A Distributed Coordination Infrastructure for Attribute-Based Interaction

Yehia Abd Alrahman, Rocco De Nicola, Giulio Garbi, Michele Loreti

A Distributed Coordination Infrastructure for Attribute-based Interaction *

Yehia Abd Alrahman[1][0000−0002−4866−6931], Rocco De Nicola[1][0000−0003−4691−7570], Giulio Garbi[1][0000−0002−2836−4434], and Michele Loreti[2][0000−0003−3061−863X]

[1] IMT School for Advanced Studies Lucca, Lucca, Italy
[2] Università di Camerino, Camerino, Italy

Abstract. Collective-adaptive systems offer an interesting notion of interaction where run-time contextual data are the driving force for interaction. The attribute-based interaction has been proposed as a foundational theoretical framework to model CAS interactions. The framework permits a group of partners to interact by considering their run-time properties and their environment. In this paper, we lay the basis for an efficient, correct, and distributed implementation of the attribute-based interaction framework. First, we present three coordination infrastructures for message exchange, then we prove their correctness, and finally we model them in terms of stochastic processes to evaluate their performance.

Keywords: Attribute-based Interaction, Semantics, Process Calculi.

1 Introduction

Collective Adaptive Systems (CAS) [12] consists of a large number of components that interact anonymously, based on their properties and on contextual data, and combine their behaviours to achieve system-level goals. The boundaries of CAS are fluid and components may enter or leave the system at any time. Components may also adapt their behaviours in response to environmental conditions.

Classical communication paradigms handle the interaction among distributed components by relying on their identities, like in the Actor model [5], or on channel names, like in channel-based binary communication [18] and broadcast communication [16]. However, since identities and channels are totally independent from run-time properties and capabilities of the interacting components, programming collective-adaptive behaviour becomes a tedious task.

To mitigate the shortcomings of the classical paradigms when dealing with CAS, in FORTE'16 [6], we have proposed a kernel calculus, named AbC [1], for modeling CAS interactions. The idea is to permit the construction of formally verifiable CAS systems by relying on a minimal set of interaction primitives. AbC's primitives are attribute-based [4] and abstract from the underlying coordination

* This research has been supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL.

infrastructure (i.e., they are infrastructure-agnostic). They rely on *anonymous* multicast communication where components interact based on mutual interests. Message transmission is non-blocking while reception is not. Each component has a set of attributes to represent its run-time status. Communication actions (both send and receive) are decorated with predicates over attributes that partners have to satisfy to make the interaction possible. The interaction predicates are also parametrised with local attribute values and when values change, the interaction groups do implicitly change, introducing opportunistic interactions.

Basing the interaction on run-time attribute values is indeed a nice idea, but it needs to be supported by a middleware that provides efficient ways for distributing messages, checking attribute values, and updating them. A typical approach is to rely on a centralised broker that keeps track of all components, intercepts every message and forwards it to registered components. It is then the responsibility of each component to decide whether to receive or discard the message. This is the approach used in the Java-based implementation [2] of *AbC*. A similar approach, still based on a centralised broker, is used in the Erlang-based implementation [10]. There however to avoid broadcasts the broker has an attribute registry where components register their attribute values and the broker is now responsible for message filtering.

Clearly, any centralised solution may not scale with CAS dynamics and thus becomes a bottleneck for performance. A distributed approach is definitely preferable for large systems. However, distributed coordination infrastructures for managing the interaction of computational systems are still scarce [15] and/or inefficient [20]. Also the correctness of their overall behaviour is often not obvious. In this paper, we propose an efficient distributed coordination infrastructure for message exchange. We prove its correctness with respect to the original semantics of *AbC* and finally we evaluate its performance in terms of stochastic simulation. Though this paper assumes perfect communication links and does not deal with dropped messages or node's failures, we believe that existing techniques for resilience and failure-recovery can be integrated transparently.

The rest of this paper is structured as follows: In Section 2, we briefly review the *AbC* calculus. In Section 3, we give a full formal account of a distributed coordination infrastructure for *AbC* and its correctness. In Section 4, we provide a detailed performance evaluation and we discuss the results. Finally, Section 5 concludes the paper and surveys related works.

2 *AbC* in a Nutshell

In this section we briefly introduce the *AbC* calculus by means of a running example. We give *an intuition* of how to model a distributed variant of the well known *Graph Colouring Problem* [14] using *AbC* constructs. We render the problem as a typical CAS scenario where a collective of agents, executing the same code, collaborate to achieve a system-level goal without any centralised control. The presentation is intended to be intuitive and full details concerning the example, the syntax, and the semantics of *AbC* can be found in [1, 3].

The problem consists of assigning a *colour* (an integer) to each vertex in a graph while avoiding that two neighbours get the same colour. The algorithm consists of a sequence of rounds for colour selection. At the end of each round at least one vertex is assigned a colour. A vertex, with identity *id*. uses messages of the form ("*try*", c, r, id) to inform its neighbours that at round r it wants to select colour c and messages of the form ("*done*", c, r, id) to communicate that colour c has been definitely chosen at the end of round r. At the beginning of a round, each vertex selects a colour and sends a *try*-message to all of its neighbours N. A vertex also collects *try*-messages from its neighbours. The selected colour is assigned to a vertex only if it has the greatest id among those that have selected the same colour in that round. After the assignment, a *done*-message (associated with the current round) is sent to neighbours.

***AbC* Syntax.** An *AbC component* (C), is either a process P associated with an *attribute environment* Γ (denoted by $\Gamma{:}P$) or the parallel composition $C_1 \| C_2$ of components. The *attribute environment* Γ is a partial map from attribute identifiers $a \in \mathscr{A}$ to values $v \in \mathscr{V}$. Values can be numbers, strings, tuples, etc.

$$C ::= \Gamma{:}P \quad | \quad C_1 \| C_2$$

Example (step 1/4): Each vertex, in the colouring scenario, can be modelled in *AbC* as a component of the form $C_i = \Gamma_i : P_C$. The overall system is the parallel composition of vertices (i.e., $C_1 \| C_2 \| , \ldots, \| C_n$).

The attribute environment of a vertex Γ_i relies on the following attributes to control the behaviour of a vertex: The attribute "round" stores the current round while "used" is a set, registering the colours used by neighbours. The attribute "$\mathsf{counter}$" counts the number of messages collected by a component while "send" is used to enable/disable forwarding of messages to neighbours. Attribute "$\mathsf{assigned}$" indicates if a vertex is assigned a colour while "colour" is a colour proposal. Finally, attributes id and N are used to represent the vertex *id* and the set of *neighbours*, respectively. These attributes initially have the following values: $\mathsf{round} = 0$, $\mathsf{used} = \emptyset$, $\mathsf{send} = \mathsf{tt}$, and $\mathsf{assigned} = \mathsf{ff}$.

It should be noted that new values for these attributes can only be learnt by means of message exchange among vertices. □

The behavior of an *AbC* process can be generated by the following grammar:

$$P ::= \mathsf{0} \quad | \quad \alpha.P \quad | \quad [\tilde{a} := \tilde{E}]P \quad | \quad \langle \Pi \rangle P \quad | \quad P_1 + P_2 \quad | \quad P_1 | P_2 \quad | \quad K$$

The process $\mathsf{0}$ denotes the inactive process; $\alpha.P$ denotes a process that executes action α and continues as P; process $[\tilde{a} := \tilde{E}]P$ behaves as P given that its attribute environment is first updated by setting the value of each attribute in the sequence $\tilde{a}$ to the evaluation of the corresponding expression in the sequence $\tilde{E}$. The attribute updates and the first move of P are atomic; $\langle \Pi \rangle P$ denotes an awareness process, it blocks the execution of process P until the predicate Π evaluates to true; the processes $P_1 + P_2$, $P_1 | P_2$, and K are standard for nondeterminism, parallel composition, and process definition respectively. The parallel operator "|" does not allow communication between P_1 and P_2, they

Table 1: AbC Communication Rules

$$\frac{\Gamma{:}P \overset{\lambda}{\longmapsto} \Gamma' : P'}{\Gamma{:}P \xrightarrow{\lambda} \Gamma' : P'}\ \textsc{iComp} \qquad \frac{\Gamma{:}P \overset{\widetilde{\Pi(\tilde{v})}}{\longmapsto} \Gamma{:}P}{\Gamma{:}P \xrightarrow{\Pi(\tilde{v})} \Gamma{:}P}\ \textsc{fComp}$$

$$\frac{C_1 \xrightarrow{\overline{\Pi}(\tilde{v})} C_1' \quad C_2 \xrightarrow{\Pi(\tilde{v})} C_2'}{C_1 \parallel C_2 \xrightarrow{\overline{\Pi}(\tilde{v})} C_1' \parallel C_2'}\ \textsc{Com} \qquad \frac{C_1 \xrightarrow{\Pi(\tilde{v})} C_1' \quad C_2 \xrightarrow{\Pi(\tilde{v})} C_2'}{C_1 \parallel C_2 \xrightarrow{\Pi(\tilde{v})} C_1' \parallel C_2'}\ \textsc{Sync}$$

can only interleave while the parallel operator "$\parallel$" at the component level allows communication between components. The expression $\texttt{this.}\mathsf{b}$ denotes the value of attribute b in the current component.

