mathcal{R}$ is a subset of (or equal to) $\{\{\mathbf{p}, \mathbf{q}\} : \mathbf{p}, \mathbf{q} \in \rho \wedge \mathbf{p} \neq \mathbf{q}\}$. We write $\mathcal{N} :: \mathcal{R}$ to denote a network $\mathcal{N}$ governed by reliability relation $\mathcal{R}$. We use shorthand $\mathcal{N} :: \mathcal{F}$ to denote a fully reliable network, and $\mathcal{N} :: \emptyset$ to denote a fully unreliable network.*

Example 2 (Load Balancer: Network). Consider a load balancer network with server $\mathbf{s}$, workers $\mathbf{w}_1$, $\mathbf{w}_2$, and client $\mathbf{c}$. Assuming server-worker communication to be reliable, the network may be configured as below:

$$\mathbf{s} \triangleleft \mathcal{P}_{\mathbf{s}} \mid\mid \mathbf{w}_1 \triangleleft \mathcal{P}_{\mathbf{w}_1} \mid\mid \mathbf{w}_2 \triangleleft \mathcal{P}_{\mathbf{w}_2} \mid\mid \mathbf{c} \triangleleft \mathcal{P}_{\mathbf{c}} \mid\mid \mathcal{B} :: \{\{\mathbf{s}, \mathbf{w}_1\}, \{\mathbf{s}, \mathbf{w}_2\}\}$$

2.2 Processes

Definition 4 (Process syntax). *The syntax for defining process instructions $\mathcal{P}$ is given by the following grammar:*

$$\begin{aligned}
\mathcal{P} &::= !_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{x_i}) \,.\, P_i \mid \boxed{\mathcal{P} \mid \mathcal{P}} \mid P \\
P &::= \mathbf{0} \mid \&_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{x_i}) \,.\, P_i\, [, \circledcirc \,.\, P'] \mid \oplus\, \mathbf{p} : \mathsf{m}\langle\tilde{c}\rangle \,.\, P \\
c &::= x \mid v \qquad v ::= \text{basic values}
\end{aligned}$$

All branching terms assume $I \neq \emptyset$ and all couples $\mathbf{p}_i : \mathsf{m}_i$ to be pairwise distinct. Receiving constructs act as binders on their payloads.

A process $\mathcal{P}$ can either be a replicated server or a linear process. *Replicated receive* $!_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{x_i}) \,.\, P_i$ denotes a server constantly available to receive any of a set of messages from roles $\mathbf{p}_i$ with labels m_i. The received payload is bound to $\tilde{x_i}$ before pulling out a copy of P_i to run in parallel with the server. *Parallel composition* $\mid$ is a $\boxed{\text{runtime}}$ only construct at the process-level. It is used to denote composition of linear continuations pulled out of a replicated receive. Linear processes $(P, Q, \dots)$ consist of: *(i)* the *empty process* $\mathbf{0}$; *(ii) linear receives* $\&_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{x_i}) \,.\, P_i\, [, \circledcirc \,.\, P']$, where a role waits for one of a set of messages from some other roles $\mathbf{p}_i$ with labels m_i, binding the received payload to $\tilde{x_i}$ before proceeding according to P_i; *(iii)* an optional *nondeterministic*

$\boxed{\text{Process semantics}}$

P-Send
$$\mathbf{p} \triangleleft \mathbf{q} \oplus \mathsf{m}\langle \tilde{v} \rangle \,.\, P \mid\mid \mathcal{B} \longrightarrow_{\mathcal{R}} \mathbf{p} \triangleleft P \mid\mid \mathcal{B} \cdot \langle \mathbf{p} \to \mathbf{q}, \mathsf{m}\langle \tilde{v} \rangle \rangle$$

P-Recv
$$\frac{\exists k \in I \text{ s.t. } \mathbf{r}_k = \mathbf{q} \text{ and } \mathsf{m}_k = \mathsf{n} \text{ and } |\tilde{y}_k| = |\tilde{v}|}{\mathbf{p} \triangleleft \&_{i \in I} \mathbf{r}_i : \mathsf{m}_i(\tilde{y}_i) \,.\, P_i \,[, \circlearrowright . \, P'] \mid\mid \mathcal{B} \cdot \langle \mathbf{q} \to \mathbf{p}, \mathsf{n}\langle \tilde{v} \rangle \rangle \longrightarrow_{\mathcal{R}} \mathbf{p} \triangleleft P_k\{\tilde{v}/\tilde{y}_k\} \mid\mid \mathcal{B}}$$

P-!Recv
$$\frac{\mathcal{P} = !_{i \in I} \mathbf{r}_i : \mathsf{m}_i(\tilde{y}_i) \,.\, P_i \qquad \exists k \in I \text{ s.t. } \mathbf{r}_k = \mathbf{q} \text{ and } \mathsf{m}_k = \mathsf{n} \text{ and } |\tilde{y}_k| = |\tilde{v}|}{\mathbf{p} \triangleleft \mathcal{P} \mid\mid \mathcal{B} \cdot \langle \mathbf{q} \to \mathbf{p}, \mathsf{n}\langle \tilde{v} \rangle \rangle \longrightarrow_{\mathcal{R}} \mathbf{p} \triangleleft \mathcal{P} \mid P_k\{\tilde{v}/\tilde{y}_k\} \mid\mid \mathcal{B}}$$

N- ||
$$\frac{\mathcal{N} \to \mathcal{N}'}{\mathcal{N} \mid\mid \mathcal{N}'' \to \mathcal{N}' \mid\mid \mathcal{N}''}$$

P- |
$$\frac{\mathcal{N} \mid\mid \mathbf{p} \triangleleft \mathcal{P} \to \mathcal{N}' \mid\mid \mathbf{p} \triangleleft \mathcal{P}'}{\mathcal{N} \mid\mid \mathbf{p} \triangleleft \mathcal{P} \mid \mathcal{P}'' \to \mathcal{N}' \mid\mid \mathbf{p} \triangleleft \mathcal{P}' \mid \mathcal{P}''}$$

$\boxed{\text{Failure semantics}}$

F-Drop
$$\frac{\{\mathbf{p}, \mathbf{q}\} \notin \mathcal{R}}{\mathcal{B} \cdot \langle \mathbf{p} \to \mathbf{q}, \mathsf{m}\langle \tilde{v} \rangle \rangle \longrightarrow_{\mathcal{R}} \mathcal{B}}$$

F-Timeout
$$\mathbf{q} \triangleleft \&_{i \in I} \mathbf{r}_i : \mathsf{m}_i(\tilde{y}_i) \,.\, P_i, \circlearrowright . \, P' \longrightarrow_{\mathcal{R}} \mathbf{q} \triangleleft P'$$

Fig. 1. Network semantics.

timeout branch $[, \circlearrowright . \, P']$ attached to linear receives to handle possible failure of messages, instructing the process to proceed according to P'; and *(iv) linear sends* $\oplus \mathbf{p} : \mathsf{m}\langle \tilde{c} \rangle \,.\, P$ which sends a message towards $\mathbf{p}$ with label m and payload $\tilde{c}$ before continuing according to P. A payload c is either a *variable* $(x, y, \dots)$ or some assumed basic value (integers, reals, strings, ...). We omit conditional branching constructs such as if-then-else and case statements as they are routine and orthogonal to our work (we assume them in examples).

