On Asynchronous Session Semantics

Dimitrios Kouzapas*, Nobuko Yoshida*, and Kohei Honda†

*Imperial College London †Queen Mary, University of London

Abstract. This paper studies a behavioural theory of the π-calculus with session types under the fundamental principles of the practice of distributed computing — asynchronous communication which is order-preserving inside each connection (session), augmented with asynchronous inspection of events (message arrivals). A new theory of bisimulations is introduced, distinct from either standard asynchronous or synchronous bisimilarity, accurately capturing the semantic nature of session-based asynchronously communicating processes augmented with event primitives. The bisimilarity coincides with the reduction-closed barbed congruence. We examine its properties and compare them with existing semantics. Using the behavioural theory, we verify that the program transformation of multithreaded into event-driven session based processes, using Lauer-Needham duality, is type and semantic preserving.

1 Introduction

Modern transports such as TCP in distributed networks provide reliable, ordered delivery of messages from a program on one computer to another, once a connection is established. In practical communications programming, two parties start a conversation by establishing a connection over such a transport and exchange semantically meaningful, formatted messages through this connection. The distinction between possibly non order-preserving communications outside of connection and order-preserving ones inside each connection is a key feature of this practice: order preservation allows proper handling of a sequence of messages following an agreed-upon conversation structure, while unordered deliveries across connections enhance asynchronous, efficient bandwidth usage. Further, asynchronous event processing [18] using locally *buffered* messages enables the receiver to asynchronously *inspect* and *consume* events/messages.

This paper investigates semantic foundations of asynchronously communicating processes, capturing these key elements of modern communications programming – distinction between non order-preserving communications outside connections and the order-preserving ones inside each connection, as well as the incorporation of asynchronous inspection of message arrivals. We use the π-calculus augmented with session primitives, buffers and a simple event inspection primitive. Typed sessions capture structured conversations in connections with type safety; while a buffer represents an intermediary between a process and its environment, capturing non-blocking nature of communications, and enabling asynchronous event processing. The formalism is intended to be an idealised but expressive core communications programming language, offering a basis for a tractable semantic study. Our study shows that the combination

of these basic elements for modern communications programming leads to a rich behavioural theory which differs from both the standard synchronous communications semantics and the fully asynchronous one [8], captured through novel equational laws for asynchrony. These laws can then be used as a semantic justification of a well-known program transformation based on Lauer and Needham's duality principle [15], which translates multithreaded programs to their equivalent single-threaded, asynchronous, event-based programs. This transformation is regularly used in practice, albeit in an ad-hoc manner, playing a key role in e.g. high-performance servers. Our translation is given formally, is type-preserving and is backed up by a rigorous semantic justification. While we do not detail in the main sections, the transform is implemented in the session-extension of Java [13, 14], resulting in competitive performance in comparison with more ad-hoc transformations.

Let us outline some of the key technical ideas of the present work informally. In the present theory, the asynchronous order-preserving communications over a connection are modelled as *asynchronous session communication*, extending the synchronous session calculus [9, 24] with *message queues* [5, 6, 12]. A message queue, written $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$, encapsulates input buffer (`i`) with elements $\vec{h}$ and output buffer (`o`) with $\vec{h}'$. Figure below represents the two end points of a session. A message v is first enqueued by a sender $s!\langle v\rangle;P$ at its output queue at s, which intuitively represents a communication pipe extending from the sender's locality to the receiver's. The message will eventually reach the receiver's locality, formalised as its transfer from the sender's output buffer (at s) to the receiver's input buffer (at $\overline{s}$). For a receiver, only when this transfer takes place, a visible (and asynchronous) message reception takes place, since only then the receiver can *inspect* and *consume* the message (as shown in **Remote** below).Note that dequeuing and enqueing actions inside a location are local to each process and is therefore invisible (τ-actions) (**Local** below).

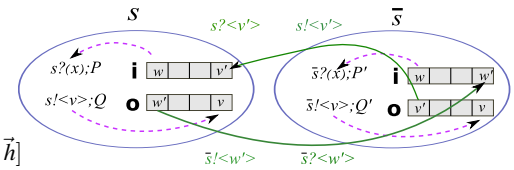
Local (the dashed arrows)

$$s!\langle v\rangle;Q \mid s[\mathtt{o}:\vec{h}] \xrightarrow{\tau} Q \mid s[\mathtt{o}:\vec{h}\cdot v]$$

$$s?(x).P \mid s[\mathtt{i}:w\cdot\vec{h}] \xrightarrow{\tau} P\{w/x\} \mid s[\mathtt{i}:\vec{h}]$$

Remote (the solid arrows)

$$s[\mathtt{i}:\vec{h}] \xrightarrow{s?\langle v\rangle} s[\mathtt{i}:\vec{h}\cdot v] \quad s[\mathtt{o}:v\cdot\vec{h}] \xrightarrow{s!\langle v\rangle} s[\mathtt{o}:\vec{h}]$$



The induced semantics captures the nature of asynchronous observables not studied before. For example, in weak asynchronous bisimilarity in the asynchronous π-calculus ($\approx_a$ in [8, 10]), the message order is not observable ($s!\langle v_1\rangle \mid s!\langle v_2\rangle \approx_a s!\langle v_2\rangle \mid s!\langle v_1\rangle$) but in our semantics, messages for the same destination do *not* commute ($s!\langle v_1\rangle;s!\langle v_2\rangle \not\approx s!\langle v_2\rangle;s!\langle v_1\rangle$) as in the synchronous semantics [20] ($\approx_s$ in [8, 10]); whereas two inputs for different targets commute ($s_1?(x);s_2?(y);P \approx s_2?(x);s_1?(y);P$) since the dequeue action is not observable, differing from the synchronous semantics, $s_1?(x);s_2?(y);P \not\approx_s s_2?(x);s_1?(y);P$.

Asynchronous event-handling [13] introduces further subtleties in observational laws. Asynchronous event-based programming is characterised by reactive flows driven by the detection of *events*, that is *message arrivals* at local buffers. In our formalism, this facility is distilled as an `arrived` predicate: e.g., $Q = \mathtt{if\ arrived}\ s\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ reduces to P_1 if the s input buffer contains one or more message; otherwise Q reduces to P_2. By `arrived`, we can observe the *movement of messages between two locations*.

For example, $Q \mid s[\mathtt{i} : \emptyset] \mid \overline{s}[\mathtt{o} : v]$ is not equivalent with $Q \mid s[\mathtt{i} : v] \mid \overline{s}[\mathtt{o} : \emptyset]$ because the former can reduce to P_2 (since v has not arrived at the local buffer at s yet) while the latter cannot.

Online appendix [23] lists the full definition of Lauer-Needham transformation, the detailed definitions, full proofs and the benchmark results in Session-based Java which demonstrate the potential of the session-type based translation as semantically transparent optimisation techniques.