Example (step 2/4): Process P_C, specifying the behaviour of a vertex is now defined as the parallel composition of these four processes: $P_C \triangleq F \mid T \mid D \mid A$.

Process F forwards *try*-messages to neighbours, T handles *try*-messages, D handles *done*-messages, and A is used for assigning a final colour. □

The AbC communication actions ranged by α can be either $(\tilde{E})@\Pi$ or $\Pi(\tilde{x})$. The construct $(\tilde{E})@\Pi$ denotes an output action, it evaluates the sequence of expressions $\tilde{E}$ under the local attribute environment and then sends the result to the components whose attributes satisfy the predicate Π. Furthermore, $\Pi(\tilde{x})$ denotes an input action, it binds to sequence $\tilde{x}$ the corresponding received values from components whose *communicated attributes* or values satisfy Π.

Example (step 3/4): We further specify process F and a part of process T.

$$F \triangleq \langle \mathsf{send} \wedge \neg \mathsf{assigned} \rangle [\mathsf{colour} := \min\{i \notin \texttt{this.}\mathit{used}\},\ \mathsf{send} := \mathsf{ff}] \\ (\text{“}\mathit{try}\text{”}, \texttt{this.}\mathsf{colour}, \texttt{this.}\mathsf{round}, \texttt{this.}\mathsf{id})@(\texttt{this.}\mathit{id} \in \mathsf{N}).F$$

$$T \triangleq [\mathsf{counter} := \mathsf{counter} + 1] \\ ((x = \text{“}\mathit{try}\text{”}) \wedge (\texttt{this.}\mathsf{id} > l) \wedge (\texttt{this.}\mathsf{round} = z))(x, y, z, l).T \ +\ \ldots$$

In process F, when the value of attribute send becomes true, a new colour is selected, send is turned off, and a message containing this colour and the current round is sent to all the vertices having $\texttt{this.}\mathsf{id}$ as neighbour. The new colour is the smallest colour that has not yet been selected by neighbours, that is $\min\{i \notin \texttt{this.}\mathit{used}\}$. The guard $\neg\mathsf{assigned}$ is used to make sure that vertices with assigned colours do not take part in the colour selection anymore.

Process T receives messages of the form $(\text{“}\mathit{try}\text{”}, c, r, \mathit{id})$. If $r = \texttt{this.}\mathsf{round}$ then the received message has been originated by a vertex performing the same round of the algorithm. The condition $\texttt{this.}\mathsf{id} > l$ means that the sender has an id smaller than the id of the receiver. In this case, the message is ignored (there is no conflict), simply the counter of collected messages ($\texttt{this.}\mathsf{counter}$) is incremented. Other cases, not reported here, e.g., $\texttt{this.}\mathsf{id} < l$, the received colour is recorded to check the presence of conflicts. □

***AbC* Semantics.** The main semantics rules of *AbC* are reported in Table 1. Rule iComp states that a component evolves with (send $\overline{\Pi}(\tilde{v})$ or receive $\Pi(\tilde{v})$, denoted by λ) if its internal behaviour, denoted by the relation $\mapsto$, allows it. Rule fComp states that a component can discard a message $\Pi(\tilde{v})$ if its internal behaviour does not allow the reception of this message by generating the discarding label $\widetilde{\Pi(\tilde{v})}$. Rule Com[3] states that if C_1 evolves to C_1' by sending a message $\overline{\Pi}(\tilde{v})$ then this message should be delivered to C_2 which evolves to C_2' as a result. Note that C_2 can be also a parallel composition of different components. Thus, rule Sync states that multiple components can be delivered the same message in a single transition.

The semantics of the parallel composition operator, in rules Com and Sync in Table 1, abstracts from the underlying coordination infrastructure that mediates the interactions between components and thus the semantics assumes atomic message exchange. This implies that no component can evolve before the sent message is delivered to all components executing in parallel. Individual components are in charge of using or discarding incoming messages. Message transmission is non-blocking, but reception is not. For instance, a component can still send a message even if there is no receiver (i.e., all the target components discard the message); a receive operation can, instead, only take place through synchronisation with an available message. However, if we want to use the attribute-based paradigm to program the interactions of distributed applications, atomicity and synchrony are neither efficient nor applicable.

One solution is to rely on existing protocols for total-order broadcast to handle message exchange. However, these protocols are mostly centralised [9] or rely on consensus [20]. Clearly, centralised solutions have always scalability and efficiency problems. Furthermore, consensus approaches are not only inefficient [20] but also impossible in asynchronous systems in the presence of even a single component's failure [13]. They also assume that components know each other and can agree on a specific order. However, this contradicts the main design principles of the *AbC* calculus where anonymity and openendedness are crucial factors. Since *AbC* components are *agnostic* to the infrastructure, they cannot participate in establishing the total order. Thus, we need an infrastructure that guarantees the total order seamlessly and without involving the interacting components.

The focus of this paper, as we will see later, is on providing an *efficient distributed* coordination infrastructure that behaves in agreement with the parallel composition operator of *AbC*. Thus in Table 1, we only formalised the external behaviour of a component, i.e., its ability to send and receive. The following example shows how interactions are derived based on internal behaviour.

Example (step 4/4): Consider the vertices C_1, C_2, and C_3 where $\Gamma_2(\mathsf{N}) = \{3\}$, $\Gamma_3(\mathsf{N}) = \{1, 4\}$, $\Gamma_3(\mathsf{id}) = 3$, and $\Gamma_3(\mathsf{round}) = 5$. Now C_1 sent a try message:

$$\Gamma_1 : P_C \xrightarrow{\overline{(1 \in \mathsf{N})}(\text{“}try\text{”},3,5,1)} \overbrace{\Gamma_1[\mathsf{colour} \leftarrow\!\shortmid 3, \mathsf{send} \leftarrow\!\shortmid \mathsf{ff}] : P_C'}^{C_1'}$$

[3] For the sake of brevity, we omit the symmetric rule of Com.

We have that C_2 discards this message because $\Gamma_2 \not\models (1 \in \mathsf{N})$ while C_3 accepts the message ($\Gamma_3 \models (1 \in \mathsf{N})$ and the receiving predicate of process T is satisfied). The system evolves with rule Com as follows:

$$C_1 \| C_2 \| C_3 \xrightarrow{\overline{(1 \in \mathsf{N})}(\text{“}try\text{”},3,5,1)} C_1' \| C_2 \| \Gamma_3[\mathsf{counter} \leftarrow\!\!\shortmid \mathsf{counter} + 1] : P_C'[\text{“}try\text{”}/x, 3/y, 5/z, 1/l] \quad \square$$

3 A Distributed Coordination Infrastructure

In this section, we consider three possible coordination infrastructures that we have also implemented[4] in Google Go. We will refer to them as *cluster*-based, *ring*-based, and *tree*-based. These infrastructures behave in agreement with the parallel composition operator of AbC. Our approach consists of labelling each message with an id that is uniquely identified at the infrastructure level. Components execute asynchronously while the semantics of the parallel composition operator is preserved by relying on the unique identities of exchanged messages. In essence, if a component wants to send a message, it sends a request to the infrastructure for a fresh id. The infrastructure replies back with a fresh id and then the component sends a data (the actual) message with the received id. A component receives a data message only when the difference between the incoming data message id and the id of the last received data message equals 1. Otherwise the data message is added to the component waiting queue until the condition is satisfied.