Definition 5 (Network Semantics). *Reduction on networks is parametric on a reliability relation* $\mathcal{R}$*. The reduction relation* $\longrightarrow_{\mathcal{R}}$ *is inductively defined by the rules listed in Fig. 1, up-to congruence (rules below):*

$$\mathcal{N}_1 \mid\mid \mathcal{N}_2 \equiv \mathcal{N}_2 \mid\mid \mathcal{N}_1 \qquad (\mathcal{N}_1 \mid\mid \mathcal{N}_2) \mid\mid \mathcal{N}_3 \equiv \mathcal{N}_1 \mid\mid (\mathcal{N}_2 \mid\mid \mathcal{N}_3) \qquad \mathcal{P}_1 \mid \mathcal{P}_2 \equiv \mathcal{P}_2 \mid \mathcal{P}_1$$
$$\mathcal{N} \mid\mid \mathbf{p} \triangleleft \mathbf{0} \equiv \mathcal{N} \text{ if } \mathbf{p} \notin \mathsf{roles}(\mathcal{N}) \;\; (\mathcal{P}_1 \mid \mathcal{P}_2) \mid \mathcal{P}_3 \equiv \mathcal{P}_1 \mid (\mathcal{P}_2 \mid \mathcal{P}_3) \;\; \mathcal{P} \mid \mathbf{0} \equiv \mathcal{P}$$

Network dynamics (Fig. 1) are divided into *process* and *failure* semantics. A process sends a message via rule [P-Send] , which places the message in the network buffer and advances the sending process to its continuation. Conversely, processes receive messages (rule [P-Recv]) by consuming a message from the buffer, advancing the process to its continuation and substituting bound payloads with the received data. In a similar manner, servers may consume messages from the buffer using rule [P-!Recv] ; instead of advancing the process, a copy of its continuation is *pulled out* and placed in parallel. This allows servers to concurrently handle and receive client requests.

Message failure is modelled through rule [F-Drop] . We recall that buffers model messages *in transit*, thus this rule may—*at any time*—drop a message from the buffer if it is unreliable. It is key to note that failure in these semantics is *nondeterministic*. A client may consume a message before it is dropped, representing a successful transmission; or the message may be dropped before consumed, representing the failure case. Reduction of *timeout branches* is also nondeterministic since it is impossible to distinguish between *dropped messages* (*no magpie*) and *delayed messages* (the magpie that has *not yet arrived*). Therefore, rule [F-Timeout] can *at any time* reduce a waiting process to its timeout branch, modelling either the handling of message failure or an incorrect assumption of failure (*i.e.,* message delay).

Example 3 (Load Balancer: Processes). We present the processes of our load balancer. An output role o, which is reliable w.r.t. the client, has been added.

$$\begin{aligned}
\mathsf{s} \triangleleft\, !\mathsf{c} : \mathsf{req}(x)\,.\,\mathtt{case\ flip()\ of}\ &\begin{cases} \mathtt{heads} \to \oplus \mathsf{w}_1 : \mathsf{req}\langle x \rangle\,.\,\mathbf{0} \\ \mathtt{tails} \to \oplus \mathsf{w}_2 : \mathsf{req}\langle x \rangle\,.\,\mathbf{0} \end{cases}\\
\mathsf{w}_1 \triangleleft\, !\mathsf{s} : \mathsf{req}(d)\,.\, \oplus \mathsf{c} : \mathsf{ans}\langle f(d) \rangle\,.\,\mathbf{0}&\\
\mathsf{w}_2 \triangleleft\, !\mathsf{s} : \mathsf{req}(d)\,.\, \oplus \mathsf{c} : \mathsf{ans}\langle f(d) \rangle\,.\,\mathbf{0}&\\
\mathsf{c} \triangleleft \oplus \mathsf{s} : \mathsf{req}\langle 42 \rangle\,.\ \&&\begin{cases} \mathsf{w}_1 : \mathsf{ans}(y)\,.\, \oplus \mathsf{o} : \mathsf{output}\langle y \rangle\,.\,\mathbf{0} \\ \mathsf{w}_2 : \mathsf{ans}(y)\,.\, \oplus \mathsf{o} : \mathsf{output}\langle y \rangle\,.\,\mathbf{0} \\ \circlearrowright.\, \oplus \mathsf{o} : \mathsf{err}\langle \text{“Request timed out”} \rangle\,.\,\mathbf{0} \end{cases}\\
\mathsf{o} \triangleleft \&\{\mathsf{c} : \mathsf{output}(out)\,.\,\mathbf{0}, \mathsf{c} : \mathsf{err}(msg)\,.\,\mathbf{0}\}&
\end{aligned}$$

Example 4 (Interactions with Failure: Processes). Now we demonstrate interactions unique to our language which result from the use of timeouts as imperfect failure detectors. Consider the following network snippet $\mathcal{N}_f :: \emptyset$:

$$\mathsf{p} \triangleleft \oplus \mathsf{q} : \mathsf{m}\langle 42 \rangle\,.\,P \ \ || \ \ \mathsf{q} \triangleleft \&\{\mathsf{p} : \mathsf{m}(x)\,.\,P',\ \circlearrowright.\,P''\} \ \ || \ \ \{\langle \mathsf{p} \to \mathsf{q}, \mathsf{m}\langle \text{“Life is”} \rangle \rangle\}$$

These processes denote communication between two roles (p and q), where a message labelled m with the string “Life is” has already been sent, and a second message *also* labelled m is to be sent with payload 42. There are *four* possible immediate reduction steps for this network: *(i)* role q consumes the message in the buffer via [P-Recv] (the intended behaviour); *(ii)* role p places message $\langle \mathsf{p} \to \mathsf{q}, \mathsf{m}\langle 42 \rangle \rangle$ in the buffer via [P-Send] , this may possibly result in message reordering due to the bag buffer semantics; *(iii)* message $\langle \mathsf{p} \to \mathsf{q}, \mathsf{m}\langle \text{“Life is”} \rangle \rangle$ is dropped from the buffer via [F-Drop] , then q may either correctly assume failure through a timeout, or if the sender is quick enough the message $\langle \mathsf{p} \to \mathsf{q}, \mathsf{m}\langle 42 \rangle \rangle$ could still be received in its place; and *(v)* role q can incorrectly assume a failure and timeout via [F-Timeout] even though message $\langle \mathsf{p} \to \mathsf{q}, \mathsf{m}\langle \text{“Life is”} \rangle \rangle$ is in the buffer. It is not difficult to see how items (*ii*) to (*iv*) may lead to errors. Our types and metatheory mitigate the occurrence of these possibly unsafe networks by enforcing a safe design of protocols.