2 Asynchronous Network Communications in Sessions

2.1 Syntax and Operational Semantics

We use a sub-calculus of the eventful session π-calculus [13], defined below.

$$
\begin{array}{l}
\text{(Identifier) } u ::= a,b \mid x,y \quad k ::= s,\overline{s} \mid x,y \quad n ::= a,b \mid s,\overline{s} \quad \text{(Value) } v ::= \mathtt{tt},\mathtt{ff} \mid a,b \mid s,\overline{s} \\
\text{(Expression) } e ::= v \mid x,y,z \mid \mathtt{arrived}\, u \mid \mathtt{arrived}\, k \mid \mathtt{arrived}\, k\, h \\
\text{(Process) } P,Q ::= u(x).P \mid \overline{u}(x);P \mid k!\langle e\rangle;P \mid k?(x).P \mid k \lhd l;P \mid k \rhd \{l_i : P_i\}_{i\in I} \\
\qquad\qquad \mid \mathtt{if}\, e\, \mathtt{then}\, P\, \mathtt{else}\, Q \mid (\nu\, a)P \mid P \mid Q \mid \mathbf{0} \mid \mu X.P \mid X \\
\qquad\qquad \mid a[\vec{s}] \mid \overline{a}\langle s\rangle \mid (\nu\, s)P \mid s[\mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}] \qquad \text{(Message) } h ::= v \mid l
\end{array}
$$

Values $v, v', \ldots$ include *constants* ($\mathtt{tt}, \mathtt{ff}$), *shared channels* a,b,c and *session channels* s,s'. A session channel denotes one endpoint of a session: s and $\overline{s}$ denote two ends of a single session, with $\overline{\overline{s}} = s$. Labels for branching and selection range over $l, l', \ldots$, variables over x,y,z, and process variables over X,Y,Z. Shared channel identifiers u,u' denote shared channels/variables; session identifiers k,k' are session endpoints and variables. n denotes either a or s. Expressions e are values, variables and the message arrival predicates ($\mathtt{arrived}\, u$, $\mathtt{arrived}\, k$ and $\mathtt{arrived}\, k\, h$: the last one checks for the arrival of the specific message h at k). $\vec{s}$ and $\vec{h}$ stand for vectors of session channels and messages respectively. ε denotes the empty vector.

We distinguish two kinds of asynchronous communications, *asynchronous session initiation* and *asynchronous session communication* (over an established session). The former involves the *unordered* delivery of a *session request message* $\overline{a}\langle s\rangle$, where $\overline{a}\langle s\rangle$ represents an asynchronous message in transit towards an acceptor at a, carrying a fresh session channel s. As in actual network, a request message will first move through the network and eventually get buffered at a receiver's end. Only then a message arrival can be detected. This aspect is formalised by the introduction of a *shared channel input queue* $a[\vec{s}]$, often called *shared input queue* for brevity, which denotes an acceptor's local buffer at a with pending session requests for $\vec{s}$. The intuitive meaning of the endpoint configuration $s[\mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}]$ is explained in Introduction.

Requester $\overline{u}(x);P$ requests a session initiation, while acceptor $u(x).P$ accepts one. Through an established session, output $k!\langle e\rangle;P$ sends e through channel k asynchronously, input $k?(x).P$ receives through k, selection $k \lhd l;P$ chooses the branch with label l, and branching $k \rhd \{l_i : P_i\}_{i\in I}$ offers branches. The $(\nu\, a)P$ binds a channel a, while $(\nu\, s)P$ binds the two endpoints, s and $\overline{s}$, making them private within P. The conditional, parallel composition, recursions and inaction are standard. $\mathbf{0}$ is often omitted. For brevity, one or more components may be omitted from a configuration when they are irrelevant,

writing e.g. $s[\mathtt{i}:\vec{h}]$ which denotes the input part of $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$. The notions of free variables and channels are standard [21]; $\mathrm{fn}(P)$ denotes the set of free channels in P. $\overline{a}\langle s\rangle$, $(\nu\, s)\,P$ and $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$ only appear at runtime. A process without free variables is called *closed* and a closed process without runtime syntax is called *program*.

[Request1]	$\overline{a}(x);P \longrightarrow (\nu\, s)\,(P\{\overline{s}/x\} \mid \overline{s}[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid \overline{a}\langle s\rangle) \quad (s \notin \mathrm{fn}(P))$
[Request2]	$a[\vec{s}] \mid \overline{a}\langle s\rangle \longrightarrow a[\vec{s}\cdot s]$
[Accept]	$a(x).P \mid a[s\cdot\vec{s}] \longrightarrow P\{s/x\} \mid s[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid a[\vec{s}]$
[Send,Recv]	$s!\langle v\rangle;P \mid s[\mathtt{o}:\vec{h}] \longrightarrow P \mid s[\mathtt{o}:\vec{h}\cdot v] \qquad s?(x).P \mid s[\mathtt{i}:v\cdot\vec{h}] \longrightarrow P\{v/x\} \mid s[\mathtt{i}:\vec{h}]$
[Sel,Bra]	$s \lhd l_i;P \mid s[\mathtt{o}:\vec{h}] \longrightarrow P \mid s[\mathtt{o}:\vec{h}\cdot l_i] \quad s \rhd \{l_j:P_j\}_{j\in J} \mid s[\mathtt{i}:l_i\cdot\vec{h}] \longrightarrow P_i \mid s[\mathtt{i}:\vec{h}] \quad (i \in J)$
[Comm]	$s[\mathtt{o}:v\cdot\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'] \longrightarrow s[\mathtt{o}:\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'\cdot v]$
[Areq]	$E[\mathtt{arrived}\ a] \mid a[\vec{s}] \longrightarrow E[\mathtt{b}] \mid a[\vec{s}] \qquad (\lvert\vec{s}\rvert \geq 1) \searrow \mathtt{b}$
[Ases]	$E[\mathtt{arrived}\ s] \mid s[\mathtt{i}:\vec{h}] \longrightarrow E[\mathtt{b}] \mid s[\mathtt{i}:\vec{h}] \qquad (\lvert\vec{h}\rvert \geq 1) \searrow \mathtt{b}$
[Amsg]	$E[\mathtt{arrived}\ s\ h] \mid s[\mathtt{i}:\vec{h}] \longrightarrow E[\mathtt{b}] \mid s[\mathtt{i}:\vec{h}] \qquad (\vec{h} = h\cdot\vec{h}') \searrow \mathtt{b}$

The above table defines the reduction relation over closed terms. The key rules are given in Figure above. We use the standard evaluation contexts $E[\_]$ defined as $E ::= - \mid s!\langle E\rangle;P \mid \mathtt{if}\ E\ \mathtt{then}\ P\ \mathtt{else}\ Q$. The structural congruence $\equiv$ and the rest of the reduction rules are standard. We set $\twoheadrightarrow = (\longrightarrow \cup \equiv)^*$.

The first three rules define the initialisation. In [Request1], a client requests a server for a fresh session via shared channel a. A fresh session channel, with two ends s (server-side) and $\overline{s}$ (client-side) as well as the empty configuration at the client side, are generated and the session request message $\overline{a}\langle s\rangle$ is dispatched. Rule [Request2] enqueues the request in the shared input queue at a. A server accepts a session request from the queue using [Accept], instantiating its variable with s in the request message; the new session is now established. Asynchronous order-preserving session communications are modelled by the next four rules. Rule [Send] enqueues a value in the o-buffer at the *local* configuration; rule [Receive] dequeues the first value from the i-buffer at the local configuration; rules [Sel] and [Bra] similarly enqueue and dequeue a label. The arrival of a message at a remote site is embodied by [Comm], which removes the first message from the o-buffer of the sender configuration and enqueues it in the i-buffer at the receiving configuration.

Output actions are always non-blocking. An input action can block if no message is available at the corresponding local input buffer. The use of the message arrivals can avoid this blocking: [Areq] evaluates $\mathtt{arrived}\ a$ to $\mathtt{tt}$ iff the queue is non-empty ($e \searrow \mathtt{b}$ means e evaluates to the boolean b); similarly for $\mathtt{arrived}\ k$ in [Areq]. [Amsg] evaluates $\mathtt{arrived}\ s\ h$ to $\mathtt{tt}$ iff the buffer is nonempty and its next message matches h.

2.2 Types and Typing

The type syntax follows the standard session types from [9].