In what follows, we give a full formal account of the proposed infrastructures and also investigate the correctness of the tree-based one. The reason is that we want to avoid redundancy and also because the tree infrastructure is theoretically the most challenging one. Actually, the proofs of correctness for the other infrastructures are simple cases of the tree's one. Moreover, as we will see in Section 4, the tree exhibits better performance characteristics.

Furthermore to provide compact semantics, we use the following definition of a *Configuration*. For the sake of clarity, we will postfix the configuration of a component, an infrastructure, and a server with the letter a, n, and s respectively.

Definition 1 (Configuration). *A configuration C, is a tuple $C = \langle c_1, \ldots, c_n \rangle$ which is commutative. The symbol '...' is formally regarded as a meta-variable ranging over unmentioned elements of the configuration. The explicit '...' is obligatory, and ensures that unmentioned elements of a configuration are never excluded, but they do not play any role in the current context. Different occurrences of '...' in the same context stand for the same set of unmentioned elements.*

We use the reduction relation $\rightsquigarrow_{\triangleright} \subseteq \textsc{Cfig} \times \textsc{Lab} \times \textsc{Cfig}$ to define the semantics of a configuration where Cfig denotes the set of configurations, Lab denotes the set of reduction labels which can be a message m, a silent transition τ, or an empty label, and $\rightsquigarrow_{\triangleright}^*$ denotes the transitive closure of $\rightsquigarrow_{\triangleright}$. Moreover, we will use the following notations:

[4] Go implementations: `https://github.com/giulio-garbi/goat`

- We have two kinds of messages, an AbC message '$_{\text{msg}}$' (i.e., $\Pi(\tilde{v})$) and an infrastructure message 'm'; the latter can be of three different templates: (i) request {'Q', route, dest}, (ii) reply {'R', id, route, dest}, and (iii) data {'D', id, src, dest, msg}. The route field in a request or a reply message is a linked list containing the addresses of the nodes that the message traversed.
- The notation $\stackrel{?}{=}$ denotes a template matching.
- The notation $T[f]$ denotes the value of the element f in T.

Also the following operations will be used: $L.get()$ returns the element at the front of a list/queue, while $L \hookleftarrow m$ returns the list/queue resulting from adding m to the back of L, and $L \backslash x$ removes x from L and returns the rest.

3.1 Infrastructure Component

Now, we formally define a general infrastructure component and its external behaviour. In the following sections, we proceed by formally defining the proposed infrastructures and their behaviours.

Definition 2 (Infrastructure component). *An infrastructure component, a, is defined by the configuration:* $a = \langle addr, nid,\ mid, on, \mathcal{W}, \mathcal{X}, G \rangle$ *where addr refers to its address, nid (initially 0) refers to the id of the next data message to be received, mid (initially -1) refers to the id of the most recent reply, on (initially 0) indicates whether a request message can be sent.* $\mathcal{W}$ *is a priority waiting queue where the top of* $\mathcal{W}$ *is the data message with the least id, and* $\mathcal{X}$ *refers to the address of the parent server. Furthermore,* G *ranges over* $\Gamma : P$ *and* $[\Gamma : P]$ *where* $[\Gamma : P]$ *indicates an* AbC *component in an intermediate state.*

The intermediate state, in Definition 2, is important to allow co-located processes (i.e, $[\Gamma : P_1 | P_2]$ where P_1 is waiting an id to send and P_2 is willing to receive) to interleave their behaviours without compromising the semantics.

The semantics of an infrastructure component is reported in Table 2. Rule Out states that if the AbC component $\Gamma : P$ encapsulated inside an infrastructure component is able to send a message $\Gamma : P \xrightarrow{\overline{\Pi}(\tilde{v})} \Gamma' : P'$, the flag on is set to 1 and $\Gamma : P$ goes into an intermediate state $[\Gamma : P]$. Rule Med states that an intermediate state component can only receive a message $\Pi(\tilde{v})$ if it was able to receive it before the intermediate state. Rule Req states that a component sends a request, to the parent server, only if $on == 1$. In this case, it adds its address to the *route* of the message and resets on to 0. Rule RcvR states that a component receives a reply if the destination field of the reply matches its address; after that mid gets the value of the id received in the reply. Rule Snd states that a component $\Gamma : P$ can send a message $\overline{\Pi}(\tilde{v})$ and evolves to $\Gamma' : P'$ only if $nid == mid$; this implies that a fresh id is received ($mid \neq -1$) and all messages with $m[id] < mid$ have been already received. By doing so, an infrastructure data message, with msg field equals to $\Pi(\tilde{v})$, is sent, nid is incremented, and mid is reset. Rule RcvD states that a component receives a data message from the infrastructure if $m[id] \geq nid$; this is important to avoid duplicate messages. The

Table 2: The semantics of a component

$$\frac{\Gamma : P \xrightarrow{\overline{\Pi}(\tilde{v})} \Gamma' : P'}{\langle on,\ \Gamma : P, \ldots\rangle_a \overset{\tau}{\rightsquigarrow} \langle 1,\ [\Gamma : P], \ldots\rangle_a}\ \textsc{Out} \qquad \frac{\Gamma : P \xrightarrow{\Pi(\tilde{v})} \Gamma' : P'}{[\Gamma : P] \xrightarrow{\Pi(\tilde{v})} [\Gamma' : P']}\ \textsc{Med}$$

$$\frac{on == 1}{\langle addr, on, \mathcal{X}, \ldots\rangle_a \overset{\{\text{`}Q\text{'},\ \{addr\},\ \mathcal{X}\}}{\rightsquigarrow} \langle addr, 0, \mathcal{X}, \ldots\rangle_a}\ \textsc{Req}$$

$$\langle addr, mid, \ldots\rangle_a \overset{\{\text{`}R\text{'},\ id,\ \{\},\ addr\}}{\rightsquigarrow} \langle addr, id, \ldots\rangle_a \quad \textsc{RcvR}$$

$$\frac{nid == mid \qquad \Gamma : P \xrightarrow{\overline{\Pi}(\tilde{v})} \Gamma' : P'}{\langle addr, nid, mid, [\Gamma : P], \mathcal{X}, \ldots\rangle_a \overset{\{\text{`}D\text{'},mid,addr,\mathcal{X},\Pi(\tilde{v})\}}{\rightsquigarrow} \langle addr, nid + 1, -1, \Gamma' : P', \mathcal{X}, \ldots\rangle_a}\ \textsc{Snd}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'},\ id,\ \mathcal{X},\ addr,\ msg\} \qquad id \geq nid}{\langle addr, nid, \mathcal{W}, \mathcal{X}, \ldots\rangle_a \overset{m}{\rightsquigarrow} \langle addr, nid, \mathcal{W} \leftarrow\!\shortmid m, \mathcal{X}, \ldots\rangle_a}\ \textsc{RcvD}$$

$$\frac{m[id] == nid \qquad G \xrightarrow{m[msg]} G'}{\langle nid, G,\ m :: \mathcal{W}', \ldots\rangle_a \overset{\tau}{\rightsquigarrow} \langle nid + 1, G',\ \mathcal{W}', \ldots\rangle_a}\ \textsc{Hnd}$$

message is then added to the priority queue, $\mathcal{W}$. Finally, rule Hnd states that when the id of the message on top of $\mathcal{W}$ matches *nid*, component G is allowed to receive that message; by doing so, *nid* is incremented and m is removed.

3.2 Cluster-based Infrastructure

We consider a set of server nodes, sharing a counter for sequencing messages and one FIFO queue to store messages sent by components. Cluster nodes can have exclusive locks on both the cluster's counter and the queue. Components register directly to the cluster and send messages to be added to the FIFO queue. When a server node retrieves a request from the cluster queue, it replies to the requester with the value of the cluster counter. By doing so, the cluster counter is incremented. If a server retrieves a data message, it forwards the message to all components in the cluster except for the sender.

