An Enhanced Exchange Operator for XC

An Enhanced Exchange Operator for XC

Giorgio Audrito[1 [0000−0002−2319−0375]], Daniele Bortoluzzi[1 [0009−0004−2056−9111]], Ferruccio Damiani[1 [0000−0001−8109−1706]], Giordano Scarso[1 [0009−0009−2114−7435]], and Gianluca Torta[1 [0000−0002−4276−7213]]

Università degli Studi di Torino, Via Verdi 8, 10124 Turin, Italy
{giorgio.audrito,daniele.bortoluzzi,ferruccio.damiani,
giordano.scarso,gianluca.torta}@unito.it

Abstract. Recent work in the area of coordination models and collective adaptive systems promotes a view of distributed computations as functions manipulating computational fields (data structures spread over space and evolving over time) and introduces the *eXchange Calculus (XC)* as a novel formal foundation for field computations. In XC, evolution (time) and neighbor interaction (space) are handled by a single communication primitive called *exchange*, working on the *neighbouring value* data structure to represent both received values and values to share. However, the exchange primitive does not allow to directly retain information about neighbours across subsequent rounds of computation. This hampers the convenient expression of useful algorithms in XC, such as the computation of a *neighbour reliability score*. In this paper, we introduce a new generalised version of the exchange primitive, also implementing it into the FCPP DSL. This primitive allows for neighbour data retention across rounds, strictly expanding the expressiveness of the exchange primitive in XC. The contribution is then evaluated through a case study on distributed sensing in a wireless sensor network of battery-powered devices, exploiting the reliability scores to improve robustness.

Keywords: Core calculus · Aggregate computing · C++ DSL.

1 Introduction

The number and density of networked computing devices distributed throughout our environment is continuing to increase rapidly. In order to manage and make effective use of such systems, there is likewise an increasing need for software engineering paradigms that simplify the engineering of resilient distributed systems. Aggregate programming [10,19] is one such promising approach, providing a layered architecture in which programmers can describe computations by combining resilient operations on "aggregate" data structures with values spread over space and evolving in time.

The foundation of this approach is on field-based computations, originally formalized by the field calculus (FC) [8], and later refined by the exchange calculus (XC) [4,5], a terse mathematical model of distributed computation that simultaneously describes both collective system behavior and the independent,

unsynchronized actions of individual devices that will produce that collective behavior. In this approach, computation (executed in asynchronous rounds), communication (which is neighbour-based), and state over time, are all expressed by means of a single communication primitive, called `exchange`.

This primitive applies a given function to a single argument `n` (a view of the values produced by neighbours), obtaining as result a view `s` of the values to send back to neighbours. This mechanism provides a general communication pattern, but does not allow to directly retain information about neighbours across subsequent rounds of computation: the result `s` obtained on a device in a round is not available in that device on the following round. This prevents from conveniently expressing in XC useful algorithms, such as the computation of a *neighbour reliability score* based on statistics on messages received by neighbours. Since XC is Turing-complete, these algorithms could still be expressed, but only by breaking the neighbouring-value abstraction given by the language.

In this paper, we address this limitation by introducing a new generalised version of the exchange primitive that allows for neighbour data retention. The generalised exchange takes two arguments: both the same `n` as in XC, and a view `o` of the values produced for neighbours in the previous round of the same device. This strictly extends XC, retaining all previous expressiveness while opening the way for new possibilities. We illustrate the increase in expressiveness by means of examples, presenting two novel neighbour reliability metrics (uni-connection and mixed-connection), that can be used in the novel *stabilised single-path* collection algorithm. The proposal is then evaluated in a simulated case study on distributed sensing, set in an unreliable wireless sensor network of battery-powered devices. The evaluation is carried out through the FCPP simulator [1,7], that has been extended to support the generalised exchange primitive and a more advanced model of unreliable communication.

Following a background on field-based approaches (focusing on XC) in Section 2, we introduce the generalised exchange construct in Section 3, evaluate the effectiveness of the construct in a case study on reliable WSN sensing in Section 4, and conclude with a summary and discussion of future work in Section 5.

2 Background

In this section, we briefly present the class of field-based coordination approaches, and then dive into the system modeling, syntax and semantics of the *eXchange Calculus (XC)*. Please refer to [4,5] for more details.

2.1 Field-Based Coordination

Both XC [4,5] and its close predecessor FC [8] belong to the class of *field-based coordination* approaches to the design of distributed computing systems. Such approaches take an important natural source of inspiration from the concept of *field* in physics. A coordination field (or co-field) was initially introduced in [14] to facilitate the formation of self-organizing patterns in agent movement within

complex environments. Building on this concept, the TOTA (Tuples On The Air) tuple-based middleware [13] was developed to enable field-based coordination for pervasive computing applications. In TOTA, each tuple inserted into a network node is endowed with content (the tuple data), a diffusion rule (determining how the tuple is to be replicated and spread), and a maintenance rule (dictating how the tuple should evolve over time or in response to events). One of the pioneering works linking field-based coordination with formalization tools, such as process algebras and transition systems, is the $\sigma\tau$-Linda model [20]. In this model, agents can inject *processes* into the space that propagate, gather, and decay tuples, thereby sustaining fields of tuples. In [12], the authors define the *SMuC* language as an extension of μ-calculus that is able to express global programs on fields that can be executed on a distributed system. Surveys reviewing approaches to abstract spatial collective adaptive systems can be found in [9,11,15,16].