3 Harmonisation

We now present the *multiparty*, *asynchronous*, and *generalised* type system for MAGπ!. To the best of our knowledge, this is the first work to introduce *replication* and *parallel composition* for local types in MPST. We show how these constructs lend themselves well to typing distributed client-server interactions.

3.1 Types

The syntax for MAGπ! types are given in Definition 6. Our type system does away with tail-recursive binders (as is standard in MPST), instead opting for a *replicated receive* type. The syntax distinguishes between different classes of types. Namely, we present *replicated-*, *session-*, *message-* and *basic-types*—each of which are used differently by the type contexts (Definition 7).

Definition 6 (Types). *The syntax for MAGπ! types is given by:*

$$R ::= !_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i$$
$$S ::= \oplus_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i \mid \&_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i \,[, \circlearrowright . S'] \mid \boxed{S \mid S} \mid \mathbf{end}$$
$$M ::= (\mathbf{p} \to \mathbf{q}, \mathsf{m}(\tilde{B}))$$
$$B ::= \mathtt{Int},\ \mathtt{Real},\ \mathtt{String}, \ldots \text{ (basic types)}$$

Branching constructs assume $I \neq \emptyset$ and couples $\mathbf{p}_i : \mathsf{m}_i$ to be pairwise distinct. Replicated types R assume a pool of labels distinct from their continuations.

A *replicated type* R defines the protocol of a server. Type $!_{i\in I}\mathbf{p}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i$ denotes the receipt of requests labelled m_i from $\mathbf{p}_i$ carrying payload types $\tilde{B}_i$ having continuation types S_i. Replicated types never appear guarded and always have linear continuations.

Session types S describe the protocol of a *linear* process. The *selection* and *branching* types ($\oplus$ and $\&$) detail possible sends and receives, indicating direction and content of payloads. Branching types may optionally include a failure-handling *timeout branch* $\circlearrowright . S$, where S details the protocol to employ upon assuming a failure. As in processes, types also have a notion of $\boxed{\text{runtime}}$ only *parallel composition*, identifying the protocols of continuations pulled out of a replicated receive. The **end** type denotes termination of a party's protocol.

Message types M are used to type messages in a buffer. They record the direction of communication, as well as the chosen branching label and types of its payload. Lastly, B represents a range of assumed *basic types*.

Definition 7 (Contexts). *Context Γ is **unrestricted** and maps variables to basic types and roles to replicated types. Context Δ is **linear** and maps roles to session types. Context Θ is **affine** and holds a multiset of message types M.*

$$\Gamma ::= \emptyset \mid \mathbf{p} : R, \Gamma \mid x : B, \Gamma \qquad \Delta ::= \emptyset \mid \mathbf{p} : S, \Delta \qquad \Theta ::= \{M_1, \ldots, M_n\}$$

$\boxed{\text{Context update}}$

$$\frac{}{\Delta + \emptyset = \Delta} \qquad \frac{\Delta_1 + \Delta_2 = \Delta_3}{\Delta_1 + \Delta_2, \mathsf{p} : S = \Delta_3, \mathsf{p} : S} \text{ if } \mathsf{p} \notin \mathsf{dom}(\Delta_1)$$

$$\frac{\Delta_1 + \Delta_2 = \Delta_3}{\Delta_1, \mathsf{p} : S_1 + \Delta_2, \mathsf{p} : S_2 = \Delta_3, \mathsf{p} : S_1 \mid S_2}$$

$\boxed{\text{Context splitting}}$

$$\frac{}{\emptyset = \emptyset \cdot \emptyset} \qquad \frac{\Delta = \Delta_1 \cdot \Delta_2}{\Delta, \mathsf{p} : S = \Delta_1, \mathsf{p} : S \cdot \Delta_2} \qquad \frac{\Delta = \Delta_1 \cdot \Delta_2}{\Delta, \mathsf{p} : S = \Delta_1 \cdot \Delta_2, \mathsf{p} : S}$$

$$\frac{\Delta = \Delta_1 \cdot \Delta_2}{\Delta, \mathsf{p} : S_1 \mid S_2 = \Delta_1, \mathsf{p} : S_1 \cdot \Delta_2, \mathsf{p} : S_2}$$

Fig. 2. Context addition and splitting.

Updating *and* ***splitting*** *operations are defined for* Δ *by the rules in Fig. 2. Context* ***composition*** Γ, Γ' *(resp.* Δ, Δ'*) is defined iff* $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$ *(resp.* $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \emptyset$*).*

Figure 2 defines two relations on Δ. *Context addition* joins two contexts by performing a union on their contents (in the case that there are no conflicts in their domains). If their domains are not unique, then the types are placed in parallel, indicating a role employing multiple active session types (this is explained in more detail after introducing context reduction, *cf.* Definition 8). Context *splitting* extracts a piece of a larger context. Notably, types placed in parallel may be split using this operation; in other cases splitting functions similar to context composition.

Definition 8 (Context Reduction). *An action* α *is given by*

$$\alpha ::= \mathsf{p}{\oplus}\mathsf{q}{:}\mathsf{m} \mid \mathsf{p}{,}\mathsf{q}{:}\mathsf{m} \mid \text{\textcircled{\tiny$\prime$}}\mathsf{p}$$

read as (left to right) ***output***, ***communication***, *and* ***timeout***. *Context* ***transition*** $\xrightarrow{\alpha}$ *is defined by the Labelled Transition System (LTS) in Fig. 3. Context* ***reduction*** $\Gamma ; \Delta ; \Theta \to \Gamma ; \Delta' ; \Theta'$ *is defined iff* $\Gamma ; \Delta ; \Theta \xrightarrow{\alpha} \Gamma ; \Delta' ; \Theta'$ *for some* α. *We write* $\Gamma ; \Delta ; \Theta \to$ *iff* $\exists \Delta', \Theta'$ *s.t.* $\Gamma ; \Delta ; \Theta \to \Gamma ; \Delta' ; \Theta'$*; and* $\to^*$ *for its transitive and reflexive closure.*

Context reduction (Definition 8) models type-level communication by means of the LTS in Fig. 3. Transition [Δ-⏲] allows a role p with a defined timeout to transition to the timeout continuation by firing a ⏲p action. Transition [Δ-$\oplus$] is a synchronisation action between a selection type and the type buffer Θ.