Definition 3 (Cluster node). *A server node, s, is defined by the configuration* $s = \langle addr,\ \mathcal{A},\ \mathcal{M},\ \mathcal{I}\rangle$ *where addr is its address,* $\mathcal{A}$ *is a set containing the addresses of all cluster components,* $\mathcal{M}$ *is a multicast set (initially* $\mathcal{M} = \mathcal{A}$*). Finally,* $\mathcal{I}$ *is a FIFO input queue.*

Definition 4 (Cluster infrastructure). *A cluster,* $\mathcal{N}$*, is defined by the configuration* $\mathcal{N} = \langle addr,\ ctr,\ \mathcal{S},\ \mathcal{A},\ \mathcal{I}\rangle$ *where ctr is a counter to generate fresh ids, initially the value of ctr equals* 0*,* $\mathcal{S}$ *is a set containing the addresses of the infrastructure server nodes, and the rest is defined as before.*

Table 3: The Cluster semantics

$$\frac{a \overset{m}{\rightsquigarrow\!\triangleright} a' \qquad m[dest] == addr}{\langle addr, \{a\} \cup \mathcal{A}', \mathcal{I}, \ldots\rangle_n \rightsquigarrow\!\triangleright \langle addr, \{a'\} \cup \mathcal{A}', \mathcal{I} \leftarrowtail m, \ldots\rangle_n}\ \textsc{Qin}$$

$$\frac{s \overset{m}{\rightsquigarrow\!\triangleright} s'}{\langle \{s\} \cup \mathcal{S}',\ m :: \mathcal{I}', \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}',\ \mathcal{I}', \ldots\rangle_n}\ \textsc{Qout}$$

$$\frac{a \overset{\tau}{\rightsquigarrow\!\triangleright} a'}{\langle \{a\} \cup \mathcal{A}', \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \{a'\} \cup \mathcal{A}' \ldots\rangle_n}\ \textsc{a}$$

$$\frac{s \overset{\{\text{`}D\text{'},id,addr',addr,msg\}}{\rightsquigarrow\!\triangleright} s' \qquad a \overset{\{\text{`}D\text{'},id,addr',addr,msg\}}{\rightsquigarrow\!\triangleright} a' \quad a[addr] == addr}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', \ldots\rangle_n}\ \textsc{Dmsg}$$

$$\frac{s \overset{\{\text{`}R\text{'},,\{\},addr\}}{\rightsquigarrow\!\triangleright} s' \qquad a \overset{\{\text{`}R\text{'},ctr,\{\},addr\}}{\rightsquigarrow\!\triangleright} a'}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', ctr, \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', ctr+1, \ldots\rangle_n}\ \textsc{Rmsg}$$

We start by defining the overall infrastructure semantics and then we zoom in and we define the semantics of individual servers. The cluster semantics is reported in Table 3. Rule Qin states that a component sends a message and the cluster adds it to its input queue. Rule Qout states that the cluster evolves when a server gets a message from the input queue of the cluster. Rule "a" states that the cluster evolves when one of its components evolves independently. Rule Dmsg states that the cluster evolves when a server can forward a data message to a component in the cluster. Rule Rmsg states that the cluster evolves when a node sends a reply message to a component. The reply is labeled with the current value of the the cluster counter and after that the counter is incremented.

The semantics of a cluster node is reported in Table 4. Rule In states that a node gets a message and adds it to its input queue. Rule Reply states that if a node gets a request message from its input queue, it sends a reply to the requester by getting and removing its address from the route of the message. Rule dFwd states that if a node gets a data message from its input queue, it forwards the message to all components in the cluster one by one except for the sender ($addr'$). Notice that this rule can be applied many times as long as the multicast set $\mathcal{M}$ contains more than one element, i.e., $|\mathcal{M}| > 1$. Once $\mathcal{M}$ has only one element, rule eFwd is applied to forward the message to the last address in $\mathcal{M}$, resets the multicast set to its initial value, and the message is removed.

3.3 Ring-based Infrastructure.

We consider a set of server nodes, organised in a logical ring and sharing a counter for sequencing messages coming from components. Each node manages a group of components and can have exclusive locks to the ring counter. When a request

Table 4: Cluster node semantics

$$\frac{}{\langle \mathcal{I}, \ldots \rangle_s \overset{m}{\rightsquigarrow\!\triangleright} \langle \mathcal{I} \leftarrow m, \ldots \rangle_s}\ \textsc{In} \qquad \frac{m \overset{?}{=} \{\text{`}Q\text{'}, \{addr'\}, addr\}}{\langle m :: \mathcal{I}', \ldots \rangle_n \overset{\{\text{`}R\text{'},,\{\},addr'\}}{\rightsquigarrow\!\triangleright} \langle \mathcal{I}', \ldots \rangle_n}\ \textsc{Reply}$$

$$\frac{|\mathcal{M}| > 1 \qquad m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr'', msg\} \qquad \mathcal{M}' = \mathcal{M} \backslash addr' \qquad x = \mathcal{M}'.get()}{\langle addr, \mathcal{A}, \mathcal{M},\ m :: \mathcal{I}' \rangle_s \overset{\{\text{`}D\text{'}, id, addr, x, msg\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{A}, \mathcal{M}' \backslash x,\ m :: \mathcal{I}' \rangle_s}\ \textsc{dFwd}$$

$$\frac{|\mathcal{M}| = 1 \qquad m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad x = \mathcal{M}.get()}{\langle addr, \mathcal{A}, \mathcal{M},\ m :: \mathcal{I}' \rangle_s \overset{\{\text{`}D\text{'}, id, addr, x, msg\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{A}, \mathcal{A},\ \mathcal{I}' \rangle_s}\ \textsc{eFwd}$$

message arrives to a node from one of its components, the node acquires a lock on the ring counter, copies it current value, releases it after incrementing it by 1, and finally sends a reply, carrying a fresh id, to the requester. Data messages are directly added to the node's waiting queue; and will be only forwarded to the node's components and to the neighbour node when all previous messages (i.e., with a smaller id) have been received.

Definition 5 (Ring node). *A server node, s, is defined by the configuration* $s = \langle addr,\ nid,\ \mathcal{X},\ \mathcal{D},\ \mathcal{M},\ \mathcal{I}, \mathcal{W} \rangle$ *where* $\mathcal{X}$ *is its neighbour's address,* $\mathcal{D}$ *is a set containing the addresses of components connected to this sever node and also the neighbour's address* $\mathcal{X}$*, and* $\mathcal{M}$ *initially equals* $\mathcal{D}$*. The rest is defined as before.*

Definition 6 (Ring infrastructure). *A ring,* $\mathcal{N}$*, is defined by the configuration* $\mathcal{N} = \langle \mathcal{S},\ \mathcal{A},\ ctr \rangle$*. We have that:*

- $\forall s \in \mathcal{N}[\mathcal{S}] :\ s[\mathcal{X}] \neq \bot \wedge\ s[\mathcal{X}] \in \mathcal{N}[\mathcal{S}]$.
- $\forall s_1, s_2 \in \mathcal{N}[\mathcal{S}] : s_1[\mathcal{X}] = s_2[\mathcal{X}]$ *implies* $s_1 = s_2$.

The semantics rules of a ring infrastructure are reported in Table 5. The rules (S↔S) and (S↔A) state that a ring evolves when a message m is exchanged either between two of its servers (s_1 and s_2) or between a server and a component respectively. The latter rule concerns only request and data messages. Furthermore, the rules (S) and (A) state that a ring evolves when one of its servers or one of its connected components evolves independently. Finally, rule RMSG states that a ring evolves also when a reply is exchanged between a server node and a component, but in this case the counter of the ring is increased.