2.2 System Model

The systems we aim to program can be conceptualized as sets of *nodes*, each capable of engaging with the environment via *sensors* and *actuators*, as well as communicating with neighboring nodes through message exchanges.

We operate under the assumption that each node executes in *asynchronous* cycles known as sense-compute-act rounds, where:

- **sense**: The node gathers current environmental data by querying sensors and collects recent messages from neighbors. This information constitutes the node's *context*.
- **compute**: The node processes a shared control program, which interprets the context (i.e., inputs from sensors and neighbors) to generate an output detailing the actions to be taken (e.g., actuations and communications).
- **act**: The node carries out the actions in output, potentially resulting in changes to the environment or the delivery of messages to neighbors.

This loop ensures continuous assessment of the context at discrete intervals, with reactions computed and executed continuously and asynchronously. An instance of system execution can be represented through an event structure (see Figure 1). Here, events (denoted by ϵ) encapsulate entire sense-compute-act rounds, and the arrows linking events signify that certain source events have furnished inputs (i.e., messages) to target events. Specifically, if event ϵ' is linked by an arrow to ϵ, we designate ϵ' as a "neighbor" of ϵ, symbolized as $\epsilon' \rightsquigarrow \epsilon$. Therefore, programming the systems described in this section consists of establishing the control rules that dictate how the context at each event is translated into the messages intended for neighboring events.

2.3 Neighbouring Values

In XC, we categorize values into two types. The *Local* values, denoted by ℓ, encompass traditional atomic and structured types like integers, floats, strings, or

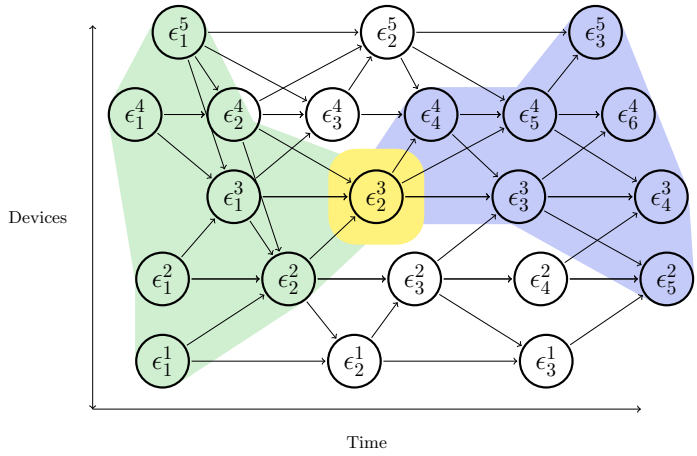

Fig. 1. An event structure modelling a distributed system execution. Node ϵ_k^δ denotes the k-th round of device δ. The yellow area contains a reference event, whose past (green) and future (blue) are identified from causal arrows between neighbour events.

lists. On the other hand, the neighboring values (referred to as nvalues) are mappings $\mathtt{w}$ from device identifiers δ_i to corresponding local values ℓ_i. Additionally, there's an extra local value ℓ acting as a default:

$$\mathtt{w} = \ell[\delta_1 \mapsto \ell_1, \dots, \delta_n \mapsto \ell_n]$$

An nvalue specifies the values received from or sent to neighbors: received values are gathered into nvalues, can then be processed locally, and the resulting nvalue can be interpreted as messages to be sent back to neighbors. Devices associated with an entry in the nvalue are typically a small subset of all devices, namely those close enough to the current device and working correctly.

The default value is used when a value isn't available for some neighbor δ'. For instance, if δ' has just been powered on and hasn't produced a value yet, or if it has just moved close enough to the current device δ to become one of its neighbors. Hence, the notation should be understood as follows: "the nvalue $\mathtt{w}$ is ℓ everywhere (i.e., for all devices) except for devices $\delta_1, \dots, \delta_n$, which have given values $\ell_1, \dots, \ell_n$, respectively."

To illustrate nvalues, consider Figure 1. Upon waking up for computation ϵ_2^3, device δ_3 might process an nvalue $\mathtt{w} = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2, \delta_2 \mapsto 3]$, representing messages carrying scalar values 1, 2, and 3 received when asleep from δ_4, δ_3 (itself at the previous round), and δ_2. Entries for all other (neighbor) devices default to 0. After computation, δ_2 might send out messages represented by nvalue $\mathtt{w}' = 0[\delta_4 \mapsto 5, \delta_3 \mapsto 6]$, sending 5 to δ_4, 6 to δ_3, and 0 to every other (neighbor) device, such as a newly-connected device. For convenience, $\mathtt{w}(\delta')$ denotes the local value (specific or default) associated with δ' by $\mathtt{w}$.

Note that a local value ℓ can be naturally converted to an nvalue $\ell[]$, that holds the default value for every device. On the other hand, functions on local

values are implicitly lifted to nvalues by applying them pointwise to the maps' content. For example, if $\mathtt{w}_1$ assigns value 2 to δ_3 and $\mathtt{w}_2$ assigns default value 1 to δ_3, then $\mathtt{w}_3 = \mathtt{w}_1 \cdot \mathtt{w}_2$ assigns value $2 \cdot 1 = 2$ to δ_3. Local values and nvalues can thus be treated uniformly by the exchange calculus.