(Shared) $U ::= \mathsf{bool} \mid \mathtt{i}\langle S\rangle \mid \mathtt{o}\langle S\rangle \mid \mathsf{X} \mid \mu\,\mathsf{X}.U$ (Value) $T ::= U \mid S$

(Session) $S ::= !(T);S \mid ?(T);S \mid \oplus\{l_i:S_i\}_{i\in I} \mid \&\{l_i:S_i\}_{i\in I} \mid \mu\,\mathsf{X}.S \mid \mathsf{X} \mid \mathtt{end}$

The shared types U include booleans bool (and, in examples, naturals nat); shared channel types $\mathtt{i}\langle S\rangle$ (input) and $\mathtt{o}\langle S\rangle$ (output) for shared channels through which a session

of type S is established; type variables ($\mathsf{X},\mathsf{Y},\mathsf{Z},..$); and recursive types. The IO-types (often called server/client types) ensure a unique server and many clients [11]. In the present work they are used for controlling locality (queues are placed only at the server sides) and associated typed transitions, playing a central role in our behavioural theory. In session types, output type $!(T);S$ represents outputting values of type T, then performing as S. Dually for input type $?(T);S$. Selection type $\oplus\{l_i : S_i\}_{i\in I}$ describes a selection of one of the labels say l_i then behaves as T_i. Branching type $\&\{l_i : S_i\}_{i\in I}$ waits with I options, and behaves as type T_i if i-th label is chosen. End type $\mathtt{end}$ represents the session completion and is often omitted. In recursive type $\mu\mathsf{X}.S$, type variables are guarded in the standard sense.

The judgements of processes and expressions are $\Gamma \vdash P \triangleright \Delta$ and $\Gamma, \Delta \vdash e : T$, with $\Gamma ::= \emptyset \mid \Gamma\cdot u:U \mid \Gamma\cdot X:\Delta$ and $\Delta ::= \emptyset \mid \Delta\cdot a \mid \Delta\cdot k:\mathsf{T} \mid \Delta\cdot s$ where session type is extended to $\mathsf{T} ::= M;S \mid M$ with $M ::= \emptyset \mid \oplus l \mid \&l \mid !(T) \mid ?(T) \mid M;M$ which represents types for values stored in queues (note $\emptyset;S = S$). Γ is called *shared environment*, which maps shared channels and process variables to, respectively, constant types and value types; Δ is called *linear environment* maps session channels to session types and recording shared channels for acceptor's input queues and session channels for end-point queues. The judgement is read: program P is typed under shared environment Γ, uses channels as linear environment Δ. In the expression judgement, expression e has type T under Γ, and uses channels as linear environment Δ. We often omit Δ if it is clear from the context. The typing system is similar with [2, 13], and can be found in online Appendix [23]. We say that Δ *well configured* if $s:S \in \Delta$, then $\overline{s}:\overline{S} \in \Delta$. We define: $\{s:!(T);S\cdot\overline{s}:?(T);S'\} \longrightarrow \{s:S\cdot\overline{s}:S'\}$, $\{s:\oplus\{l_i : S_i\}_{i\in I}\cdot\overline{s}:\&\{l_i : S'_i\}_{i\in I}\} \longrightarrow \{s:S_k\cdot\overline{s}:S'_k\}$ $(k \in I)$, and $\Delta\cup\Delta'' \longrightarrow \Delta'\cup\Delta''$ if $\Delta \longrightarrow \Delta'$.[1]

Proposition 2.1 (Subject Reduction). *if $\Gamma \vdash P\triangleright\Delta$ and $P \twoheadrightarrow Q$ and Δ is well-configured, then we have $\Gamma \vdash Q\triangleright\Delta'$ such that $\Delta \longrightarrow^* \Delta'$ and Δ' is well-configured.*

3 Asynchronous Session Bisimulations and its Properties

3.1 Labelled Transitions and Bisimilarity

Untyped and Typed LTS. This section studies the basic properties of behavioural equivalences. We use the following labels $(\ell, \ell', ...)$:

$$\ell \;::=\; a\langle s\rangle \mid \overline{a}\langle s\rangle \mid \overline{a}(s) \mid s?\langle v\rangle \mid s!\langle v\rangle \mid s!(a) \mid s\&l \mid s\oplus l \mid \tau$$

where the labels denote the session accept, request, bound request, input, output, bound output, branching, selection and the τ-action. $\mathsf{sbj}(\ell)$ denotes the set of free subjects in ℓ; and $\mathsf{fn}(\ell)$ (resp. $\mathsf{bn}(\ell)$) denotes the set of free (resp. bound) names in ℓ. The symmetric operator $\ell \asymp \ell'$ on labels that denotes that ℓ is a dual of ℓ', is defined as: $a\langle s\rangle \asymp \overline{a}\langle s\rangle$, $a\langle s\rangle \asymp \overline{a}(s)$, $s?\langle v\rangle \asymp \overline{s}!\langle v\rangle$, $s?\langle a\rangle \asymp \overline{s}!(a)$, and $s\&l \asymp \overline{s}\oplus l$.

[1] In the following sections, we study semantic properties of typed processes: however these developments can be understood without knowing the details of the typing rules. This is because the properties of the typing system are captured by the typed LTS defined in section 3.1 later.

$$\langle \mathsf{Acc}\rangle \quad a[\vec{s}] \xrightarrow{a\langle s\rangle} a[\vec{s}\cdot s] \qquad \langle \mathsf{Req}\rangle \quad \overline{a}\langle s\rangle \xrightarrow{\overline{a}\langle s\rangle} \mathbf{0} \qquad \langle \mathsf{In}\rangle \quad s[\mathtt{i}:\vec{h}] \xrightarrow{s?\langle v\rangle} s[\mathtt{i}:\vec{h}\cdot v]$$

$$\langle \mathsf{Out}\rangle \quad s[\mathtt{o}:v\cdot\vec{h}] \xrightarrow{s!\langle v\rangle} s[\mathtt{o}:\vec{h}] \qquad \langle \mathsf{Bra}\rangle \quad s[\mathtt{i}:\vec{h}] \xrightarrow{s\&l} s[\mathtt{i}:\vec{h}\cdot l] \qquad \langle \mathsf{Sel}\rangle \quad s[\mathtt{o}:l\cdot\vec{h}] \xrightarrow{s\oplus l} s[\mathtt{o}:h]$$

$$\langle \mathsf{Local}\rangle \frac{P \longrightarrow Q}{P \xrightarrow{\tau} Q} \quad \langle \mathsf{Par}\rangle \frac{P \xrightarrow{\ell} P' \ \mathsf{bn}(\ell)\cap \mathsf{fn}(Q)=\emptyset}{P|Q \xrightarrow{\ell} P'|Q} \quad \langle \mathsf{Tau}\rangle \frac{P \xrightarrow{\ell} P' \ Q \xrightarrow{\ell'} Q' \ \ell \asymp \ell'}{P|Q \xrightarrow{\tau} (\nu\, \mathsf{bn}(\ell,\ell'))(P'|Q')}$$

$$\langle \mathsf{Res}\rangle \frac{P \xrightarrow{\ell} P' \ n \notin \mathsf{fn}(\ell)}{(\nu\, n)P \xrightarrow{\ell} (\nu\, n)P'} \quad \langle \mathsf{OpS}\rangle \frac{P \xrightarrow{\overline{a}\langle s\rangle} P'}{(\nu\, s)P \xrightarrow{\overline{a}(s)} P'} \quad \langle \mathsf{OpN}\rangle \frac{P \xrightarrow{s!\langle a\rangle} P'}{(\nu\, a)P \xrightarrow{s!(a)} P'} \quad \langle \mathsf{Alpha}\rangle \frac{P \equiv_\alpha P' \ P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

Rule $\langle \mathsf{Local}\rangle$ is defined from the reductions by [Request1, Accept, Send, Recv, Bra, Sel, Areq, Ases, Amsg] as well as [Comm] when the communication object is a session (delegation).