The semantics rules of a ring node are reported in Tabel 6. Rule IN states that a node receives a message m and adds it to its input queue ($\mathcal{I} \leftarrow m$) if the destination field of m matches its own address $addr$. Rule REPLY states that if a node gets a request message from its input queue $m :: \mathcal{I}$, it sends a reply, to the requester. Rule WIN states that if a node gets a data message from its input queue, it adds the message to its waiting queue $\mathcal{W}$ only if $m[id] \geq nid$ otherwise the message is discarded as stated by rule DISCARD. This is important to avoid

Table 5: Ring infrastructure semantics

$$\frac{s_1 \overset{m}{\rightsquigarrow\!\triangleright} s_1' \qquad s_2 \overset{m}{\rightsquigarrow\!\triangleright} s_2'}{\langle \{s_1, s_2\} \cup \mathcal{S}', \mathcal{A}, \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \{s_1', s_2'\} \cup \mathcal{S}', \mathcal{A}, \ldots\rangle_n}\ \text{S}{\leftrightarrow}\text{S}$$

$$\frac{s \overset{\tau}{\rightsquigarrow\!\triangleright} s'}{\langle \{s\} \cup \mathcal{S}', \mathcal{A}, \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}', \mathcal{A}, \ldots\rangle_n}\ \text{S}$$

$$\frac{s \overset{m}{\rightsquigarrow\!\triangleright} s' \qquad a \overset{m}{\rightsquigarrow\!\triangleright} a' \qquad m \overset{?}{\neq} \{\text{`}R\text{'}, , \{\}, addr\}}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', \ldots\rangle_n}\ \text{S}{\leftrightarrow}\text{A}$$

$$\frac{a \overset{\tau}{\rightsquigarrow\!\triangleright} a'}{\langle \mathcal{S}, \{a\} \cup \mathcal{A}', \ldots\rangle_n \rightsquigarrow\!\triangleright \langle \mathcal{S}, \{a'\} \cup \mathcal{A}', \ldots\rangle_n}\ \text{A}$$

$$\frac{s \overset{\{\text{`}R\text{'},,\{\},addr\}}{\rightsquigarrow\!\triangleright} s' \qquad a \overset{\{\text{`}R\text{'},ctr,\{\},addr\}}{\rightsquigarrow\!\triangleright} a'}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', ctr\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', ctr+1\rangle_n}\ \text{RMSG}$$

duplicates. Furthermore, rule DFWD states that when the id of the message on top of $\mathcal{W}$ matches *nid* (i.e., $m[id] == nid$), the server starts forwarding m to its children one by one except for the sender. Notice that this rule can be applied many times as long as the multicast set $\mathcal{M}$ contains more than one element, i.e., $|\mathcal{M}| > 1$. Once $\mathcal{M}$ has only one element, rule EFWD is applied to forward the message to the last address in $\mathcal{M}$. As a result, *nid* is incremented, m is removed from $\mathcal{W}$, and the multicast set $\mathcal{M}$ is reset to its initial value.

3.4 A Tree-based Infrastructure

We consider a set of servers, organised in a logical tree. A component can be connected to one server (its parent) in the tree and can interact with others in any part of the tree by only dealing with its parent. When a component wants to send a message, it asks for a fresh id from its parent. If the parent is the root of the tree, it replies with a fresh id, otherwise it forwards the message to its own parent in the tree. Only the root of the tree can sequence messages.

Definition 7 (Tree server). *A tree server, s, is defined by the configuration: $s = \langle addr,\ ctr,\ nid,\ \mathcal{D},\ \mathcal{M},\ \mathcal{I},\ \mathcal{W},\ \mathcal{X}\rangle$ where $\mathcal{D}$ is a set containing the addresses of the server's children which include connected components and servers, $\mathcal{M}$ is a multicast set (initially $\mathcal{M} = \mathcal{D}$). The rest are defined as before.*

Definition 8 (Tree infrastructure). *A tree infrastructure, $\mathcal{N}$, is defined by the configuration: $\mathcal{N} = \langle \mathcal{S}, \mathcal{A}\rangle$ where $\mathcal{S}$ denotes the set of servers and $\mathcal{A}$ denotes the set of connected components such that:*

- *$\forall s_1, s_2 \in \mathcal{S}$, we say that s_1 is a direct child of s_2, written $s_1 \prec s_2$, if and only if $s_1[\mathcal{X}] = s_2[addr]$; $\prec^+$ denotes the transitive closure of $\prec$.*

Table 6: Ring Node Semantics

$$\frac{m[dest] == addr}{\langle addr, \mathcal{I}, \ldots\rangle_s \overset{m}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{I} \leftarrow\!\shortmid m, \ldots\rangle_s}\ \textsc{In}$$

$$\frac{m \overset{?}{=} \{\text{`}Q\text{'}, \{addr'\}, addr\}}{\langle addr,\ m :: \mathcal{I}', \ldots\rangle_s \overset{\{\text{`}R\text{'},,\{\},addr'\}}{\rightsquigarrow\!\triangleright} \langle addr,\ \mathcal{I}', \ldots\rangle_s}\ \textsc{Reply}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \quad id \geq nid}{\langle addr,\ nid,\ \mathcal{W},\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\tau}{\rightsquigarrow\!\triangleright} \langle addr,\ nid,\ \mathcal{W} \leftarrow\!\shortmid m,\ \mathcal{I}',\ \ldots\rangle_s}\ \textsc{wIn}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \quad id < nid}{\langle addr,\ nid,\ \mathcal{W},\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\tau}{\rightsquigarrow\!\triangleright} \langle addr,\ nid,\ \mathcal{W},\ \mathcal{I}',\ \ldots\rangle_s}\ \textsc{Discard}$$

$$\frac{|\mathcal{M}| > 1 \qquad \begin{array}{c} m[id] == nid \\ m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \end{array} \qquad \mathcal{M}' = \mathcal{M}\backslash addr' \quad x = \mathcal{M}'.get()}{\langle addr, nid, \mathcal{D}, \mathcal{M},\ m :: \mathcal{W}', \ldots\rangle_s \overset{\{\text{`}D\text{'},id,addr,x,msg\}}{\rightsquigarrow\!\triangleright} \langle addr, nid, \mathcal{D}, \mathcal{M}'\backslash x,\ m :: \mathcal{W}', \ldots\rangle_s}\ \textsc{dFwd}$$

$$\frac{|\mathcal{M}| = 1 \qquad m[id] == nid \qquad m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad x = \mathcal{M}.get()}{\langle addr, nid, \mathcal{D}, \mathcal{M},\ m :: \mathcal{W}', \ldots\rangle_s \overset{\{\text{`}D\text{'},id,addr,x,msg\}}{\rightsquigarrow\!\triangleright} \langle addr,\ nid + 1, \mathcal{D}, \mathcal{D},\ \mathcal{W}', \ldots\rangle_s}\ \textsc{eFwd}$$

- $\forall s \in \mathcal{S}$, *we have that* $s \not\prec^+ s$.
- *The root:* $\exists s \in \mathcal{S}$ *such that for any* $s' \in (\mathcal{S}\backslash\{s\})$, $s' \prec^+ s$ *and we have that:*
  - $s'[nid] \leq s[ctr]$.
  - *For any message* $m \in s'[\mathcal{W}]$ *we have that* $m[id] \leq s[ctr]$.
- *A root is unique: if* $s, s' \in \mathcal{S}$ *and* $s[\mathcal{X}] = s'[\mathcal{X}] = \perp$ *then we have that* $s = s'$.
- $\forall s \in \mathcal{S}$ *and for each message* $m \in s[\mathcal{W}]$, *we have that* $m[id] \geq s[nid]$.

The semantics rules of a tree infrastructure are reported in Table 7. The rules (S↔S) and (S↔A) state that a tree evolves when a message m is exchanged either between two of its servers (s_1 and s_2) or between a server and a component respectively. Furthermore, the rules (S) and (A) state that a tree evolves when one of its servers or one of its connected components evolves independently.