A fundamental operation on nvalues is provided by the built-in function $\mathtt{nfold}(f : (A, B) \to A, \mathtt{w} : B, \ell : A) : A$. This function folds over an nvalue $\mathtt{w}$, starting from a base local value ℓ, repeatedly applying function f to neighbors' values in $\mathtt{w}$, excluding the value for the current device. For instance, if δ_2 with a set of neighbors $\{\delta_1, \delta_3, \delta_4\}$ performs a `nfold` operation $\mathtt{nfold}(*, \mathtt{w}, 1)$, the output will be $1 \cdot \mathtt{w}(\delta_1) \cdot \mathtt{w}(\delta_3) \cdot \mathtt{w}(\delta_4)$. We usually assume that f is associative and commutative since nvalues are unordered maps. Other common built-in operator are:

- $\mathtt{self}(\mathtt{w} : A) : A$, that returns the local value $\mathtt{w}(\delta)$ in $\mathtt{w}$ for the self device δ.
- $\mathtt{modOther}(\mathtt{w} : A, \ell : A) : A$, that returns an nvalue where the default of $\mathtt{w}$ is changed to ℓ, and all other values are the same as in $\mathtt{w}$.
- $\mathtt{mux}(\ell_1 : \mathtt{bool}, \ell_2 : A, \ell_3 : A)$, that returns ℓ_2 if ℓ_1 is `True`, ℓ_3 otherwise (note that all arguments are evaluated).

Moreover, XC features a single communication primitive:

$$\mathtt{exchange}(\mathtt{e}_i, (\mathtt{n}) \texttt{=>} (\mathtt{e}_r, \mathtt{e}_s))$$

evaluated as follows:

- The device computes the local value ℓ_i of $\mathtt{e}_i$ (the *initial* value).
- To evaluate the function provided as the second argument, it substitutes variable `n` with the nvalue `w` of messages received from neighbors for this exchange, using ℓ_i as default.
- The expression returns the value $\mathtt{w}_r$ from the evaluation of $\mathtt{e}_r$.
- The second expression $\mathtt{e}_s$ in the argument function's output evaluates to an nvalue $\mathtt{w}_s$ consisting of local values to be sent to neighbor devices δ', which will use their corresponding $\mathtt{w}_s(\delta')$ upon waking up and performing their next execution round.

In order to clarify the purpose of the two elements of the pair returned by an exchange, in the remainder of this paper we use the notation `return` $\mathtt{e}_r$ `send` $\mathtt{e}_s$ as syntactic sugar for the pair $(\mathtt{e}_r, \mathtt{e}_s)$. When $\mathtt{e}_r$ and $\mathtt{e}_s$ coincide, we also use `retsend` $\mathtt{e}$ as syntactic sugar for the pair $(\mathtt{e}, \mathtt{e})$. The *exchange* construct abstracts the general concept of message exchange and is expressive enough to allow common communication patterns to be expressed through it. If a program executes multiple exchange-expressions, XC ensures through *alignment* that the messages are dispatched across rounds to corresponding exchange-expressions (in the same position in the program AST, and with the same stack frames of function calls).

Syntax:

$\texttt{e} ::= \texttt{x} \mid \texttt{fun}\ \texttt{x}(\overline{\texttt{x}})\{\texttt{e}\} \mid \texttt{e}(\overline{\texttt{e}}) \mid \texttt{val}\ \texttt{x} = \texttt{e}; \texttt{e} \mid \ell \mid \texttt{w}$	expression
$\texttt{w} ::= \ell[\overline{\delta} \mapsto \overline{\ell}]$	nvalue
$\ell ::= \texttt{b} \mid \texttt{fun}\ \texttt{x}(\overline{\texttt{x}})\{\texttt{e}\} \mid \texttt{c}(\overline{\ell})$	local literal
$\texttt{b} ::= \texttt{exchange} \mid \texttt{nfold} \mid \texttt{self} \mid \texttt{modOther} \mid \texttt{uid} \mid \texttt{mux} \mid \ldots$	built-in function

Free variables of an expression:

$\mathsf{FV}(\texttt{x}) = \{\texttt{x}\} \quad \mathsf{FV}(\ell) = \mathsf{FV}(\texttt{w}) = \emptyset \quad \mathsf{FV}(\texttt{fun}\ \texttt{x}_0(\texttt{x}_1, \ldots, \texttt{x}_n)\{\texttt{e}\}) = \mathsf{FV}(\texttt{e}) \setminus \{\texttt{x}_0, \ldots, \texttt{x}_n\}$

$\mathsf{FV}(\texttt{e}_0(\texttt{e}_1, \_, \texttt{e}_n)) = \bigcup_{i=0\_n} \mathsf{FV}(\texttt{e}_i) \quad \mathsf{FV}(\texttt{val}\ \texttt{x} = \texttt{e}; \texttt{e}') = \mathsf{FV}(\texttt{e}) \cup \mathsf{FV}(\texttt{e}') \setminus \{\texttt{x}\}$

Syntactic sugar:

$(\overline{\texttt{x}})\ \texttt{=>}\ \texttt{e}$	$::=$	$\texttt{fun}\ \texttt{y}(\overline{\texttt{x}})\{\texttt{e}\}$ *where* $\texttt{y}$ *is a fresh variable*
$\texttt{def}\ \texttt{x}(\overline{\texttt{x}})\{\texttt{e}\}$	$::=$	$\texttt{val}\ \texttt{x} = \texttt{fun}\ \texttt{x}(\overline{\texttt{x}})\{\texttt{e}\};$
$\texttt{if}(\texttt{e})\{\texttt{e}_\top\}\ \texttt{else}\ \{\texttt{e}_\bot\}$	$::=$	$\texttt{mux}(\texttt{e}, ()\ \texttt{=>}\ \texttt{e}_\top, ()\ \texttt{=>}\ \texttt{e}_\bot)()$