In the untyped labelled transition system (LTS) defined above, $\langle \mathsf{Acc}\rangle$/$\langle \mathsf{Req}\rangle$ are for the session initialisation. The next four rules $\langle \mathsf{In}\rangle$/$\langle \mathsf{Out}\rangle$/$\langle \mathsf{Bra}\rangle$/$\langle \mathsf{Sel}\rangle$ say the action is observable when it moves from its local queue to its remote queue. When the process accesses its local queue, the action is *invisible* from the outside, as formalised by $\langle \mathsf{Local}\rangle$. In contrast, $\langle \mathsf{Com}\rangle$ expresses an interaction between two local configutations. This distinction is useful in our later proofs. Other compositional rules are standard. Based on the LTS, we use the standard notations [19] such as $P \stackrel{\ell}{\Longrightarrow} Q$, $P \stackrel{\vec{\ell}}{\Longrightarrow} Q$ and $P \stackrel{\hat{\ell}}{\Longrightarrow} Q$.

We define the typed LTS on the basis of the untyped one, using the type information to control the enabling of actions. This is realised by introducing the *environment transition*, defined below. A transition $(\Gamma,\Delta) \xrightarrow{\ell} (\Gamma',\Delta')$ means that an environment (Γ,Δ) allows an action ℓ to take place, and the resulting environment is (Γ',Δ'), constraining process transitions through the linear and shared environments. This constraint is at the heart of our typed LTS, accurately capturing interactions in the presence of sessions and local buffers. We write $\Gamma_1 \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma_2 \vdash P_2 \triangleright \Delta_2$ if $P_1 \xrightarrow{\ell} P_2$ and $(\Gamma_1,\Delta_1) \xrightarrow{\ell} (\Gamma_2,\Delta_2)$ with $\Gamma_i \vdash P_i \triangleright \Delta_i$. Similarly for other transition relations.

$$\begin{array}{rr}
\Gamma(a) = \mathtt{i}\langle S\rangle, a \in \Delta, s \text{ fresh} \Rightarrow & (\Gamma,\Delta) \xrightarrow{a\langle s\rangle} (\Gamma,\Delta\cdot s:\overline{S}) \\
\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta \Rightarrow & (\Gamma,\Delta) \xrightarrow{\overline{a}\langle s\rangle} (\Gamma,\Delta) \\
\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta, s \text{ fresh} \Rightarrow & (\Gamma,\Delta) \xrightarrow{\overline{a}(s)} (\Gamma,\Delta\cdot s:S) \\
\Gamma \vdash v:U \text{ and } U \neq \mathtt{i}\langle S'\rangle \text{ and } \overline{s} \notin \mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:\mathbin{!}(U);S) \xrightarrow{s!\langle v\rangle} (\Gamma,\Delta\cdot s:S) \\
\overline{s} \notin \mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:\mathbin{!}(\mathtt{o}\langle S'\rangle);S) \xrightarrow{s!(a)} (\Gamma\cdot a:\mathtt{o}\langle S'\rangle,\Delta\cdot s:S) \\
\Gamma \vdash v:U \text{ and } U \neq \mathtt{i}\langle S'\rangle \text{ and } \overline{s} \notin \mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:?(U);S) \xrightarrow{s?\langle v\rangle} (\Gamma,\Delta\cdot s:S) \\
\overline{s} \notin \mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:\oplus\{l_i:S_i\}_{i\in I}) \xrightarrow{s\oplus l_k} (\Gamma,\Delta\cdot s:S_k) \\
\overline{s} \notin \mathrm{dom}(\Delta) \Rightarrow & (\Gamma,\Delta\cdot s:\&\{l_i:S_i\}_{i\in I}) \xrightarrow{s\&l_k} (\Gamma,\Delta\cdot s:S_k) \\
\Delta \longrightarrow \Delta' \Rightarrow & (\Gamma,\Delta) \xrightarrow{\tau} (\Gamma,\Delta')
\end{array}$$

The first rule says that reception of a message via a is possible only when a is input-typed ($\mathtt{i}$-mode) and its queue is present ($a \in \Delta$). The second is dual, saying that an output at a is possible only when a has $\mathtt{o}$-mode and no queue exists. Similarly for a bound output action. The two session output rules ($\ell = s!\langle v\rangle$ and $s!(a)$) are the standard value output and a scope opening rule. The next is for value input. Label input and output are defined similarly. Note that we send and receive only a shared channel which

has o-mode. This is because a new accept should not be created without its queue in the same location. The final rule ($\ell = \tau$) follows the reduction rules defined before Proposition 2.1. The LTS omits delegations since it is not necessary in the bisimulation we consider (due to the notion of localisation, see the next paragraph).

Write $\bowtie$ for the symmetric and transitive closure of $\longrightarrow$ over linear environments. We say a relation on typed processes is a *typed relation* if, whenever it relates two typed processes, we have $\Gamma \vdash P_1 \triangleright \Delta_1$ and $\Gamma \vdash P_2 \triangleright \Delta_2$ such that $\Delta_1 \bowtie \Delta_2$. We write $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ if $(\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2)$ are in a typed relation $\mathscr{R}$. Further we often leave the environments implicit, writing simply $P_1 \mathscr{R} P_2$.

Localisation and Bisimulation. Our bisimulation is a typed relation over those processes which are *localised*, in the sense that they are equipped with all necessary local queues. We say an environment Δ is *delegation-free* if it contains types which are generated by deleting S from value type T in the syntax of types defined in § 2.2 (i.e. either $!(S);S'$ or $?(S);S'$ does not appear in Δ). Similarly for Γ. Now let P be closed and $\Gamma \vdash P \triangleright \Delta$ where Γ and Δ are delegation-free (note that P can perform delegations at hidden channels by $\langle \mathsf{Local} \rangle$). Then we say P is *localised* w.r.t. Γ, Δ if (1) For each $s : S \in \mathsf{dom}(\Delta)$, $s \in \Delta$; and (2) if $\Gamma(a) = \mathtt{i}\langle S \rangle$, then $a \in \Delta$. We say P *is localised* if it is so for a suitable pair of environments. For example, $s?(x);s!\langle x+1\rangle;\mathbf{0}$ is not localised, but $s?(x);s!\langle x+1\rangle;\mathbf{0} \mid s\,[\mathtt{i}:\vec{h}_1, \mathtt{o}:\vec{h}_2]$ is. Similarly, $a(x).P$ is not localised, but $a(x).P \mid a\,[\vec{s}]$ is. By composing buffers at appropriate channels, any typable closed process can become localised. If P is localised w.r.t. (Γ, Δ) then $\Gamma \vdash P \triangleright \Delta \xrightarrow{l} \Gamma' \vdash P' \triangleright \Delta'$ implies P' is localised w.r.t. (Γ', Δ') (in the case of τ-transition, note queues always stay). We can now introduce the reduction congruence and the asynchronous bisimilarity.

Definition 3.1 (Reduction Congruence). We write $P \downarrow a$ if $P \equiv (\nu\, \vec{n})(\overline{a}\langle s \rangle \mid R)$ with $a \notin \vec{n}$. Similarly we write $P \downarrow s$ if $P \equiv (\nu\, \vec{n})(s\,[\mathtt{o} : h \cdot \vec{h}] \mid R)$ with $s \notin \vec{n}$. $P \Downarrow n$ means $\exists P'.P \longrightarrow\!\!\!\!\!\rightarrow P' \downarrow n$. A typed relation $\mathscr{R}$ is *reduction congruence* if it is a congruence and satisfies the following conditions for each $P_1 \mathscr{R} P_2$ whenever they are localised w.r.t. their given environments.