$$\Delta\text{-}\circlearrowright \quad \frac{}{\Gamma \,;\, \Delta \cdot \mathbf{p} : \&_{i \in I} \mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i, \circlearrowright .\, S' \,;\, \Theta \xrightarrow{\circlearrowright_{\mathbf{p}}} \Gamma \,;\, \Delta + \mathbf{p} : S' \,;\, \Theta}$$

$$\Delta\text{-}\oplus \quad \frac{S = \oplus_{i \in I} \mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i \qquad k \in I}{\Gamma \,;\, \Delta \cdot \mathbf{p} : S \,;\, \Theta \xrightarrow{\mathbf{p} \oplus \mathbf{q}_k : \mathsf{m}_k} \Gamma \,;\, \Delta + \mathbf{p} : S_k \,;\, \Theta \cdot (\mathbf{p} \to \mathbf{q}_k, \mathsf{m}_k(\tilde{B}_k))}$$

$$\Delta\text{-C} \quad \frac{S = \&_{i \in I} \mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i[, \circlearrowright .\, S'] \qquad k \in I}{\Gamma \,;\, \Delta \cdot \mathbf{p} : S \,;\, \Theta \cdot (\mathbf{q}_k \to \mathbf{p}, \mathsf{m}_k(\tilde{B}_k)) \xrightarrow{\mathbf{q}_k, \mathbf{p} : \mathsf{m}_k} \Gamma \,;\, \Delta + \mathbf{p} : S_k \,;\, \Theta}$$

$$\Gamma\text{-!C} \quad \frac{R = !_{i \in I} \mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) \,.\, S_i \qquad k \in I}{\Gamma, \mathbf{p} : R \,;\, \Delta \,;\, \Theta \cdot (\mathbf{q}_k \to \mathbf{p}, \mathsf{m}_k(\tilde{B}_k)) \xrightarrow{\mathbf{q}_k, \mathbf{p} : \mathsf{m}_k} \Gamma, \mathbf{p} : R \,;\, \Delta + \mathbf{p} : S_k \,;\, \Theta}$$

Fig. 3. Type LTS

Effectively, a role with a send type can transition to its continuation by firing any of the paths indicated in the selection ($\mathbf{p} \oplus \mathbf{q}_k : \mathsf{m}_k$) and adding the message into the buffer context. On the receiving end, a role with a branch type can consume a message from the type buffer to model a communication action via transition [Δ-C]. Communication with replicated servers is handled seperately by transition [Γ-!C] . This rule allows a communication action to be fired when a replicated type in Γ can receive a message in the buffer. This transition has no effect on Γ (since it is an unrestricted context) and instead updates the linear context Δ with the continuation of the replicated receive. This is why types require runtime parallel composition, and context updating and splitting operations (Fig. 2), as multiple requests may be made to a replicated receive.

3.2 Typing Rules

Protocols defined in MAGπ! types are used in type judgements (Definition 9) to check whether network implementations conform to their specifications.

Definition 9 (Typing Judgement). *Type contexts are used in judgements as $\Gamma \,;\, \Delta \,;\, \Theta \vdash \mathcal{N}$, inductively defined by the rules in Fig. 4. To improve readability, empty type contexts are omitted from rules.*

Definition 10 (End Predicate). *A context Δ is* **end***-typed, by:*

$$\frac{\forall i \in 1..n : S_i = \mathbf{end}}{\mathsf{end}(\mathbf{p}_1 : S_1 \cdot \ldots \cdot \mathbf{p}_n : S_n)}$$

Typing rules [T-S], [T-Var], [T-Val] are auxiliary judgements typing linear roles, variables and values. A role $\mathbf{p}$ of type S is typed by a linear context

$$\frac{}{\Gamma\,;\mathbf{p} : S \vdash \mathbf{p} : S}\ \text{T-S} \qquad \frac{\Gamma(x) = B}{\Gamma \vdash x : B}\ \text{T-VAR} \qquad \frac{v \in B}{\Gamma \vdash v : B}\ \text{T-VAL} \qquad \frac{\mathsf{end}(\Delta)}{\Gamma\,;\Delta \vdash 0}\ \text{T-0}$$

$$\frac{\begin{array}{c}\Gamma\,;\Delta \vdash \mathbf{p} : \oplus_{i\in I}\mathbf{q}_i : \mathsf{m}_i(B_{i1},\ldots,B_{in})\,.\,S_i\\ k \in I \qquad \forall j \in 1..n : \Gamma \vdash c_j : B_{kj} \qquad \Gamma\,;\mathbf{p} : S_k \vdash P\end{array}}{\Gamma\,;\Delta \vdash \mathbf{p} \triangleleft \mathbf{q}_k \oplus \mathsf{m}_k\langle c_1,\ldots,c_n\rangle\,.\,P}\ \text{T-}\oplus$$

$$\frac{\begin{array}{c}\Gamma\,;\Delta \vdash \mathbf{p} : \&_{i\in I}\mathbf{q}_i : \mathsf{m}_i(B_{i1},\ldots,B_{in})\,.\,S_i[,\circlearrowright.\,S']\\ \forall i \in I : \Gamma, y_{i1} : B_{i1},\ldots,y_{in} : B_{in}\ ;\ \mathbf{p} : S_i \vdash P_i \qquad [\Gamma\,;\mathbf{p} : S' \vdash P']\end{array}}{\Gamma\,;\Delta \vdash \mathbf{p} \triangleleft \&_{i\in I}\mathbf{q}_i : \mathsf{m}_i(y_{i1},\ldots,y_{in})\,.\,P_i[,\circlearrowright.\,P']}\ \text{T-}\&$$

$$\frac{\begin{array}{c}\Gamma(\mathbf{p}) = !_{i\in I}\mathbf{q}_i : \mathsf{m}_i(B_{i1},\ldots,B_{in})\,.\,S_i\\ \forall i \in I : \Gamma, y_{i1} : B_{i1},\ldots,y_{in} : B_{in}\ ;\ \mathbf{p} : S_i \vdash P_i\end{array}}{\Gamma \vdash \mathbf{p} \triangleleft !_{i\in I}\mathbf{q}_i : \mathsf{m}_i(y_{i1},\ldots,y_{in})\,.\,P_i}\ \text{T-!}$$

$$\frac{\Gamma\ ;\ \Delta\ ;\ \emptyset \vdash \mathcal{N} \qquad \Gamma\ ;\ \emptyset\ ;\ \Theta \vdash \mathcal{B}}{\Gamma\ ;\ \Delta\ ;\ \Theta \vdash \mathcal{N} \mid\mid \mathcal{B}}\ \text{T-}\mid\mid_1 \qquad \frac{\Gamma\,;\Delta_1 \vdash \mathcal{N}_1 \qquad \Gamma\,;\Delta_2 \vdash \mathcal{N}_2}{\Gamma\,;\Delta_1,\Delta_2 \vdash \mathcal{N}_1 \mid\mid \mathcal{N}_2}\ \text{T-}\mid\mid_2$$

$$\frac{\Gamma\,;\Delta_1 \vdash \mathbf{p} \triangleleft \mathcal{P}_1 \qquad \Gamma\,;\Delta_2 \vdash \mathbf{p} \triangleleft \mathcal{P}_2}{\Gamma\,;\Delta_1 \cdot \Delta_2 \vdash \mathbf{p} \triangleleft \mathcal{P}_1 \mid \mathcal{P}_2}\ \text{T-}\mid \qquad \frac{}{\Gamma\ ;\ \emptyset\ ;\ \Theta \vdash \emptyset}\ \text{T-EMPTY}$$

$$\frac{\forall j \in 1..n : \Gamma \vdash v_j : B_j \qquad \Gamma\ ;\ \emptyset\ ;\ \Theta \vdash \mathcal{B}}{\Gamma\ ;\ \emptyset\ ;\ \Theta \cdot (\mathbf{p} \to \mathbf{q}, \mathsf{m}(B_1,\ldots,B_n)) \vdash \mathcal{B} \cdot \langle \mathbf{p} \to \mathbf{q}, \mathsf{m}\langle v_1,\ldots,v_n\rangle\rangle}\ \text{T-BUF}$$