Fig. 2. Syntax (top), free variables (middle) and syntactic sugar (bottom) in XC

2.4 Syntax and Semantics

Figure 2 (top) illustrates the syntax of XC. The overbar notation denotes a (possibly empty) sequence of elements, where $\overline{x}$ represents $x_1, \ldots, x_n$ ($n \geq 0$). The syntax adheres to that of a standard functional language, without any distinct features for distribution, which become evident in the semantics.

An XC *expression* $\texttt{e}$ can take various forms:

- a *variable* $\texttt{x}$;
- a (possibly recursive) *function* $\texttt{fun}\ \texttt{x}(\overline{\texttt{x}})\{\texttt{e}\}$, which may contain free variables;
- a *function call* $\texttt{e}(\overline{\texttt{e}})$;
- a *let-style* expression $\texttt{val}\ \texttt{x} = \texttt{e}; \texttt{e}$;
- a *local literal* ℓ, such as a built-in function $\texttt{b}$, a defined function $\texttt{fun}\ \texttt{x}(\overline{\texttt{x}})\{\texttt{e}\}$ *without* free variables, or a data constructor $\texttt{c}$ applied to local literals (possibly none);
- an *nvalue* $\texttt{w}$, as explained in Section 2.3.

XC can be typed with higher-order let-polymorphism, without differentiation between types for local and neighboring values. This is reflected in the semantics by having language constructs and built-in functions that accept nvalues as arguments, and by implicitly promoting local values ℓ to nvalues $\ell[]$. Free variables are defined conventionally (Figure 2, middle), and an expression $\texttt{e}$ is considered *closed* if $\mathsf{FV}(\texttt{e}) = \emptyset$. Programs, which are closed expressions, do not contain nvalues as sub-expressions, in order to not explicitly mention devices in them: nvalues only emerge during computations and are the sole values produced by evaluating programs. The basic syntax presented is then expanded through syntactic sugar to incorporate infix operators, omitted parentheses in 0-ary constructors, and other non-trivial encodings described in Figure 2 (bottom).

The semantics of XC comprises *(i)* a big-step operational *device semantics*, delineating the computation of a device within a single round; and *(ii)*

a denotational *network semantics*, formalizing communication between different device rounds. The device semantics is expressed through the judgement $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \mathtt{w}; \theta$, interpreted as "expression $\mathtt{e}$ evaluates to nvalue $\mathtt{w}$ and value-tree θ on device δ with respect to sensor values σ and value-tree environment Θ". In this context:

- $\mathtt{w}$ is termed the *result* of $\mathtt{e}$;
- values in σ may be accessed by built-in functions (sensors) called in $\mathtt{e}$;
- θ is an ordered tree with nvalues on certain nodes, representing messages to be sent to neighbors by tracking the nvalues produced by exchange-expressions in $\mathtt{e}$, and the stack frames of function calls; $v\langle\overline{\theta}\rangle$ denotes a value tree with root v and subtrees $\overline{\theta}$;
- Θ accumulates the (non-expired) value-trees received by the most recent rounds of δ's neighbors, stored as a map $\delta_1 \mapsto \theta_1$, ..., $\delta_n \mapsto \theta_n$ ($n \geq 0$) from device identifiers to value-trees. Notation $\pi_1(\Theta)$ denotes the mapping of each device δ_i to the leftmost subtree of θ_i.

The big-step rules of the device semantics define the behaviour of this judgement in mostly standard terms, although taking care of correctly passing around the context Θ in order to ensure alignment, and referring to an auxiliary predicate $\Downarrow^*$ for the evaluation of built-in primitives. Of particular interest for the present work is the semantics of the *exchange* primitive, which is expressed by the following rule, capturing the behavior described in Section 2.3:

$$\text{[A-XC]}\ \frac{\begin{array}{c}\Theta = \overline{\delta} \mapsto \overline{\mathtt{w}}\langle\ldots\rangle \quad \mathtt{w}^{\mathrm{nbr}} = \mathtt{w}^{\mathrm{init}}[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)] \\ \delta; \sigma; \pi_1(\Theta) \vdash \mathtt{w}^{\mathrm{fun}}(\mathtt{w}^{\mathrm{nbr}}) \Downarrow (\mathtt{w}^{\mathrm{ret}}, \mathtt{w}^{\mathrm{send}}); \theta\end{array}}{\delta; \sigma; \Theta \vdash \mathtt{exchange}(\mathtt{w}^{\mathrm{init}}, \mathtt{w}^{\mathrm{fun}}) \Downarrow^* \mathtt{w}^{\mathrm{ret}}; \mathtt{w}^{\mathrm{send}}\langle\theta\rangle}$$

Let δ be the device where the *exchange* operator is executed, and $\overline{\mathtt{w}}$ be the sequence of nvalues received from neighbours $\overline{\delta}$, stored in the environment Θ. The notation $\mathtt{w}^{\mathrm{init}}[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)]$ represents the nvalue $\mathtt{w}^{\mathrm{init}}$ after replacing the value $\mathtt{w}^{\mathrm{init}}(\delta_k)$ for each neighbour device $\delta_k \in \overline{\delta}$ with the message $\mathtt{w}_k(\delta)$ that δ_k sent to the local device δ. The resulting value $\mathtt{w}^{\mathrm{nbr}}$ serves as an argument to function $\mathtt{w}^{\mathrm{fun}}$: the first element of the resulting pair is used as the overall expression result, while the second is used to label the root of the resulting value-tree (and is therefore exchanged with neighbours).