1. $P_1 \Downarrow n$ iff $P_2 \Downarrow n$.
2. Whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ holds, $P_1 \longrightarrow\!\!\!\!\!\rightarrow P_1'$ implies $P_2 \longrightarrow\!\!\!\!\!\rightarrow P_2'$ such that $\Gamma \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ holds with $\Delta_1' \bowtie \Delta_2'$ and the symmetric case.

The maximum reduction congruence which is not a universal relation exists [10] which we call *reduction congruency*, denoted by $\cong$.

Definition 3.2 (Asynchronous Session Bisimulation). A typed relation $\mathscr{R}$ over localised processes is a *weak asynchronous session bisimulation* or often a *bisimulation* for brevity, if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, the following two conditions holds: (1) $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\hat{\ell}}{\Longrightarrow} \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $\Gamma' \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ with $\Delta_1' \bowtie \Delta_2'$ holds and (2) the symmetric case of (1). The maximum bisimulation exists which we call *bisimilarity*, denoted by $\approx$. We sometimes leave environments implicit, writing e.g. $P \approx Q$.

We extend $\approx$ to possibly non-localised closed terms by relating them when their minimal localisations are related by $\approx$ (given $\Gamma \vdash P \triangleright \Delta$, its *minimal localisation* adds empty

queues to P for the input shared channels in Γ and session channels in Δ that are missing their queues). Further $\approx$ is extended to open terms in the standard way [10].

3.2 Properties of Asynchronous Session Bisimilarity

Characterisation of Reduction Congruence. This subsection studies central properties of asynchronous session semantics. We first show that the bisimilarity coincides with the naturally defined reduction-closed congruence [10], given below.

Theorem 3.3 (Soundness and Completeness). $\approx = \cong$.

The soundness ($\approx \subset \cong$) is by showing $\approx$ is congruent. The most difficult case is a closure under parallel composition, which requires to check the side condition $\Delta_1' \bowtie \Delta_2'$ for each case. The completeness ($\cong \subset \approx$) follows [7, § 2.6] where we prove that every external action is definable by a testing process, see [23].

Asynchrony and Session Determinacy. Let us call ℓ an *output action* if ℓ is one of $\overline{a}\langle s\rangle, \overline{a}(s), s!\langle v\rangle, s!(a), s \oplus l$; and an *input action* if ℓ is one of $a\langle s\rangle, s?\langle v\rangle, s\&l$. In the following, the first property says that we can delay an output arbitrarily, while the second says that we can always immediately perform a (well-typed) input.

Lemma 3.4 (Input and Output Asynchrony). *Suppose* $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\Longrightarrow} P' \triangleright \Delta'$.

- (output delay) *If* ℓ *is an output action, then* $\Gamma \vdash P \triangleright \Delta \Longrightarrow \stackrel{\ell}{\longrightarrow} P' \triangleright \Delta'$.
- (input advance) *If* ℓ *is an input action, then* $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\longrightarrow} \Longrightarrow P' \triangleright \Delta'$.

The asynchronous interaction on the session buffers enables inputs to happen before multi-internal steps and outputs to happen after multi-internal steps.

Following [22], we define determinacy and confluence. Below and henceforth we often omit the environments in typed transitions.

Definition 3.5 (Determinacy). We say $\Gamma' \vdash Q \triangleright \Delta'$ is *derivative* of $\Gamma \vdash P \triangleright \Delta$ if there exists $\vec{\ell}$ such that $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} \Gamma' \vdash Q \triangleright \Delta'$. We say $\Gamma \vdash P \triangleright \Delta$ is *determinate* if for each derivative Q of P and action ℓ, if $Q \stackrel{\ell}{\longrightarrow} Q'$ and $Q \stackrel{\hat{\ell}}{\Longrightarrow} Q''$ then $Q' \approx Q''$.

We then extend the above notions to session communications.

Definition 3.6 (Session Determinacy). Let us write $P \stackrel{\ell}{\longrightarrow}_s Q$ if $P \stackrel{\ell}{\longrightarrow} Q$ where if $\ell = \tau$ then it is generated without using [Request1], [Request2], [Accept], [Areq] nor [Amsg] from reduction rules (i.e. a communication is performed without arrival predicates or accept actions). We extend the definition to $\stackrel{\vec{\ell}}{\Longrightarrow}_s$ and $\stackrel{\hat{\ell}}{\Longrightarrow}_s$ etc. We say P is *session determinate* if P is typable, is localised and if $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} Q \triangleright \Delta'$ then $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow}_s Q \triangleright \Delta'$. We call such Q a *session derivative* of P.

We define $\ell_1 \lfloor \ell_2$ ("residual of ℓ_1 after ℓ_2") as (1) $\overline{a}\langle s\rangle$ if $\ell_1 = \overline{a}(s')$ and $s' \in \mathsf{bn}(\ell_2)$; (2) $s!\langle s'\rangle$ if $\ell_1 = s!(s')$ and $s' \in \mathsf{bn}(\ell_2)$; (3) $s!\langle a\rangle$ if $\ell_1 = s!(a)$ and $a \in \mathsf{bn}(\ell_2)$; and otherwise ℓ_1. We write $l_1 \bowtie l_2$ when $l_1 \neq l_2$ and if l_1, l_2 are input actions, $\mathsf{sbj}(l_1) \neq \mathsf{sbj}(l_2)$.

Definition 3.7 (Confluence). $\Gamma \vdash P \triangleright \Delta$ is *confluent* if for each derivative Q of P and ℓ_1, ℓ_2 such that $\ell_1 \bowtie \ell_2$, (i) if $Q \xrightarrow{\ell} Q_1$ and $Q \stackrel{\ell}{\Longrightarrow} Q_2$, then $Q_1 \Longrightarrow Q_1'$ and $Q_2 \Longrightarrow Q_2' \approx Q_1'$; and (ii) if $Q \xrightarrow{\ell_1} Q_1$ and $Q \stackrel{\ell_2}{\Longrightarrow} Q_2$, then $Q_1 \stackrel{\widehat{\ell_2 | \ell_1}}{\Longrightarrow} Q_1'$ and $Q_2 \stackrel{\widehat{\ell_1 | \ell_2}}{\Longrightarrow} Q_2' \approx Q_1'$.

Lemma 3.8. *Let P be session determinate and $\Gamma \vdash P \Longrightarrow Q \triangleright \Delta$. Then $P \approx Q$.*

Theorem 3.9 (Session Determinacy). *Let P be session determinate. Then P is determinate and confluent.*

The following relation is used to prove the event-based optimisation. The proof of the following lemma is by showing $\Longrightarrow \mathscr{R} \Longleftarrow$ with $\Longrightarrow$ determinate is a bisimulation.

Definition 3.10 (Determinate Upto-expansion Relation). Let $\mathscr{R}$ be a symmetric, typed relation such that if $\Gamma \vdash P \triangleright \Delta \; \mathscr{R} \; Q \triangleright \Delta$ and (1) P, Q are determinate; (2) If $\Gamma \vdash P \triangleright \Delta \xrightarrow{l} \Gamma' \vdash P'' \triangleright \Delta''$ then $\Gamma \vdash Q \triangleright \Delta \stackrel{l}{\Longrightarrow} \Gamma' \vdash Q' \triangleright \Delta'$ and $\Gamma' \vdash P'' \triangleright \Delta'' \Longrightarrow \Gamma' \vdash P' \triangleright \Delta'$ with $\Gamma' \vdash P' \triangleright \Delta' \mathscr{R} \; Q' \triangleright \Delta'$; and (3) the symmetric case. Then we call $\mathscr{R}$ a *determinate upto-expansion relation*, or often simply *upto-expansion relation*.