The semantics rules of a tree server are defined by the rules in Table 8. Rule In states that a server receives a message m and adds it to the back of its input queue ($\mathcal{I} \leftarrow\!\shortmid m$) if the destination field of m matches its own address $addr$. Rule Reply states that if a root server gets a request from the front of its input queue $m :: \mathcal{I}'$, it sends a reply to the requester by getting its address from the route of the message $x = route.get()$. The id of the reply is assigned the value of the root's counter ctr. By doing so, the counter is incremented. On the other hand, a non-root server adds its address to the message's route and forwards it to its parent as stated by rule qFwd. Rule rFwd instead is used for forwarding

Table 7: Tree infrastructure semantics

$$\frac{s_1 \overset{m}{\rightsquigarrow\!\triangleright} s_1' \qquad s_2 \overset{m}{\rightsquigarrow\!\triangleright} s_2'}{\langle \{s_1, s_2\} \cup \mathcal{S}', \mathcal{A}\rangle_n \rightsquigarrow\!\triangleright \langle \{s_1', s_2'\} \cup \mathcal{S}', \mathcal{A}\rangle_n}\ \text{S}\leftrightarrow\text{S} \qquad \frac{s \overset{\tau}{\rightsquigarrow\!\triangleright} s'}{\langle \{s\} \cup \mathcal{S}', \mathcal{A}\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}', \mathcal{A}\rangle_n}\ \text{S}$$

$$\frac{s \overset{m}{\rightsquigarrow\!\triangleright} s' \qquad a \overset{m}{\rightsquigarrow\!\triangleright} a'}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}'\rangle_n \rightsquigarrow\!\triangleright \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}'\rangle_n}\ \text{S}\leftrightarrow\text{A} \qquad \frac{a \overset{\tau}{\rightsquigarrow\!\triangleright} a'}{\langle \mathcal{S}, \{a\} \cup \mathcal{A}'\rangle_n \rightsquigarrow\!\triangleright \langle \mathcal{S}, \{a'\} \cup \mathcal{A}'\rangle_n}\ \text{A}$$

reply messages. Rule wIn states that if a server gets a data message from its input queue $\mathcal{I}$ and it is the root or its parent is the source of the message (i.e., $\mathcal{X} == addr' \vee \mathcal{X} == \bot$), the server evolves silently and the message is added to its waiting queue. If the condition ($\mathcal{X} == addr' \vee \mathcal{X} == \bot$) does not hold, the message is also forwarded to the parent as stated by rule wNxt. Furthermore, rule dFwd states that when the id of the message on top of $\mathcal{W}$ matches nid (i.e., $m[id] == nid$), the server starts forwarding m to its children one by one except for the sender. Notice that this rule can be applied many times as long as the multicast set $\mathcal{M}$ contains more than one element, i.e., $|\mathcal{M}| > 1$. Once $\mathcal{M}$ has only one element, rule eFwd is applied to forward the message to the last address in $\mathcal{M}$. As a result, nid is incremented, m is removed from $\mathcal{W}$, and the multicast set $\mathcal{M}$ is reset to its initial value.

Correctness. Since there is a single sequencer in the tree, i.e., the root, two messages can never have the same id. We only need the following propositions to ensure that the tree behaves in agreement with the AbC parallel composition operator. In essence, Proposition 1, ensures that if any component in the tree sends a request for a fresh id, it will get it. Proposition 3, ensures that any two components in the tree with different nid will converge to the same one. However, to prove Proposition 3, we need to prove Lemma 1 and Proposition 2 which guarantee the same results among tree' servers. This implies that messages are delivered to all components. Proposition 4 instead ensures that no message stays in the waiting queue indefinitely. Due to space limitations all proofs are omitted.

Proposition 1. *For any component, with address addr and a parent $\mathcal{X}$, connected to a tree infrastructure $\mathcal{N}$, we have that: if $\langle addr, on, \mathcal{X}, \ldots\rangle_a \overset{\{\text{'Q'},\{addr\},\mathcal{X}\}}{\rightsquigarrow\!\triangleright}$ $\langle addr, 0, \mathcal{X}, \ldots\rangle_a$ then $\mathcal{N} \rightsquigarrow\!\triangleright^* \mathcal{N}'$ and $\langle addr, mid, \ldots\rangle_a \overset{\{\text{'R'},id,\{\},addr\}}{\rightsquigarrow\!\triangleright} \langle addr, id, \ldots\rangle_a$.*

Lemma 1. *For every two tree nodes s_1 and s_2 and a tree-based infrastructure $\mathcal{N}$ such that $s_1, s_2 \in \mathcal{N}[\mathcal{S}]$, we have that:*

- *If $s_1 \prec s_2 \wedge\ s_1[nid] < s_2[nid]$ then $\mathcal{N} \rightsquigarrow\!\triangleright^* \mathcal{N}'$ and $s_1[nid] = s_2[nid]$.*
- *If $s_2 \prec s_1 \wedge\ s_1[nid] < s_2[nid]$ then $\mathcal{N} \rightsquigarrow\!\triangleright^* \mathcal{N}'$ and $s_1[nid] = s_2[nid]$.*

Proposition 2. *Let s_1 and s_2 be two tree nodes and $\mathcal{N}$ be a tree-based infrastructure, $\forall s_1, s_2 \in \mathcal{N}[\mathcal{S}] \wedge\ s_1[nid] < s_2[nid]$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^* \mathcal{N}'$ and $s_1[nid] = s_2[nid]$.*

Table 8: Tree server semantics

$$\frac{m[dest] == addr}{\langle addr, \mathcal{I}, \ldots\rangle_s \overset{m}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{I} \leftarrow\!\shortmid m, \ldots\rangle_s}\ \textsc{In}$$

$$\frac{m \overset{?}{=} \{\text{`}Q\text{'}, route, addr\} \qquad \mathcal{X} == \bot \qquad x = route.get()}{\langle addr, ctr, \mathcal{X},\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\{\text{`}R\text{'}, ctr, route \setminus x, x\}}{\rightsquigarrow\!\triangleright} \langle addr, ctr+1, \mathcal{X}, \mathcal{I}', \ldots\rangle_s}\ \textsc{Reply}$$

$$\frac{m \overset{?}{=} \{\text{`}Q\text{'}, route, addr\} \qquad \mathcal{X} \neq \bot \qquad route' = route.add(addr)}{\langle addr, \mathcal{X},\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\{\text{`}Q\text{'}, route', \mathcal{X}\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{X}, \mathcal{I}', \ldots\rangle_s}\ \textsc{qFwd}$$

$$\frac{m \overset{?}{=} \{\text{`}R\text{'}, id, route, addr\} \qquad x = route.get()}{\langle addr,\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\{\text{`}R\text{'}, id, route \setminus x, x\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{I}', \ldots\rangle_s}\ \textsc{rFwd}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad (\mathcal{X} == addr' \vee \mathcal{X} == \bot)}{\langle addr, \mathcal{X}, \mathcal{W},\ m :: \mathcal{I}', \ldots\rangle_s \overset{\tau}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{X}, \mathcal{W} \leftarrow\!\shortmid m,\ \mathcal{I}', \ldots\rangle_s}\ \textsc{wIn}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad (\mathcal{X} \neq addr' \wedge \mathcal{X} \neq \bot)}{\langle addr, \mathcal{X}, \mathcal{W},\ m :: \mathcal{I}', \ldots\rangle_s \overset{\{\text{`}D\text{'}, id, addr, \mathcal{X}, msg\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{X}, \mathcal{W} \leftarrow\!\shortmid m, \mathcal{I}', \ldots\rangle_s}\ \textsc{wNxt}$$

$$\frac{|\mathcal{M}| > 1 \qquad \begin{array}{c} m[id] == nid \\ m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \end{array} \qquad \mathcal{M}' = \mathcal{M} \setminus addr' \quad x = \mathcal{M}'.get()}{\langle addr, nid, \mathcal{D}, \mathcal{M},\ m :: \mathcal{W}', \ldots\rangle_s \overset{\{\text{`}D\text{'}, id, addr, x, msg\}}{\rightsquigarrow\!\triangleright} \langle addr, nid, \mathcal{D}, \mathcal{M}' \setminus x,\ m :: \mathcal{W}', \ldots\rangle_s}\ \textsc{dFwd}$$

$$\frac{|\mathcal{M}| = 1 \qquad m[id] == nid \qquad m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad x = \mathcal{M}.get()}{\langle addr, nid, \mathcal{D}, \mathcal{M},\ m :: \mathcal{W}', \ldots\rangle_s \overset{\{\text{`}D\text{'}, id, addr, x, msg\}}{\rightsquigarrow\!\triangleright} \langle addr,\ nid+1, \mathcal{D}, \mathcal{D}, \mathcal{W}', \ldots\rangle_s}\ \textsc{eFwd}$$