The denotational network semantics employs the device semantics to define the result of evaluating a program across an entire event structure. For detailed, formal descriptions of the XC semantics, please refer to [4,5].

3 Enhanced Exchange Operator

In this paper, we extend the exchange operator by allowing the use of a function $\mathtt{w}^{\mathrm{fun}}$ that accepts two arguments: $\mathtt{w}^{\mathrm{old}}$ and $\mathtt{w}^{\mathrm{nbr}}$ (the changes w.r.t. the semantics of the previous exchange operator are colored in red).

$$\text{[A-XC']}\ \frac{\Theta = \overline{\delta} \mapsto \overline{\mathtt{w}}\langle ... \rangle \quad \mathtt{w}^{\mathrm{nbr}} = \mathtt{w}^{\mathrm{init}}[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)] \quad \mathtt{w}^{\mathrm{old}} = \begin{cases} \mathtt{w}_k\,|_{\overline{\delta}} & \text{if } \delta = \delta_k \text{ for a } k \\ \mathtt{w}^{\mathrm{init}} & \delta \text{ not in } \overline{\delta} \end{cases} \quad \delta; \sigma; \pi_1(\Theta) \vdash \mathtt{w}^{\mathrm{fun}}(\mathtt{w}^{\mathrm{old}}, \mathtt{w}^{\mathrm{nbr}}) \Downarrow (\mathtt{w}^{\mathrm{ret}}, \mathtt{w}^{\mathrm{send}}); \theta}{\delta; \sigma; \Theta \vdash \mathtt{exchange}(\mathtt{w}^{\mathrm{init}}, \mathtt{w}^{\mathrm{fun}}) \Downarrow^* \mathtt{w}^{\mathrm{ret}}; \mathtt{w}^{\mathrm{send}}\langle\theta\rangle}$$

We define $\mathtt{w}^{\mathrm{nbr}}$ as for the original *exchange* operator. If there is no value for the local device δ in the environment Θ (and thus, it is the first round that the current device is computing this exchange expression), we define $\mathtt{w}^{\mathrm{old}}$ to equal the initialization value $\mathtt{w}^{\mathrm{init}}$. If instead δ appears in the sequence of neighbours (and thus $\delta = \delta_k$ for some k), $\mathtt{w}^{\mathrm{old}}$ is set to the nvalue $\mathtt{w}_k$ (extracted from $\overline{\mathtt{w}}$ in Θ) computed in the previous round by the local device as $\mathtt{w}^{\mathrm{send}}$, with the domain restricted to the current neighbors $\overline{\delta}$. This is written with notation $\mathtt{w}\,|_D$, that corresponds to setting the value in $\mathtt{w}$ for any device δ' not in D to the default value of $\mathtt{w}$. Note that the extended *exchange* operator can exploit only the values sent by δ in the *previous* round. However, multi-step memory can be implemented in terms of single-step memory, e.g., by sending a history of previously sent values as an nvalue associating lists of values to devices. Note that an essentially analogous single-step approach is adopted for preserving state in other field-based languages, such as in the distributed implementation of the SMuC calculus [12], and in the FC *rep* operator [8]. In that case, multi-step memory can be built on top of *rep*, although for local values only.

In the following examples, we exploit the extended *exchange* operator to implement different connection counters between the current device and its neighbours. Such counters can be viewed as quantitative ranks of the connection reliability between nodes, an important piece of information that can be exploited in several contexts. In particular, in Section 4 we will present a single-path collection algorithm that uses such ranks to determine the best node to select as parent of the current node in order to find a path toward the source node.

Example 1 (Unidirectional Connection Counter). The following *uniconn-count* function produces an nvalue of inbound connection counters, associating every neighbour to the number of times a message has been received from it.

```
def uniconn-count() {
  exchange( 0, (o,n) => retsend o + modOther(1,0) )
}
```

We set the initial value of the exchange to 0. The update function increases the nvalue `o` by one unit for all neighbours, without changing its default value. This is achieved through the *modOther* function, which applied on 1 and 0 creates an nvalue with a default value of 0 and mapping all current neighbours to 1. Thus, this function counts the number of consecutive rounds in which neighbours remain neighbours, resetting to 0 each time a former neighbour stops being a neighbour. Since the neighbours are exactly those devices that successfully sent a

message to the current device in recent times, this function measures the inbound connection's quality between the current device δ and its neighbours. On the other hand, it does not depend on the outbound connection's quality: even if the current device never sends messages successfully, the counters still increase. Notice that *uniconn-count* is based on argument `o`, and in fact it does not use argument `n`. Indeed, such function could not be expressed (without breaking the nvalue abstraction) with the classic one-argument exchange operator.