Fig. 4. Typing rules.

containing exactly a mapping of $\mathbf{p}$ to S; variables are typed to a basic type if that mapping is held by Γ; and values are typed to a basic type if they are constants of that type. The empty process **0** is typed by [T-0] if the linear context is **end**-typed (Definition 10), *i.e.*, Δ only contains roles mapped to **end**.

The send process $\mathbf{p} \triangleleft \mathbf{q}_k \oplus \mathsf{m}_k\langle c_1,\ldots,c_n\rangle\,.\,P$ is well typed by [T-$\oplus$] if: Δ can map $\mathbf{p}$ to a selection type containing the path chosen by the process; Γ verifies all payloads with their types indicated in the session type; and the continuation type can check the continuation process.

The receive process $\mathbf{p} \triangleleft \&_{i\in I}\mathbf{q}_i : \mathsf{m}_i(y_{i1},\ldots,y_{in})\,.\,P_i[,\circlearrowright.\,P']$ is well typed by [T-&] if: Δ maps $\mathbf{p}$ to a branch with all the same paths contained in I; the payloads and continuation types of every path in the branch can type all process continuations P_i; and if a timeout process P' is defined, then it must be typed under a timeout branch in the session type.

Replicated receive $\mathbf{p} \triangleleft !_{i \in I} \mathbf{q}_i : \mathsf{m}_i(y_{i1}, \ldots, y_{in}) . P_i$ is typed using [T-!] in a similar manner to [T-&] ; the type of $\mathbf{p}$ instead lives in the unrestricted context.

Network composition is typed by [T- $\|_1$] and [T- $\|_2$] . The former separates the linear context to be used on processes and the buffer context to be used on the network buffer; the latter splits context domains to type different roles in the network. Process-level composition is typed via [T- $|$] which utilises the context splitting operation (Fig. 2) to separate parallel session types.

Network buffers are typed by repeated applications of [T-Buf] , which removes messages from the buffer one at a time if they match a message type in the type buffer. The empty buffer is typed under [T-Empty] , allowing for possible leftover types in Θ. It is key to note that the buffer context is *affine*, as any message that gets dropped at runtime will result in an unused message type.

Example 5 (Interactions with Failure: Types). Due to the generalised nature of the type system, the type judgement alone is not enough to detect the errors that may occur in $\mathcal{N}_f$. This is because the type system does not provide *syntactic* guarantees, but rather should be used in conjunction with exhaustive verification techniques post protocol design (this is standard in generalised MPST [4,17,25]). In fact, network $\mathcal{N}_f$ can be typed under the following contexts:

$$\Gamma \; ; \; \mathbf{p} : \oplus \mathbf{q} : \mathsf{m}(\mathbb{N}) . S, \mathbf{q} : \&\{\mathbf{p} : \mathsf{m}(\mathtt{String}) . S', \; \circlearrowright . S''\} \; ; \; \Theta \cdot (\mathbf{p} \to \mathbf{q}, \mathsf{m}(\mathtt{String}))$$

for some $\Gamma, \Theta, S, S', S''$ assuming that P, P' and P'' are well typed using S, S' and S'' respectively. Note that Γ and Θ can be non-empty since the former is unrestricted and the latter is affine. In contrast, the linear context must be exactly as stated above. We now need a way to determine this protocol as unsafe.

4 Songs About Songs

Unlike most session type theories, *generalised* MPST do not syntactically guarantee any properties on the processes they type. Rather, they provide a framework for *exhaustively checking* runtime properties on the type context, from which process-level properties may be inferred. This seemingly unconventional approach to session types was discovered to be *more expressive* than its syntactic counterpart w.r.t. the amount of well-typed programs it can capture [25]. Furthermore, its generalised nature allows for fine-tuning based on specific requirements of its applications. Informally, generalisation of the type system works by proving the metatheory parametric of a safety property; *i.e.,* all theorems proved and presented assume that the type contexts are *safe* (Sect. 4.1). With this assumption we present our main results in Sect. 4.2.

4.1 Type Safety

The technical definition of *safety* refers to the *minimal requirements* on types to guarantee *subjection reduction* (*cf.* Sect. 4.2, Theorem 1). But what does safety even mean for a distributed network with message loss, delays and reordering?

It is impossible for our type system to adopt standard notions of safety which may guarantee properties such as *no unexpected messages* or *correct ordering of messages*, since the failures experienced at runtime can mitigate such guarantees. Hence, the minimal guarantee of safety (Definition 11) in MAGπ! ensures that:

1. timeout branches are always (and only) defined for failure-prone communication between *linear* processes; and
2. if a message eventually reaches its destination, then the expected types of the payload from the recipient should match the data carried on the message.

Definition 11 (Safety Property). *$\varphi_{\mathcal{R}}$ is a safety property on contexts iff:*

φ–R$_1$

$$\varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta \cdot \mathbf{p} : \&_{i\in I}\mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) . S_i\ ;\ \Theta) \text{ implies } \forall i \in I : \{\mathbf{q}_i, \mathbf{p}\} \in \mathcal{R}$$

φ–R$_2$

$$\varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta \cdot \mathbf{p} : \&_{i\in I}\mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) . S_i, \circlearrowright . S'\ ;\ \Theta) \text{ implies } \exists k \in I : \{\mathbf{q}_k, \mathbf{p}\} \notin \mathcal{R}$$

φ–C

$$\varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta \cdot \mathbf{p} : \&_{i\in I}\mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) . S_i[, \circlearrowright . S']\ ;\ \Theta \cdot (\mathbf{q}_k \to \mathbf{p}, \mathsf{m}_k(\tilde{B}')))$$
$$\text{and } k \in I \text{ implies } |\tilde{B}_k| = |\tilde{B}'| \text{ and } \forall j \in 1..|\tilde{B}_k| : B_{kj} = B'_j$$