Proposition 3. *Given any two components a_1 and a_2 in a tree infrastructure $\mathcal{N}$ such that $a_1[nid] < a_2[nid]$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^* \mathcal{N}'$ and $a_1[nid] = a_2[nid]$.*

Proposition 4. *Given a tree infrastructure $\mathcal{N} = \langle \mathcal{S}, \mathcal{A}\rangle$, for any $c \in \mathcal{S} \cup \mathcal{A}$ where $c[\mathcal{W}] = m :: \mathcal{W}'$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^* \mathcal{N}'$ and $c[\mathcal{W}] = \mathcal{W}' +\!\!+ \mathcal{W}''$ where $+\!\!+$ returns a priority queue composed by the sub queues $\mathcal{W}'$ and $\mathcal{W}''$.*

4 Performance Evaluation

We compare the above mentioned infrastructures by modeling them in terms of a Continuous Time Markov Process [17]. The state of a process represents possible infrastructure configurations, while the transitions (that are selected probabilistically) are associated with events on messages. We can consider three types of events: a new message *sent* by a component; a message *transmitted* from

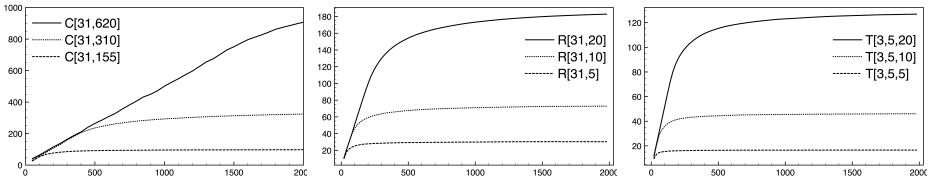

Fig. 1: DP scenario: Avg. Delivery Time for Cluster, Ring, and Tree.

a node to another in the infrastructure; a message locally *handled* by a node (i.e. removed from an input/waiting queue). Each event is associated with a *rate* that is the parameter of the *exponentially distributed* random variable governing the *event duration*. We developed a simulator[5] for performance evaluation.

To perform the simulation we need to fix three parameters: the *component sending rate* λ_s; the *infrastructure transmission rate* λ_t; and the *handling rate* λ_h. In all experiments, we fix the following values: $\lambda_s = 1.0$, $\lambda_t = 15.0$, and $\lambda_h = 1000.0$ and rely on kinetic Monte Carlo simulation [19]. The infrastructure configurations are defined as follows:

- $C[x, y]$, indicates a *cluster* with x nodes and y components;
- $R[x, y]$ indicates a *ring* with x nodes each of which manages y components;
- $T[x, y, z]$ indicates a *tree* with x levels. Each node (but the leafs) has $y + z$ children: y nodes and z components. A leaf node has z components.

We consider two scenarios: (1) *Data Providers* (DP): In this scenario only a fraction of components sends data messages that they, for example, acquire via sensors in the environment where they operate. An example could be a *Traffic Control System* where *data providers* are devices located in the city and *data receivers* are the vehicles traveling in the area; (2) *communication intensive* (CI): This scenario is used to estimate the performance when all components send messages continuously at a fixed rate so that we can evaluate situations of overloaded infrastructures. The former scenario is more realistic for CAS.

We consider two measures: the average delivery time and the average message time gap. The first measure indicates the time needed for a message to reach all components, while the latter indicates the interval between two different messages received by a single component (i.e., an indication of throughput).

Data provider scenario (DP) We consider configurations with 31 server nodes 155, 310, or 620 components and assume that only 10% of the components is sending data. The average delivery time is reported in Fig. 1 while the average message time gap (with confidence intervals) is reported in Fig. 2. The tree structure offers the best performance while the cluster one is the worst. When the cluster reaches an equilibrium (at time ~ 2000), ~ 90 time units are needed to deliver a message to 155 components while the ring and the tree need only ~ 25 and ~ 10 time units, respectively. The reason is that in the cluster all server

[5] The simulator: `https://bitbucket.org/Lazkany/abcsimulator`

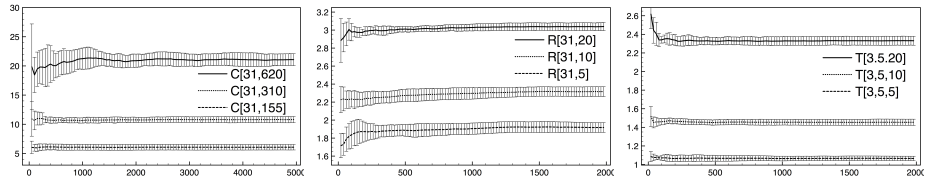

Fig. 2: DP scenario: Avg. Message Time Gap for Cluster, Ring, and Tree.

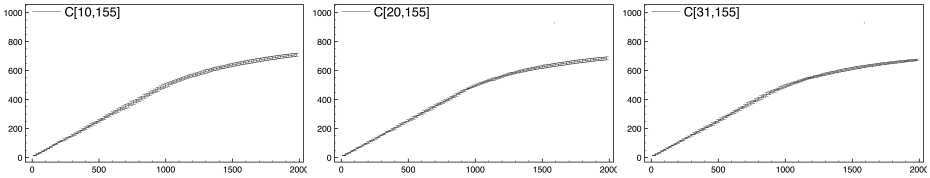

Fig. 3: CI scenario: Avg. Delivery Time for Cluster with 10, 20 and 31 servers.

nodes share the same input queue while in the tree and the ring each server node has its own queue. We can also observe that the performance of the ring in this scenario is close to the one of the tree. Moreover, in the cluster, the performance degrades when the number of components increases. This does not happen for the tree and the ring. Finally, we can observe that messages are delivered more frequently in the ring (~ 1.9 time units) and the tree (~ 1.1 time units) than in the cluster (~ 5.5 time units) as reported in Fig. 2.

Communication intensive scenario (CI) We consider infrastructures composed by 155 components that continuously send messages to all the others. Simulations are performed by considering the following configurations:

- Cluster-based infrastructure: $C[10, 155]$, $C[20, 155]$ and $C[31, 155]$;
- Ring-based infrastructure: $R[5, 31]$ and $R[31, 5]$;
- Tree-based infrastructure: $T[5, 2, 5]$ and $T[3, 5, 5]$.

Fig. 3 shows that the cluster has the worst performance. One can easily notice that when the cluster reaches an equilibrium (~ 2000), ~ 800 time units are needed to deliver a message to all components. We also observe that the number of nodes in the cluster has a minimal impact on this measure because they all

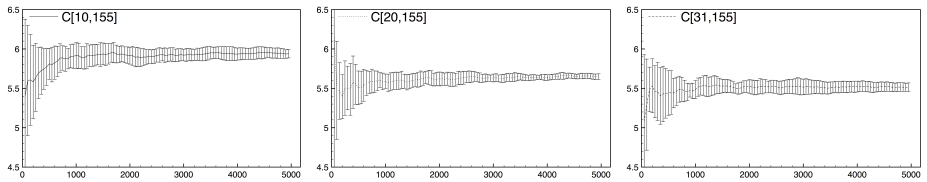

Fig. 4: CI scenario: Avg. Message Time Gap for Cluster with 10, 20 and 31 servers.

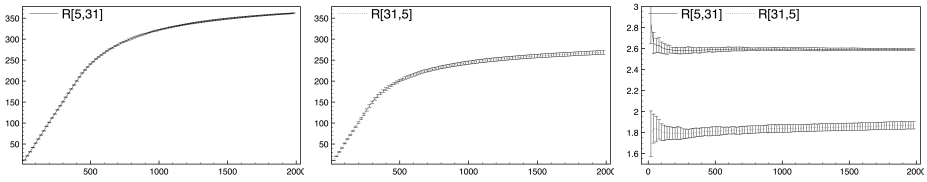

Fig. 5: CI scenario: Avg. Delivery Time and Avg. Message Time Gap for Ring with 5 and 31 servers.