Example 2 (Bidirectional Connection Counter). The following *biconn-count* function produces an nvalue of bidirectional connection counters, associating every neighbour to the number of times a message has bounced back and forth between it and the current device.

```
def biconn-count() {
  exchange( 0, (o,n) => retsend n + modOther(1,0) )
}
```

This function is identical to *uniconn-count*, except that the update function passed to exchange uses argument `n` instead of `o`. Thus, in order for the counter to produce a value k larger than 1 for a current neighbour, such neighbour need to have sent a counter value of $k-1$ to the local device. This can only happen if the neighbour had received a value of $k-2$ from the local device, and so on.

In fact, if the local device δ is not able to send to a neighbour δ' but only to receive from it, the counter for δ' will never get larger than 1. Thus, this function allows to measure the combined inbound and outbound connection quality between the current device and its neighbours.

Notice that this function ignores the o parameter of the function passed to *exchange*, and thus essentially uses the original *exchange* described in Section 2.4. Indeed, this function was first introduced as an example of XC capabilities [4,5].

Example 3 (Mixed Connection Counter). Finally, the *mixedconn-count* function exploits the extended *exchange* operator with different *return* and *send* expressions, to enable a behavior which combines those of the two preceding examples, allowing to measure an outbound connection's counter.

```
1 def mixedconn-count() {
2   exchange( 0, (o,n) =>
3         return n
4         send mux(o == 0, n/2, o) + modOther(1, 0)
5   )
6 }
```

The value of the *send* expression (line 4) is the one that is sent to neighbours (contributing in forming their value of `n` in the following round), and will become parameter `o` in the next round of the same device. For each neighbour that has not a zero connection counter in `o` (i.e., is not a new neighbour), the *send* expression increases its connection counter by one, as in *uniconn-count*. Thus, this expression mostly computes a connection counter of received messages.

However, differently from *uniconn-count*, a new neighbour does not necessarily start from zero, but it starts instead from `n/2`, that is, half of the connection counter that the neighbour device has for the current device. In this way, occasionally losing a connection drops the counter without setting it drastically to zero: thus, the expression computes a counter of received messages "smoothed out" with a bidirectional connection pattern.

As a final difference from *uniconn-count*, the `n` value received from neighbours is the one returned by the function as the computed value (line 3). Since `o` is a smoothed counter of received messages, it follows that `n` will contain the smoothed counters of received messages that the neighbours have with me: in other words, it represents counters of sent messages. This allows to measure the outbound connection's quality between the current device and its neighbours in a more reliable way than both *uniconn-count* and *biconn-count*. As we will show in Section 4, this is the best computed information to use as rating for scenarios of data collection where information flows in just one direction from the peripheral nodes toward the source node.

4 Case Study: Reliable WSN Sensing

In order to illustrate the potential applications of the newly introduced *exchange* operator, we simulated a network of battery-powered IoT sensors recharging using solar panels, with varying battery charge level that influences their communication power. In such a network, we assumed the presence of a *gateway* device (the node with *uid* equal to 0) that is connected to the power supply, and thus is always at maximum battery charge and has the best connection parameters.

We assume that the network has to perform a distributed sensing task, where data perceived by individual sensors is aggregated towards the gateway. For simulation purposes, we consider the test scenario where the data to be collected is the sum of a sensed value of 1 for each device (that is, the task is counting the number of active nodes in the network). Even though this task may be too abstract as a distributed sensing application, from a coordination perspective it has the same complexity as any other data collection task, and thus it provides a useful test bed that is easy to evaluate and measure. The implementation of this simulated case study is publicly available online.[1]

4.1 The Aggregate Program

We implement the case study through the novel *stabilized* single-path collection strategy, shown as function `ssp_collection` in Figure 3, with arguments:

- `dist`: estimated distance of the current device from the collection source;
- `value`: the value to be aggregated;
- `null`: the neutral element of the aggregation;
- `accum`: the aggregation function;

[1] https://github.com/fcpp-experiments/oldnbr-evaluation

```
1  // type: (num) -> num
2  def nbr(v) {
3   exchange( v, (o,n) => return n send v )
4  }
5
6  // type: (num, num, num, (num, num) -> num, num, num) -> num
7  def ssp_collection(dist, value, null, accum, rating, stale_factor) {
8   fst(exchange((null, 0.0, uid()), (o,n) =>
9    val result = nfold(accum, mux(trd(n) == uid(), fst(n), null), value);
10
11   val best_neigh = nfold( min, (nbr(dist), -rating, nbr(uid())) );
12   val best_neigh_rating = -snd(best_neigh);
13   val best_neigh_uid = trd(best_neigh);
14
15   val parent_rating = snd(o) * stale_factor;
16   val parent = trd(o);
17
18   if (parent_rating > best_neigh_rating) {
19     retsend (result, parent_rating, parent)
20   } else {
21     retsend (result, best_neigh_rating, best_neigh_uid)
22   }))
23 }
```

Fig. 3. Stabilized Single-Path Collection in XC.

- `rating`: an nvalue ranking neighbours by reliability;
- `stale_factor`: factor resisting to changes of the aggregation path.

This function uses the common XC routine `nbr(v)`, which sends `v` to neighbours and returns the nvalue `n` composed of the values received by neighbours (that is, their value for `v`), implemented in lines 2–4. It also uses the common functions `fst`, `snd`, and `trd` which return the first, second and third element of a tuple respectively; and function `uid` which returns the local device identifier.

The function consists of a single exchange operation, evolving a triple that consists of:

- the partial aggregate computed in the current node;
- the current rating of the parent chosen;
- the id of the *parent* chosen.