φ–!C

$$\varphi_{\mathcal{R}}(\Gamma, \mathbf{p} : !_{i\in I}\mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i) . S_i\ ;\ \Delta\ ;\ \Theta \cdot (\mathbf{q}_k \to \mathbf{p}, \mathsf{m}_k(\tilde{B}')))$$
$$\text{and } k \in I \text{ implies } |\tilde{B}_k| = |\tilde{B}'| \text{ and } \forall j \in 1..|\tilde{B}_k| : B_{kj} = B'_j$$

φ– $\to$

$$\forall \Delta' : \varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta\ ;\ \Theta) \text{ and } \Gamma\ ;\ \Delta\ ;\ \Theta \to \Gamma\ ;\ \Delta'\ ;\ \Theta' \text{ implies } \varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta'\ ;\ \Theta')$$

Conditions [φ-r$_1$] and [φ-r$_2$] ensure that timeouts are *only* omitted (resp. defined) when communication is reliable (resp. unreliable). [φ-c] and [φ-!c] require payload types to match for any communication; note that no message is ever incorrectly delivered to a linear channel instead of a replicated (and *vice versa*) because we assume that message labels for replicated receives are not reused in their continuations. The last condition, [φ-!$\to$] , requires *all* possible reductions of safe contexts to also be safe.

Example 6 (Interactions with Failure: Safety). The type contexts presented in Example 5 do not abide by the conditions of $\varphi_{\emptyset}$ and thus are not safe. The types do meet conditions [φ-r$_1$] to [φ-!c] , but fail [φ-!$\to$] . We observe the following traces of the LTS:

$$\cdot \xrightarrow{\mathbf{p},\mathbf{q}:\mathsf{m}} \Gamma\ ;\ \mathbf{p} : \oplus\, \mathbf{q} : \mathsf{m}(\mathbb{N}) . S, \mathbf{q} : S'\ ;\ \Theta$$

$$\cdot \xrightarrow{\mathbf{p}\oplus\mathbf{q}:\mathsf{m}} \Gamma\ ;\ \mathbf{p} : S, \mathbf{q} : \&\{\mathbf{p} : \mathsf{m}(\mathtt{String}) . S',\ \circlearrowright . S''\}\ ;\ \Theta \cdot (\mathbf{p} \to \mathbf{q}, \mathsf{m}(\mathtt{String})) \cdot (\mathbf{p} \to \mathbf{q}, \mathsf{m}(\mathbb{N}))\quad ✗$$

$$\cdot \xrightarrow{\circlearrowright \mathbf{p}} \Gamma\ ;\ \mathbf{p} : \oplus\, \mathbf{q} : \mathsf{m}(\mathbb{N}) . S, \mathbf{q} : S''\ ;\ \Theta \cdot (\mathbf{p} \to \mathbf{q}, \mathsf{m}(\mathtt{String}))$$

The transition over label $\mathbf{p}\oplus\mathbf{q}$:m yields contexts in violation of [φ-c]. This example highlights the impact of message labels in protocol design, as reusing labels may lead to nondeterministic receipt of messages. However, this does not mean that messages with the same label can never be reused—it is possible for this nondeterminism to still be safe w.r.t. Definition 11. *E.g.* consider the types in Example 1 reusing labels ping and pong. This is safe because the protocol has no dependency on receiving messages with the same label in a specific order.

Example 7 (Load Balancer: Types). We type our load balancer using the protocol below in a judgement as $\mathsf{s}:R_{\mathsf{s}}, \mathsf{w}_1:R_{\mathsf{w}_1}, \mathsf{w}_2:R_{\mathsf{w}_2} \;;\; \mathsf{c}:S_{\mathsf{c}}, \mathsf{o}:S_{\mathsf{o}} \;;\; \emptyset \vdash \mathcal{N} \mid\mid \emptyset$ where $\mathcal{N}$ contains the processes from Example 3. The protocol observes the safety property w.r.t. the reliability relation defined in Example 2, as well as with reliability $\{\{\mathsf{c}, \mathsf{o}\}\}$, *i.e.,* even if server-worker communication is unreliable.

$$
\begin{aligned}
R_{\mathsf{s}} &= !\mathsf{c} : \mathsf{req}(\mathbb{N}) \,.\, \oplus \begin{cases} \mathsf{w}_1 : \mathsf{req}(\mathbb{N}) \,.\, \mathbf{end} \\ \mathsf{w}_2 : \mathsf{req}(\mathbb{N}) \,.\, \mathbf{end} \end{cases} \\
R_{\mathsf{w}_1} &= !\mathsf{s} : \mathsf{req}(\mathbb{N}) \,.\, \oplus\, \mathsf{c} : \mathsf{ans}(\mathtt{Real}) \,.\, \mathbf{end} \\
R_{\mathsf{w}_2} &= !\mathsf{s} : \mathsf{req}(\mathbb{N}) \,.\, \oplus\, \mathsf{c} : \mathsf{ans}(\mathtt{Real}) \,.\, \mathbf{end} \\
S_{\mathsf{c}} &= \oplus \mathsf{s} : \mathsf{req}(\mathbb{N}) \,.\, \& \begin{cases} \mathsf{w}_1 : \mathsf{ans}(\mathtt{Real}) \,.\, \oplus\, \mathsf{o} : \mathsf{output}(\mathtt{Real}) \,.\, \mathbf{end} \\ \mathsf{w}_2 : \mathsf{ans}(\mathtt{Real}) \,.\, \oplus\, \mathsf{o} : \mathsf{output}(\mathtt{Real}) \,.\, \mathbf{end} \\ \circlearrowleft .\, \oplus\, \mathsf{o} : \mathsf{err}(\mathtt{String}) \,.\, \mathbf{end} \end{cases} \\
S_{\mathsf{o}} &= \&\{\mathsf{c} : \mathsf{output}(\mathtt{Real}) \,.\, \mathbf{end}, \mathsf{c} : \mathsf{err}(\mathtt{String}) \,.\, \mathbf{end}\}
\end{aligned}
$$

4.2 Type Properties

Our main results are presented below (proof details in the technical report [18]). *Subject reduction* ((Theorem 1) states that any process typed under a safe context remains well-typed and safe after reduction (even in the presence of failures). From this we obtain Corollary 1, stating that timeout branches are only omitted from linear receives if communication is reliable; hence certifying that all processes typed by safe contexts guarantee that no *linear* failure-prone communication goes unhandled. A key contribution of our work is that this corollary is relaxed to *linear* processes instead of *all* processes, since we do not wish for replicated servers to handle dropped client requests.

Theorem 1 (Subject Reduction). *If* $\Gamma \;;\; \Delta \;;\; \Theta \vdash \mathcal{N}$ *with* $\varphi_{\mathcal{R}}(\Gamma \;;\; \Delta \;;\; \Theta)$ *and* $\mathcal{N} \to_{\mathcal{R}} \mathcal{N}'$*, then* $\exists \Delta', \Theta'$ *s.t.* $\Gamma \;;\; \Delta \;;\; \Theta \to^* \Gamma \;;\; \Delta' \;;\; \Theta'$ *and* $\Gamma \;;\; \Delta' \;;\; \Theta' \vdash \mathcal{N}'$ *with* $\varphi_{\mathcal{R}}(\Gamma \;;\; \Delta' \;;\; \Theta')$.