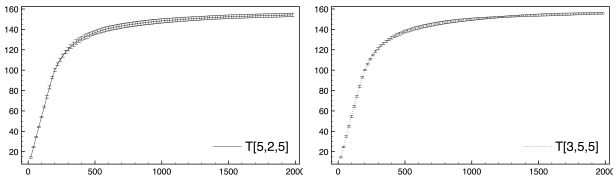

Fig. 6: CI scenario: Avg. Delivery Time: Tree/ $T[5,2,5]$ and $T[3,5,5]$.

share the same input queue. The *Average Message Time Gap*, in Fig. 4, indicates that in the long run a component receives a message every 6/5.5 time units.

Better performance can be obtained if the ring infrastructure is used. In the first two plots of Fig. 5 we report the average delivery time for the configurations $R[5,31]$ and $R[31,5]$. The last plot compares the average message time gap of the two configurations. In the first one, a message is delivered to all the components in 350 time units while in the second one 250 time units are needed. This indicates that increasing the number of nodes in the ring enhances performance. This is because in the ring all nodes cooperate to deliver a given message. Also the time gap decreases, i.e., a message is received every 2.6 and 1.8 time units.

Fig. 6 shows how the *average delivery time* changes during the simulation for $T[5,2,5]$ and $T[3,5,5]$. The two configurations have exactly the same number of nodes (31) with a different arrangement. The two configurations work almost in the same way: a message is delivered to all the components in about 120 time units. Clearly, the tree is 5-time faster than the cluster and 2-time faster than the ring. Moreover, in the tree-based approach, a message is delivered to components every ~ 1.1 time units as reported in Fig. 7. This means that messages in the tree are constantly delivered after an initial delay.

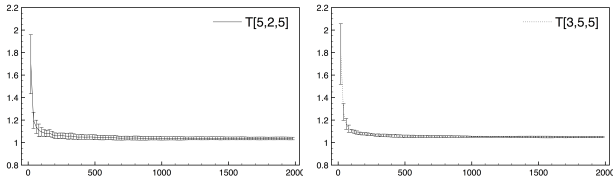

Fig. 7: CI scenario: Avg. Message Time Gap: Tree/ $T[5,2,5]$ and $T[3,5,5]$.

The results show that tree infrastructures offer the best performance; cluster-based ones do not work well while ring-based ones are in between the two.

5 Concluding Remarks, Future Work, and Related Work

The contribution of our paper is twofold: (i) the definition of a distributed tree-based infrastructure for coordinating attribute-based interaction and its actual implementation in Google Go; (ii) the proof of correctness of the proposed infrastructure and its performance evaluation. The results showed that the tree infrastructure has a better performance when compared with others in terms of minimising the average delivery time and maximising throughput.

As for future work, we plan to integrate (possibly) existing techniques for resilience to deal with imperfect communication links, dropped messages, and node's failures. Apart from simulation, we will consider large and realistic case studies to investigate the actual performance of the current Go implementation.

We would like to conclude by relating to existing approaches. For implementations of attribute-based interaction, we refer to the Java-based [2] and the Erlang-based [10] implementations. As we mentioned before, these implementations are centralised while we are aiming for a distributed one.

Many approaches have been proposed to deal with distributed coordination, but they are difficult to compare as they differ in their assumptions, properties, objectives, or target applications [11]. However, they can be classified according to their ordering mechanisms. Below we relate to well-known approaches.

In the fixed sequencer approach [9], a single sequencer maintains the order of message delivery and components communicate by interacting only with the sequencer. The cluster infrastructure is a natural *distributed* extension; and also the tree is a generalisation of this approach where instead of a single sequencer, we consider a propagation tree. The ordering decisions are resolved along tree paths. Actually, a propagation tree with depth 1 is a fixed sequencer.

The moving sequencer approach [7] avoids the bottleneck of a single sequencer by transferring the sequencer role among several nodes. Nodes form a logical ring and circulate a token, carrying a counter and a list of sequenced messages. Once token is received, a sequencer sequences its messages, sends all sequenced messages to its connected components, updates the token, and passes it along with sequenced messages to next node; thus the load is distributed among several nodes. However, the liveness of the algorithm depends on the token and, if the number of senders in one node is larger than others, fairness is hard to achieve. The ring-based infrastructure can be viewed as a generalisation of this technique where fairness is "resolved" by sharing a common counter.

In the privilege-based approach [8], senders circulate a token and each sender has to wait for the token. Upon receipt of token, the sender sequences its messages, sends them to destinations, and passes the token to the next sender. This approach is not suitable for open systems, since it assumes that all senders know each other. Also fairness is hard to achieve, e.g., some components send larger number of messages than others.

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP International Conference, FORTE. pp. 1–18. Springer (2016), http://dx.doi.org/10.1007/978-3-319-39570-8_1
2. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming of CAS systems by relying on attribute-based communication. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA '16, 2016, Proceedings, Part I. pp. 539–553. Springer (2016), https://doi.org/10.1007/978-3-319-47166-2_38
3. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming the Interactions of Collective Adaptive Systems by Relying on Attribute-based Communication. ArXiv e-prints (Oct 2017), http://arxiv.org/abs/1711.06092
4. Abd Alrahman, Y., De Nicola, R., Loreti, M., Tiezzi, F., Vigo, R.: A calculus for attribute-based communication. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. pp. 1840–1845. SAC '15, ACM (2015), http://doi.acm.org/10.1145/2695664.2695668
5. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
6. Albert, E., Lanese, I. (eds.): Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Greece, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9688. Springer (2016), https://doi.org/10.1007/978-3-319-39570-8
7. Chang, J.M., Maxemchuk, N.F.: Reliable broadcast protocols. ACM Trans. Comput. Syst. 2, 251–273 (Aug 1984), http://doi.acm.org/10.1145/989.357400
8. Cristian, F.: Asynchronous atomic broadcast. IBM Technical Disclosure Bulletin 33(9), 115–116 (1991)
9. Cristian, F., Mishra, S.: The pinwheel asynchronous atomic broadcast protocols. In: Autonomous Decentralized Systems, 1995. Proceedings. ISADS 95., Second International Symposium on. pp. 215–221. IEEE (1995), https://doi.org/10.1109/ISADS.1995.398975
10. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: Aerlang at work. In: SOFSEM 2017: 43rd International Conference on Current Trends in Theory and Practice of Computer Science,. Lecture Notes in Computer Science, vol. 10139, pp. 485–497. Springer (2017), https://doi.org/10.1007/978-3-319-51963-0_38
11. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Survey 36, 372–421 (2004), http://doi.acm.org/10.1145/1041680.1041682
12. Ferscha, A.: Collective adaptive systems. In: Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers. pp. 893–895. UbiComp/ISWC'15 Adjunct, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2800835.2809508
13. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985), http://doi.acm.org/10.1145/3149.214121
14. Jensen, T.R., Toft, B.: Graph coloring problems, vol. 39. John Wiley & Sons (1995)
15. Lopes, L., Silva, F., Vasconcelos, V.T.: A virtual machine for a process calculus. In: Nadathur, G. (ed.) Principles and Practice of Declarative Programming. pp. 244–260. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)

16. Prasad, K.: A calculus of broadcasting systems. In: TAPSOFT'91. pp. 338–358. Springer (1991)
17. Robertson, J.B.: Continuous-time markov chains (W. j. anderson). SIAM Review 36(2), 316–317 (1994), https://doi.org/10.1137/1036084
18. Sangiorgi, D., Walker, D.: The pi-calculus: a Theory of Mobile Processes. Cambridge university press (2003)
19. Schulze, T.P.: Efficient kinetic monte carlo simulation. Journal of Computational Physics 227(4), 2455 – 2462 (2008), http://www.sciencedirect.com/science/article/pii/S0021999107004755
20. Vukolić, M.: The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In: Camenisch, J., Kesdoğan, D. (eds.) Open Problems in Network Security. pp. 112–125. Springer International Publishing, Cham (2016)