Initially, the partial aggregate is equal to `null`, and the parent is the local device with zero rating. As in classic single-path collection [18], the aggregation happens following the chosen *parents* of each node. This is realised in line 9, where the partial aggregation result is computed by folding the partial aggregation results of neighbouring nodes that selected the current device as their parent, via the given accumulate function. Whenever parents are chosen closer to the collection source than the current node, the aggregation is guaranteed to converge

```
val dist = abf_hops(uid() == 0);
val sum = (x, y) => x + y;
val uni-res = ssp_collection(dist, 1.0, 0, sum, uniconn-count(),   0.7);
val bi-res  = ssp_collection(dist, 1.0, 0, sum, biconn-count(),    0.7);
val mix-res = ssp_collection(dist, 1.0, 0, sum, mixedconn-count(), 0.7);
```

Fig. 4. `ssp_collection` calls using different connection counters as rating.

each value towards the source, where the aggregation result will be equal to the aggregation of each value in the network (except for values that have been lost during communication).

What distinguishes this algorithm from classic single-path collection is the strategy for choosing the parent. First, the best candidate among neighbours is computed in line 11-13, as the one with minimal distance (to ensure that we are getting closer to the source), distinguishing nodes with the same distance by choosing the one with maximal rating. We save the identifier of the best candidate in *best_neigh_uid* and its rating in *best_neigh_rating*.

Such candidate is then compared with the previous parent, extracted from `o` in lines 15-16, while reducing its rating by the `stale_factor`. If the reduced rating of the previous parent exceeds that of the best candidate among neighbours (line 18), the previous parent is retained as parent for the current round (line 19), otherwise the best neighbour becomes the new parent (line 21).

The usage of `ssp_collection` in the case study is shown in Figure 4. We calculate argument `dist` using the adaptive Bellman-Ford algorithm implemented in XC,[2] measuring distance in hops. The `value` parameter is set to 1.0 (i.e., each device contributes 1.0 to the collected value), while the `null` value is 0 (the neutral element of the sum). The `accum` parameter is set to a function summing its arguments. The `rating` parameter is an nvalue computed with one of the three connection counters defined in Section 3. Finally, we set the value of `stale_factor` by observing that values close to 0.0 result in a system with faster response to perturbation but increased instability, while values close to 1.0 led to a more stable system with slower reaction to changes. We thus empirically set the `stale_factor` to 0.7 to balance sensitivity to changes and stability.

4.2 Simulation Settings

We tested the proposed SSP collection algorithm in a simulated network of stationary nodes through the FCPP simulator [1,7] and its new feature to emulate unreliable connectivity. Table 1 shows that there are three possible battery level-profiles for battery-powered devices, which are *HIGH*, *MEDIUM*, and *LOW*, which in turn determine the values of three node parameters influencing its connectivity, namely *sleep_ratio*, *send_power_ratio*, and *recv_power_ratio*.

[2] The Bellman-Ford algorithm lends itself very well to distributed implementations in Field Calculus and XC [6].

PROFILE	sleep_ratio	send_power_ratio	recv_power_ratio
SOURCE	0.00	1.00	1.00
HIGH	0.00	0.90	1.00
MEDIUM	0.00	0.75	0.99
LOW	0.10	0.25	0.75

Table 1. Battery profiles and their parameters

The gateway has a special *SOURCE* profile which grants higher send power, modeling the advantage of being connected to the power supply.

The results in this paper are obtained by running 1000 simulations using different random generation seeds. Each simulation lasted 250 seconds and comprised 100 devices randomly spread in a $150m \times 150m$ square area. Each device performed asynchronous rounds every second on average, with a variance of 10%.

The maximum communication range between two nodes varies depending on their battery profile, and is calculated as:

$$\text{range}(\delta_s, \delta_r) = \textit{comm\_range} \cdot \text{send\_power\_ratio}(\delta_s) \cdot \text{recv\_power\_ratio}(\delta_r)$$

where the reference *comm_range* is set to $50m$, δ_s is the sender device and δ_r is the receiving device. Note that depending on the battery profile of δ_s and δ_r, $\text{range}(\delta_s, \delta_r)$ may be different from $\text{range}(\delta_r, \delta_s)$.

In the simulation, 100% of the messages from δ_s to δ_r are lost if their distance is past $\text{range}(\delta_s, \delta_r)$, and 0% of the messages are lost if their distance is zero. In order to accurately model unreliability of communication, the probability of failure at intermediate distances is calculated according to a continuous step-like function inferred from real-world measurements [17], that reaches 50% of communication failure at a distance of $0.7 \cdot \text{range}(\delta_s, \delta_r)$. Parameter *sleep_ratio* models a further message failure probability, due to the device being in a sleeping state during the time window when the message should be received. Note that *sleep_ratio* equal to zero does not entail that the device is never sleeping: in real-world networks, it is usually achieved by negotiating listening time-windows with neighbours, sleeping only outside of those. To better emulate battery preserving policies, we consider that in the *LOW* state a device might miss some of this time-windows allowing an increase in the sleeping time.

In order to model varying lighting conditions on the solar panels, all nodes have a 1% probability to improve or to worsen their battery profile in every round. A connection with a neighbour is dropped if the current device is unable to receive a message from it for 5 seconds.