Lemma 3.11. *Let $\mathscr{R}$ be an upto-expansion relation. Then $\mathscr{R} \subset \approx$.*

4 Lauer-Needham Transform

In an early work [15], Lauer and Needham observed that a concurrent program may be written equivalently either in a thread-based programming style (with shared memory primitives) or in an event-based style (with a single-threaded event loop processing messages sequentially with non-blocking handlers). Following this framework and using high-level asynchronous event primitives such as *selectors* [18] for the event-based style, many studies compare these two programming styles, often focusing on performance of server architectures (see [13, § 6] for recent studies on event programming). These implementations implicitly or explicitly assume a *transformation* from a program written in the thread-based style, especially those which generate a new thread for each service request (as in thread-based web servers), to its *equivalent* event-based program, which treats concurrent services using a single threaded event-loop (as in event-based web servers). However the precise semantic effects of such a transformation nor the exact meaning of the associated "equivalence" has not been clarified.

We study the semantic effects of such a transformation using the asynchronous session bisimulation. We first specify both event and thread based programming models and introduce a formal mapping from a thread-based process to their event-based one, following [15]. As a threaded system we assume a server process whose code creates fresh threads at each service invocation. The key idea is to decompose this whole code into distinct smaller code segments, each handling the part of the original code starting from a blocking action. Such a blocking action is represented as reception of a message (input or branching). Then a single global event-loop can treat each message arrival by processing the corresponding code segment combined with an environment, returning to inspect the content of event/message buffers. We first stipulate a class of processes which we consider for our translation. Below $*a(x);P$ denotes an *input replication* abbreviating $\mu X.a(x).(P|X)$.

Definition 4.1 (Server). A *simple server at a* is a closed process $*a(x).P$ with a typing of form $a:\mathtt{i}\langle S\rangle, b_1:\mathtt{o}\langle S_1\rangle, .., b_n:\mathtt{o}\langle S_n\rangle$ where P is sequential (i.e. contains no parallel composition $|$) and is determinate and under any localisation. A simple server is often considered with its localisation with an empty queue $a[\varepsilon]$.

A server spawns an unbounded number of threads as it receives session requests repeatedly. Each thread may initiate other sessions with outside, and its interactions may involve delegations and name passing. Definition 4.1 assumes two conditions: (1) determinacy and (2) sequential processing of each event. A practical example of (1) is a web server which only serves static web pages. As will be discussed later, determinacy plays an essential role in our proofs while sequentiality is for simplicity of the mapping. Given a server $*a(w:S);P \mid a[\varepsilon]$, its translation, which we call *Lauer-Needham transform* or *LN-transform* for short, is written $\mathscr{LN}[\![*a(w:S);P \mid a[\varepsilon]]\!]$(the full mapping is non-trivial and given in [23]). The key elements of $\mathscr{LN}[\![*a(w:S);P]\!]$ follow:

1. A *selector* handles events on message arrival. Its function includes a *selector queue* $q\langle\varepsilon\rangle$ that stores sessions whose arrival is continuously inspected. Its initial element is $\langle a, c_0\rangle$. This data says: "if a message comes at a, jump to the code block (CPS procedure) whose subject is c_0".
2. A collection of *code blocks* $\mathsf{CodeBlocks}\langle a, o, q, \vec{c}\rangle$, CPS procedures handling incoming messages. A code block originates from a threaded server *blocking subterm*, i.e. a subterm starting from an input or a branching.
3. $\mathsf{Loop}\langle o, q\rangle$ implements the *event-loop*. It passes execution from the selector to a code block in CPS style, after a successful arrive inspection from the selector.

We use the standard "select" primitive represented as a process, called *selector* [13]. It stores *a collection of session channels*, with each channel associated with an environment, binding variables to values. It then picks up one of them at which a message arrives, receives that message via that channel and has it be processed by the corresponding code block. Finally it stores the session and the associated environment back in the collection, and moves to the next iteration. Since a selector should handle channels of different types, it uses the `typecase` construct from [13]. $\mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I}$ takes a session endpoint k and a list of cases $(x_i:T_i)$, each binding the free variable x_i of type pattern T_i in P_i. Its reduction is defined as:

$$\mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I} \mid s\,[S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h'}] \longrightarrow P_j\{s/x_j\} \mid s\,[S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h'}]$$

where $j \in I$ such that $(\forall i < j.T_i \not\leq S \wedge T_j \leq S)$ where $\leq$ denotes a subtyping relation. The `typecase` construct finds a match of the session type of the tested channel among the session types in its list, and proceeds with the corresponding process. For the matching to take place, session endpoint configuration syntax is extended with the runtime session typing [13]. The selectors are defined by the following reduction relations:

$$\begin{array}{l}
\mathtt{new\ selector}\ r\ \mathtt{in}\ P \longrightarrow (\nu\, r)(P \mid \mathsf{sel}\langle r, \varepsilon\rangle) \quad \mathtt{register}\langle s', r\rangle;P \mid \mathsf{sel}\langle r, \vec{s}\rangle \longrightarrow P \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle \\
\mathtt{let}\ x = \mathtt{select}(r)\ \mathtt{in}\ \mathtt{typecase}\ x\ \mathtt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[S, \mathtt{i}:\vec{h}] \\
\qquad\qquad \longrightarrow P_i\{s'/x_i\} \mid \mathsf{sel}\langle r, \vec{s}\rangle \mid s'\,[S, \mathtt{i}:\vec{h}] \quad (\vec{h} \neq \varepsilon) \\
\mathtt{let}\ x = \mathtt{select}(r)\ \mathtt{in}\ \mathtt{typecase}\ x\ \mathtt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[\mathtt{i}:\varepsilon] \\
\qquad\qquad \longrightarrow \mathtt{let}\ x = \mathtt{select}(r)\ \mathtt{in}\ \mathtt{typecase}\ x\ \mathtt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle \mid s'\,[\mathtt{i}:\varepsilon]
\end{array}$$

where in the third line S and T_i satisfy the condition for `typecase` in the reduction rule. The last two rules integrate reductions for `typecase` to highlight their combined usage (which is in effect the only way the selection is meaningfully performed). Operator `new selector` r `in` P (binding r in P) creates a new selector $\mathsf{sel}\langle r, \varepsilon\rangle$, named r and with the empty queue ε. Operator $\texttt{register}\langle s', r\rangle; P$ registers a session channel s to r, adding s' to the original queue $\vec{s}$. The next `let` retrieves a registered session and checks the availability to test if an event has been triggered. If so, find the match of the type of s' among $\{T_i\}$ and select P_i; if not, the next session is tested. As proved in [13], these primitives are encodable in the original calculus augmented with `typecase`. The bisimulations and their properties (such as congruency of $\approx$) remain unchanged.

Example 4.1 (Lauer-Needham Transform). As an example of a server, consider:

$$P \;=\; *a(x); x?(y).x!\langle y+1\rangle; x?(z).x!\langle y+z\rangle; \mathbf{0} \mid a[\varepsilon]$$

This process has the session type $?(\mathsf{nat}); !(\mathsf{nat})?(\mathsf{nat}); !(\mathsf{nat})$ at a, and can be read: *a process should first expect to receive (?) a message of type* nat *and send (!) it, then to receive (? again) a* nat*, and finish by sending (!) a result.* We extract the blocking subterms from this process as follows.