Corollary 1 (Failure Handling Guarantee). *If* $\Gamma \;;\; \Delta \;;\; \Theta \vdash \mathcal{N}$ *with* $\varphi_{\mathcal{R}}(\Gamma \;;\; \Delta \;;\; \Theta)$ *and* $\mathcal{N} \to_{\mathcal{R}}^* \mathsf{p} \triangleleft \&_{i \in I} \mathsf{q}_i : \mathsf{m}_i(\tilde{c}_i) \,.\, P_i \mid \mathcal{P} \mid\mid \mathcal{N}'$*, then* $\forall i \in I : \{\mathsf{p}, \mathsf{q}_i\} \in \mathcal{R}$.

Session fidelity (Theorem 2) states the opposite implication w.r.t. subjection reduction, *i.e.,* processes typed under a safe context can always match *at least one* reduction available to the context.

Theorem 2 (Session Fidelity). *If $\Gamma\ ;\ \Delta\ ;\ \Theta \rightarrow$ and $\Gamma\ ;\ \Delta\ ;\ \Theta \vdash \mathcal{N}$ with $\varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta\ ;\ \Theta)$, then $\exists \Delta', \Theta', \mathcal{N}'$ s.t. $\Gamma\ ;\ \Delta\ ;\ \Theta \rightarrow \Gamma\ ;\ \Delta'\ ;\ \Theta'$ and $\mathcal{N} \rightarrow^{*}_{\mathcal{R}} \mathcal{N}'$ and $\Gamma\ ;\ \Delta'\ ;\ \Theta' \vdash \mathcal{N}'$ with $\varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta'\ ;\ \Theta')$.*

Using this result we can verify properties other than just safety. This is the benefit of the generalised approach to MPST, where instead of forcing protocols to abide by specific properties, types can be checked *a posteriori* to determine any properties they observe. We demonstrate for *deadlock freedom* (Definition 12).

Definition 12 (DF: Networks). *A network $\mathcal{N}$ is deadlock free, written $\mathsf{df}(\mathcal{N})$, iff $\mathcal{N} \rightarrow^{*} \mathcal{N}' \not\rightarrow$ implies either*

1. *$\mathcal{N}' \equiv \mathbf{0} \mid\mid \mathcal{B}$; or*
2. *$\mathcal{N}' \equiv\ \mathcal{N}'_1 \mid\mid \cdots \mid\mid \mathcal{N}'_n \mid\mid \mathcal{B}$ s.t. $\forall i \in 1..n : \mathcal{N}'_i = \mathbf{p}_i \triangleleft !_{j\in J}\mathbf{q}_j : \mathsf{m}_j(\tilde{x_j})\,.\,P_j$.*

A deadlock free network is one that only gets stuck when all processes reach $\mathbf{0}$, or when the only non-$\mathbf{0}$ processes left in the network are servers. (Note, the buffer is allowed to be non-empty because of message delays.) We define deadlock freedom on types in Definition 13, stating that type contexts are deadlock free if they only get stuck when the linear context is **end**-typed.

Definition 13 (DF: Types). *Contexts $\Gamma\ ;\ \Delta\ ;\ \Theta$ are deadlock free, written $\mathsf{df}(\Gamma\ ;\ \Delta\ ;\ \Theta)$, iff $\Gamma\ ;\ \Delta\ ;\ \Theta \rightarrow^{*} \Gamma\ ;\ \Delta'\ ;\ \Theta' \not\rightarrow$ implies $\mathsf{end}(\Delta')$.*

Proposition 1 (Property Verification: DF). *If $\Gamma\ ;\ \Delta\ ;\ \Theta \vdash \mathcal{N}$ with $\varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta\ ;\ \Theta)$, then $\mathsf{df}(\Gamma\ ;\ \Delta\ ;\ \Theta)$ implies $\mathsf{df}(\mathcal{N})$.*

Lastly, in Proposition 1 we state that deadlock free contexts imply deadlock freedom in the networks they type, a result which follows from Theorem 2.

Decidability. Asynchronous generalised MPST are known to be undecidable in general [17,25]. This stems from the fact that session types with asynchronous buffers can encode Turing machines [3, Theorem 2.5]. However, we note that this simulation relies on buffers with queue semantics and tail-recursion; whereas our type system uses bag buffers and replication. Comparing the expressive power of recursion and replication, previous studies show that for π-calculi with communication of free names the two are equally as expressive [22]; whereas without communication of free names (*e.g.* CCS) recursion is strictly more expressive than replication [7]. Thus, we raise the question: "*What is the expressive power of asynchronous session types with bag buffers and replication?*", which we aim to answer in future work.

For now, we present a predicate on type contexts which can be used to determine decidable subsets of the type system. This predicate, called *trivially terminating* (Definition 14) is decidable and guarantees a finite traversable state-space, thus implying decidability of safety (and subsequently property verification).

Definition 14 (Trivially Terminating). *We say* $\Gamma\ ;\ \Delta\ ;\ \Theta$ *are trivially terminating, written* $\mathsf{tt}(\Gamma\ ;\ \Delta\ ;\ \Theta)$*, iff* $\forall \mathbf{p} \in \mathsf{dom}(\Gamma) : \Gamma(\mathbf{p}) = !_{i\in I}\mathbf{q}_i : \mathsf{m}_i(\tilde{B}_i)\,.\,\tilde{S}_i$ where $\forall i \in I : \mathbf{q}_i \notin \mathsf{dom}(\Gamma)$.

Proposition 2 (Decidable Subset). *For any contexts,* $\mathsf{tt}(\Gamma\ ;\ \Delta\ ;\ \Theta)$ *is decidable and* $\mathsf{tt}(\Gamma\ ;\ \Delta\ ;\ \Theta)$ *implies checking* $\varphi_{\mathcal{R}}(\Gamma\ ;\ \Delta\ ;\ \Theta)$ *is decidable.*

5 Encore

Modelling of failures and distributed communication is increasingly becoming a more relevant and widely researched topic within the area of programming languages. We highlight below some key related work, identifying the main differences w.r.t. MAGπ!.