Blocking Process	Type at Blocking Prefix
$a(x).x?(y).x!\langle y+1\rangle x?(z).x!\langle y+z\rangle; \mathbf{0}$	$\mathtt{i}\langle ?(\mathsf{nat}); !(\mathsf{nat}); ?(\mathsf{nat}); !(\mathsf{nat})\rangle$
$x?(y).x!\langle y+1\rangle x?(z).x!\langle y+z\rangle; \mathbf{0}$	$?(\mathsf{nat}); !(\mathsf{nat}); ?(\mathsf{nat}); !(\mathsf{nat})$
$x?(z).x!\langle y+z\rangle; \mathbf{0}$	$?(\mathsf{nat}); !(\mathsf{nat})$

These blocking processes are translated into *code blocks* (CodeBlocks) given as:

$$\begin{array}{l} *c_0(y); a(x).\texttt{update}(y,x,x); \texttt{register}\ \langle sel, x, y, c_1\rangle; \overline{o} \mid \\ *c_1(x,y); x?(z); \texttt{update}(y,z,z); x!\langle [\![z]\!]_y + 1\rangle; \texttt{register}\ \langle sel, x, y, c_2\rangle; \overline{o} \mid \\ *c_2(x,y); x?(z'); \texttt{update}(y,z',z'); x!\langle [\![z]\!]_y + [\![z']\!]_y\rangle; \overline{o} \end{array}$$

which processes each message, using environments to record threads' states. The operation $\texttt{update}(y,x,x)$; updates an environment, while `register` stores the blocking session channel, the associated continuation c_i and the current environment y in the selector queue *sel*.

Finally, using these code blocks, the main event-loop denoted Loop, is given as:

$$\begin{array}{l} \mathsf{Loop} = *o.\texttt{let}\ (x,y,z) = \texttt{select from}\ sel\ \texttt{in typecase}\ x\ \texttt{of}\ \{ \\ \qquad \mathtt{i}\langle ?(\mathsf{nat}); !(\mathsf{nat}); ?(\mathsf{nat}); !(\mathsf{nat})\rangle \ : \texttt{new}\ y : \texttt{env in}\ \overline{z}(y) \\ \qquad ?(\mathsf{nat}); !(\mathsf{nat}); ?(\mathsf{nat}); !(\mathsf{nat}) \quad : \overline{z}(x,y) \\ \qquad ?(\mathsf{nat}); !(\mathsf{nat}) \qquad\qquad\qquad\quad : \overline{z}(x,y)\} \end{array}$$

Above `select from` *sel* `in` selects a message from the selector queue *sel*, and treats it in P. The `new` construct creates a new environment y. The `typecase` construct then branches into different processes depending on the session of the received message, and dispatch the task to each code block.

The determinate property allows us to conclude that:

Lemma 4.2. $*a(w:S); R \mid a\,[\varepsilon]$ *is confluent.*

We can now establish the correctness of the transform. The proofs of the following results are found in [23], which use a determinate upto-expansion relation (Definition 3.10) through Lemmas 3.11. First we give a basic equation for the standard event loop, handling events by non-blocking handlers. We use recursive equations of agents for legibility, which can be easily encoded into recursions.

$P_1 = \texttt{if arrived}\, s_1\, \texttt{then}\, (s_1?(x).R_1); P_2\, \texttt{elseif arrived}\, s_2\, \texttt{then}\, (s_2?(x).R_2); P_1\, \texttt{else}\, P_1$

$P_2 = \texttt{if arrived}\, s_2\, \texttt{then}\, (s_2?(x).R_2); P_1\, \texttt{elseif arrived}\, s_1\, \texttt{then}\, (s_1?(x).R_1); P_2\, \texttt{else}\, P_2$

where we assume well-typedness and each of $R_{1,2}$, under any closing substitution and localisation, is determinate and reaches $\mathbf{0}$ after a series of outputs and τ-actions. The sequencing $(s_1?(x).R_1); P_2$ denotes the process obtained by replacing each $\mathbf{0}$ in R_1 with P_2. We can then show $P_1 \approx P_2$ by using the up-to-expansion relation. The following lemma proves its generalisation, elucidating the selectors behaviour in the stateless environment. It says that we can permute the session channels in a selector queue while keeping the same behaviour because of the determinacy of the server's code.

Lemma 4.3. *Let* $P \stackrel{def}{=} \mu X.\texttt{let}\ x = \texttt{select}(r)\ \texttt{in}\ \texttt{typecase}\ x\ \texttt{of}\ \{(x_i : T_i) : R_i; X\}_{i \in I}$ *where each* R_i *is determinate and reaches* $\mathbf{0}$ *after a sequence of non-blocking actions (outputs and* τ*-actions as well as a single input/branching action at* x_i*). The sequencing* $R_i; X$ *is defined as above. Then, assuming typability, we have* $P \mid \mathsf{sel}\langle r, \vec{s}_1 \cdot s_1' \cdot s_2' \cdot \vec{s}_2\rangle \approx P \mid \mathsf{sel}\langle r, \vec{s}_1 \cdot s_2' \cdot s_1' \cdot \vec{s}_2\rangle$.

Thus the selector's behaviour can be matched with the original threaded behaviour step by step in the sense that there is no difference between which event (resp. thread) is selected to be executed first. We conclude:

Theorem 4.4 (Semantic Preservation). *Let* $*a(w : S); R \mid a\,[\varepsilon]$ *be a simple server. Then* $*a(w : S); P \mid a\,[\varepsilon] \approx \mathscr{LN}[\![a(w : S); P \mid a\,[\varepsilon]]\!]$.

5 Discussions

Comparisons with Asynchronous/Synchronous Calculi. We give comprehensive comparisons with other calculi, clarifying the relationship between (1) the session-typed asynchronous π-calculus [8] without queues ($\approx_a$, the asynchronous version of the labelled transition relation for the asynchronous π-calculus), (2) the session-typed synchronous π-calculus [9, 24] without queues ($\approx_s$), (3) the asynchronous session π-calculus with two end-point queues without IO queues [5, 6, 21] ($\approx_2$), and (4) the asynchronous session π-calculus with two end-point IO-queues ($\approx$), i.e. the one developed in this paper. The semantics of (2) is called *non-local* since the output process directly puts the value into the input queue. The transition relation for non-local semantics (2) is defined by replacing the output and selection rules in the LTS relation to:

$$\langle \mathsf{Out}_n \rangle \quad s!\langle v\rangle; P \xrightarrow{s!\langle v\rangle} P \qquad \langle \mathsf{Sel}_n \rangle \quad s \oplus l; P \xrightarrow{s \oplus l} P$$

See [23] for the full definitions and proofs. The following figure summarises distinguishing examples. Non-Blocking Input/Output means inputs/outputs on different channels, while the Input/Output Order-Preserving means that the messages will be received/delivered preserving the order. The final table explains whether Lemma 3.4 (1)

(input advance) or (2) (output delay) is satisfied or not. If not, we place a counterexample (in (4), $\Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon]$ means $[s_1, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid [s_2, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon]$).

	Non-Blocking Input	Non-Blocking Output
(1)	$s_1?(x);s_2?(y);P \approx_a s_2?(y);s_1?(x);P$	$\overline{s_1}\langle v\rangle \mid \overline{s_2}\langle w\rangle \mid P \approx_a \overline{s_1}\langle w\rangle \mid \overline{s_2}\langle v\rangle \mid P$
(2)	$s_1?(x);s_2?(y);P \not\approx_s s_2?(y);s_1?(x);P$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \not\approx_s s_2!\langle w\rangle;s_1!\langle v\rangle;P$
(3)	$s_1?(x);s_2?(y);P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \approx_2$ $s_2?(y);s_1?(x);P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \not\approx_2$ $s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$
(4)	$s_1?(x);s_2?(y);P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \approx$ $s_2?(y);s_1?(x);P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \approx$ $s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon]$

	Input Order-Preserving	Output Order-Preserving
(1)	$s?(x);s?(y);P \approx_a s?(y);s?(x);P$	$\overline{s}\langle v\rangle \mid \overline{s}\langle w\rangle \mid P \approx_a \overline{s}\langle w\rangle \mid \overline{s}\langle v\rangle \mid P$
(2)	$s?(x);s?(y);P \not\approx_s s?(y);s?(x);P$	$s!\langle v\rangle;s!\langle w\rangle;P \not\approx_s s!\langle w\rangle;s!\langle v\rangle;P$
(3)	$s?(x);s?(y);P \mid s[\varepsilon] \not\approx_2 s?(x);s?(y);P \mid s[\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid s[\varepsilon] \not\approx_2 s!\langle w\rangle;s!\langle v\rangle;P \mid s[\varepsilon]$
(4)	$s?(x);s?(y);P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \not\approx$ $s?(x);s?(y);P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \not\approx$ $s!\langle w\rangle;s!\langle v\rangle;P \mid \Pi_{1,2}[s_i, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon]$

	Lemma 3.4 (1)	Lemma 3.4 (2)
(1)	yes	yes
(2)	$(\nu\, s)(s!\langle v\rangle;s'?(x);\mathbf{0} \mid s?(x);\mathbf{0})$	$(\nu\, s)(s!\langle v\rangle;s'!\langle v'\rangle;\mathbf{0} \mid s'?(x);\mathbf{0})$
(3)	yes	$s!\langle v\rangle;\, s'?(x);\mathbf{0} \mid s'[v']$
(4)	yes	yes

Another technical interest is the effects of the `arrived` predicate on these combinations. We define the synchronous and asynchronous π-calculi augmented with the `arrived` predicate and local buffers. For the asynchronous π-calculus, we add $a[\vec{h}]$ and `arrived` a in the syntax, and define the following rules for input and outputs.

$$\overline{a}\langle v\rangle \xrightarrow{\overline{a}\langle v\rangle} \mathbf{0} \qquad a[\vec{h}] \xrightarrow{a\langle h\rangle} a[\vec{h}\cdot h] \qquad \texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q | a[\varepsilon] \xrightarrow{\tau} Q \mid a[\varepsilon]$$

$$a?(x).P \mid a[\vec{h_1}\cdot h_i\cdot \vec{h_2}] \longrightarrow P\{h_i/x\} \mid a[\vec{h_1}\cdot\vec{h_2}] \qquad \texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q | a[\vec{h}] \xrightarrow{\tau} P \mid a[\vec{h}]$$

where, in the last rule, $|\vec{h}| \geq 1$. The above definition precludes the order preservation as the property of transport, but still keeps the non-blocking property as in the asynchronous π-calculus. The synchronous version is similarly defined by setting the buffer size to be one. The non-local version is defined just by adding `arrived` predicate.

Let $Q = \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$ with $P_1 \not\approx P_2$. If the syntax does not include arrival predicates, we have $Q \mid s[\mathtt{i}:\emptyset] \mid \overline{s}[\mathtt{o}:v] \approx Q \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\emptyset]$. In the presence of the arrival predicate, we have $Q \mid s[\mathtt{i}:\emptyset] \mid \overline{s}[\mathtt{o}:v] \not\approx Q \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\emptyset]$ with $e = \texttt{arrived}\ s$. Interestingly in all of the calculi (1–4), the same example as the above, which separate semantics with/without the `arrived`, are effective.

The IO queues provide non-blocking inputs and outputs, while preserving the input/output ordering, which distinguishes the present framework from other known semantics. As a whole, we observe that the present semantic framework is closer to the asynchronous bisimulation (1) $\approx_a$, augmented with order-preserving nature per session. Its key properties arise from local, buffered session semantics and typing. We have also seen the semantic significance of the `arrived` predicates, which enables processes to observe the effects of fine-grained synchronisations.

Related Work. Some of the key proof methods of our work draw their ideas from [22], which study an extension of the confluence theory on the π-calculus. Our work differs in that we investigate the effect of asynchronous IO queues and its relationship to confluence. The work [1] examines expressiveness of various messaging mediums by adding message bags (no ordering), stacks (LIFO policy) and message queues (FIFO policy) in the asynchronous π-calculus [8]. They show that the calculus with the message bags is encodable into the asynchronous π-calculus, but it is impossible to encode the message queues and stacks. Neither the effects of locality, queues, typed transitions, and event-based programming are studied.

Programming constructs that can test the presence of actions or events are studied in the context of the Linda language [3] and CSP [16, 17]. The work [3] measures expressive powers between three variants of asynchronous Linda-like calculi, with a construct for inspecting the output in the tuple space, which is reminiscent of the *inp* predicate of Linda. The first calculus (called *instantaneous*) corresponds to (1) [8], the second one (called *ordered*) formalises emissions of messages to the tuple spaces, and the third one (called *unordered*) models unordered outputs in the tuple space by decomposing one messaging into two stages — emission from an output process and rendering from the tuple space. It shows that the instantaneous and ordered calculi are Turing powerful, while the unordered is not. The work [16] studies CSP with a construct that checks if a parallel process is able to perform an output action on a given channel and a subsequent work [17] investigates the expressiveness of its variants focusing on the full abstraction theorem of the trace equivalence. Our calculi (1,2,3,4) are Turing powerful and we aim to examine properties and applications of the typed bisimilarity characterised by buffered sessions: on the other hand, the focus of [3] is a tuple space where our input/output order preserving examples (which treat different objects with the same session channel) cannot be naturally (and efficiently) defined. The same point applies to [16, 17]. As another difference, the nature of localities has not been considered either in [3, 16, 17] since no notion of a local or remote tuple or environment is defined. Further, none of the above work [1, 3, 16, 17, 22] treats large applications which include these constructs (§ 4) or the performance analysis of the proposed primitives.

Using eventful session types, we have demonstrated that our bisimulation theory is applicable, through the verification of the correctness of the Lauer-Needham transform. The asynchronous nature realised through IO message queues provides a precise analysis of local and eventful behaviours, found in major distributed transports such as TCP. The benchmark results from high-performance clusters in [23] show that the throughput for the thread-eliminated Server implementations in Session Java [13] exhibits higher throughput than the multithreaded Server implementations, justifying the effect of the type and semantic preserving LN-transformation.

As the future work, we plan to investigate bisimulation theories under multiparty session types [12] and a relationship with a linear logic interpretation of sessions, which connects a behavioural theory and permutation laws under locality assumption [4].

Acknowledgements. We thank Raymond Hu for collaborations and the reviewers for their useful comments. The work is partially supported by EP/F003757/1, EP/G015635/1, EP/G015481/1 and EP/F002114/1.

References

1. R. Beauxis, C. Palamidessi, and F. D. Valencia. On the asynchronous nature of the asynchronous pi-calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 473–492. Springer, 2008.
2. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
3. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing three semantics for Linda-like languages. *Theor. Comput. Sci.*, 240(1):49–90, 2000.
4. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.
5. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31, 2007.
6. S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 2009.
7. M. Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.
8. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147, 1991.
9. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
10. K. Honda and N. Yoshida. On reduction-based process semantics. *TCS*, 151(2):437–486, 1995.
11. K. Honda and N. Yoshida. A uniform type structure for secure information flow. *TOPLAS*, 29(6), 2007.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
13. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag, 2010.
14. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
15. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.
16. G. Lowe. Extending csp with tests for availability. *Procedings of Communicating Process Architectures (CPA 2009)*, 2009.
17. G. Lowe. Models for csp with availability information. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 91–105, 2010.
18. S. Microsystems Inc. New IO APIs. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html`.
19. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
20. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Info.& Comp.*, 100(1), 1992.
21. D. Mostrous and N. Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA'09*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.
22. A. Philippou and D. Walker. On confluence in the pi-calculus. In *ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 314–324. Springer, 1997.
23. On-line Appendix of this paper. `http://www.doc.ic.ac.uk/~dk208/semantics.html`.
24. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.