Affine session types [10,14,20] use affine typing to allow sessions to be prematurely cancelled in the event of failure. They may be used in a similar fashion to try-catch blocks, where a main protocol is followed until a possible failure is met and handled gracefully. Similar in approach to MAGπ! is work by Barwell *et al.* [4], where generalised MPST theory is extended to reason about *crash-stop failures*. Where MAGπ! uses timeouts, the previous uses a "crash" message label which can be fed to a receiving process via some assumed failure detection mechanism. Viering *et al.* [27] present an *event-driven and distributed* MPST theory, where a central robust node is assumed and is capable of restarting crashed processes. Chen *et al.* [11] remove the dependency on a reliable node, instead using *synchronisation points* to handle failures as they are detected. Adameit *et al.* [1] consider session types for *link failures* where *default values* act as failure-handling mechanism to substitute lost data. MAGπ! models lower-level failures than all of these works. Most of the aforementioned assume some perfect failure-detection mechanism, whereas MAGπ! embraces timeouts as a weak failure detector to show that some degree of safety can still be achieved. Our theory is designed to operate at a lower level of abstraction, thus often providing weaker guarantees (*e.g.* consider our minimal definition of type safety) in exchange for modeling a wider set of communication failures.

The adoption of replication in MPST theory is a novel contribution of this paper. Replication in broader session types research has been utilised on numerous accounts [8,9,12,23], specifically in work pertaining to Curry-Howard interpretations of linear logic as session types, where the exponential modality from linear logic $!A$ is typically linked to replication from the π-calculus. Disregarding our modeling of failure, the largest difference between these works and ours is that we focus on a multiparty setting, whereas these theories are all based on binary communication. Furthermore, we did not opt to approach our problem from a logic-perspective, as is the main motivation behind this line of research. Instead, we build upon already-standard generalised MPST theory, adapting it towards our problem domain. We do note, however, that exploring a logical approach to replication in MPST (and, in turn, to failures in session types) is an interesting direction for future work. A more related use of replication in types is by Marshall and Orchard [19], where the authors discuss how non-linear types can be used in a controlled fashion to type behaviours such as repeatedly spawning processes. This resembles the semantics of our type system and dynamic definition of replication in our language, where replicated processes (resp. types) can be reused as necessary to pull out linear copies of their continuations. The mentioned work focuses on how to control the use of non-linear types and how this can be utilised with session types in a functional programming language. Our work, on the other hand, uses replication as a means of better modeling client-server interactions and distinguishing between failure-prone communication that should be handled by the recipient or the sender.

On session types for client-server communication, research largely takes the approach of linear-logic correspondences [8,21,23,28]. The topology these works target are of binary sessions between a pool of clients and a single server. In Qian et al. [23] a logic is developed, called CSLL (client-server linear logic), utilising the *coexponential* modality ¡A. The subtle difference between this modality and the exponential $!A$ is that the latter represents an unlimited number of a type A, while the former serves type A as many times as required according to client requests, in a sequential yet still unordered manner. This is very similar to how our type system operates, given that replicated receives only pull out copies of continuations upon communication. Multiple requests induce non-determinism into further reductions, in our work this is seen in the extension of parallel types, which in Qian et al. [23] is observed through hyper-environments. The difference in goal between the work of [23] (followed up by [21]) and ours, is that the mentioned works focus on providing fixed static guarantees on the processes they type (the former work with a focus on deadlock freedom, the latter on weak termination) whilst we take a generalised approach. Our type system does not force programs to be deadlock-free or terminating, but rather requires a less restrictive *safety* property and allows verification of deadlock freedom and termination to be done *a posteriori*—the trade-off being our weaker form of type safety given the failure-prone nature of our setting.

To conclude, we presented MAGπ!, an extension to MAGπ made to use replication (instead of recursion) to express infinite computation—both at the language and type levels. We did so with the aim of better modelling multiparty client-server interactions, where servers are designed to remain infinitely available. Specifically, we find type-level replication to be a clean mechanism for offloading the handling of certain failures from the recipient to the sender—a practical procedure for client-server interactions. We have generalised our theory by proving our metatheoretic results parametric of the largest safety property, allowing for more specific properties to be instantiated and used to verify runtime behaviours. As future work, we plan to investigate more specific properties for verification through our general type system. We aim to explore in detail the decidability of type-level properties and if/how they may be restricted to obtain decidable bounds in cases where they are not. Lastly, we wish to conduct a foundational study of the use of replication in MPST—we anticipate their use for modelling client-server interactions to have further benefit outwith a failure-prone setting.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. A special thanks also goes to Simon Fowler for his guidance and invaluable feedback. Supported by the UK EPSRC New Investigator Award grant EP/X027309/1 "Uni-pi: safety, adaptability and resilience in distributed ecosystems, by construction".

References

1. Adameit, M., Peters, K., Nestmann, U.: Session types for link failures. In: Bouajjani, A., Silva, A. (eds.) FORTE 2017. LNCS, vol. 10321, pp. 1–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60225-7_1
2. Akkoyunlu, E.A., Ekanandham, K., Huber, R.V.: Some constraints and tradeoffs in the design of network communications. In: Browne, J.C., Rodriguez-Rosell, J. (eds.) Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, 19-21 November 1975, pp. 67–74. ACM (1975). https://doi.org/10.1145/800213.806523
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. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_33
6. Brand, J.: Observations on popular antiquities (1780)
7. 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
8. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 229–259. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_9
9. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Math. Struct. Comput. Sci. **26**(3), 367–423 (2016). https://doi.org/10.1017/S0960129514000218
10. 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
11. Chen, T.-C., Viering, M., Bejleri, A., Ziarek, L., Eugster, P.: A type theory for robust failure handling in distributed systems. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 96–113. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_7
12. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, pp. 91–109. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_5

13. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/j.ic.2017.06.002
14. 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
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, 7-12 January 2008, pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
17. Le Brun, M.A., Dardha, O.: Magπ: types for failure-prone communication. In: Wies, T. (ed.) Programming Languages and Systems. ESOP 2023. LNCS, vol. 13990, pp 363–391. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30044-8_14
18. Le Brun, M.A., Dardha, O.: Magπ!: The role of replication in typing failure-prone communication (2024). https://doi.org/10.48550/arXiv.2404.16213
19. Marshall, D., Orchard, D.: Replicate, reuse, repeat: capturing non-linear communication via session types and graded modal types. In: Carbone, M., Neykova, R. (eds.) Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022. EPTCS, vol. 356, pp. 1–11 (2022). https://doi.org/10.4204/EPTCS.356.1
20. Mostrous, D., Vasconcelos, V.T.: Affine sessions. Log. Methods Comput. Sci. **14**(4) (2018). https://doi.org/10.23638/LMCS-14(4:14)2018
21. Padovani, L.: On the fair termination of client-server sessions. In: Kesner, D., Pédrot, P. (eds.) 28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France. LIPIcs, vol. 269, pp. 5:1–5:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.TYPES.2022.5
22. Palamidessi, C., Valencia, F.D.: Recursion vs replication in process calculi: expressiveness. Bull. EATCS **87**, 105–125 (2005)
23. Qian, Z., Kavvos, G.A., Birkedal, L.: Client-server sessions in linear logic. Proc. ACM Program. Lang. **5**(ICFP), 1–31 (2021). https://doi.org/10.1145/3473567
24. Sangiorgi, D., Walker, D.: The Pi-Calculus - a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
25. 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
26. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
27. 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
28. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X