4.3 Experimental Results

Figure 5 shows the screenshot of a running simulation, with devices as nodes and existing connection as edges of a graph in the simulation area. The color of each

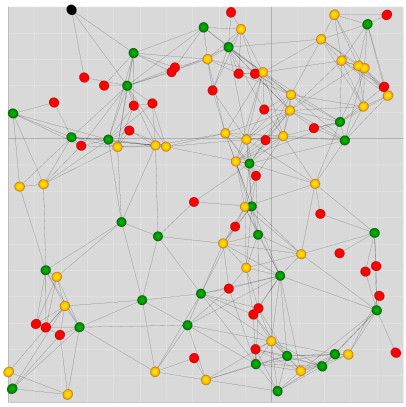

Fig. 5. Screenshot of a simulation of the case study in FCPP.

node represents its current battery profile: red for LOW, yellow for MEDIUM, green for HIGH, and black for the gateway. Figure 6 shows the average (over 1000 simulations) of the number of nodes in the network the algorithm was able to count over time using different functions to compute the reliability rank.

We include a line with a count of the *working* nodes, i.e., with high or medium battery as reference. As expected the computed value is lower than the total number of nodes, due to communication failures, but it's closer to the number of working nodes. The classic (not stabilised) single-path collection strategy, in these scenarios with unreliable connectivity, performs poorly compared to SSP collection. This is due to always selecting as parent the node in the neighbourhood closest to the source with the lowest uid, which may not grant a high-quality connection and may change often. All three versions of SSP collection improve over it, as they make a more informed selection using rating information, and change parents less often thanks to the retention mechanism guided by the *stale factor*. Among the possible ratings, the *uniconn* rating performs the worst, as it computes its connection-counter from received messages only. That may cause the selection as parent of a node to which the current device is not able to reliably send messages, if connection links are sufficiently asymmetrical. The *biconn* rating improves over *uniconn* by deriving its connection counter from a two-way message exchange, which ensure that both the sender and the receiver of the communication are able to communicate.

The *mixedconn* rating further improves over *biconn*, by measuring the number of messages successfully sent to neighbours without the additional requirement to reliably receive data back. It also handles temporary disconnection of a node more gracefully, by retaining the rating decreased by a penalty, for nodes that exited the neighbourhood in the current round rather than resetting their rating back to the default value. This allows to further stabilise the parent selection, improving the collection performance.

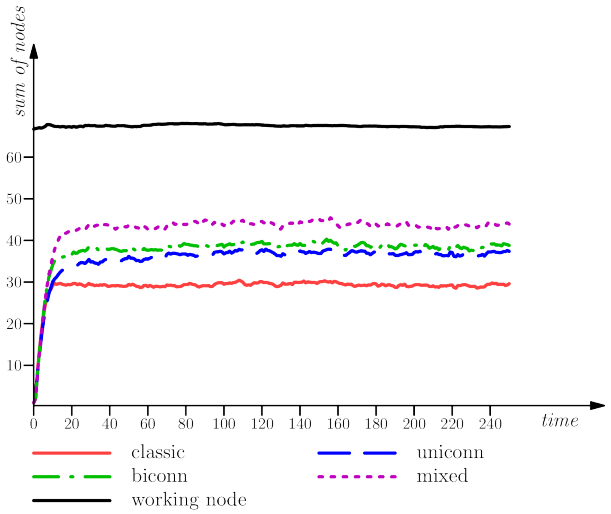

Fig. 6. Average number of nodes in the network counted by the source node using different connection counter functions over 1000 simulations. Time is in seconds.

5 Conclusions and Outlook

In this paper, we introduced a new generalised version of the exchange primitive in XC [4,5], also implementing it into the FCPP DSL [1,7]. This primitive allows for neighbour data retention across rounds, strictly expanding the expressiveness of the exchange primitive in XC. We illustrate the increased expressiveness by means of examples, introducing new connection metrics. We then evaluated the contribution by simulation of a case study on distributed sensing in a wireless sensor network of battery-powered devices. In the future, we plan to further investigate on which neighbour reliability score function can work best in different scenarios. Further, we will try to apply the reliability scoring technique, and in general the possibilities opened by the enhanced exchange primitive, to more advanced algorithms that have been shown to be more effective than classical single-path collection, such as the LIST and BLIST algorithms [2,3]. Finally, further investigations could be performed on the expressiveness of the enhanced exchange primitive possibly expanding the self-stabilising patterns in [18].

Acknowledgments. This paper is part of the project NODES which has received funding from the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036. It has also been carried out within the Agritech National Research Center, funded by the European Union Next-GenerationEU of PNRR, MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1032 17/06/2022, CN00000022. The work was also partially supported by the Italian PRIN project "CommonWears" (2020HCWWLP).

Data Availability Statement The artefact is available at https://zenodo.org/doi/10.5281/zenodo.10797596.

References

1. Audrito, G.: FCPP: an efficient and extensible field calculus framework. In: Proceedings of the 1st International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS. pp. 153–159. IEEE Computer Society (2020). https://doi.org/10.1109/ACSOS49614.2020.00037
2. Audrito, G., Bergamini, S., Damiani, F., Viroli, M.: Resilient distributed collection through information speed thresholds. In: Proceedings of the 22nd International Conference on Coordination Models and Languages (COORDINATION). Lecture Notes in Computer Science, vol. 12134, pp. 211–229. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_14
3. Audrito, G., Casadei, R., Damiani, F., Pianini, D., Viroli, M.: Optimal resilient distributed data collection in mobile edge environments. Computers & Electrical Engineering (2021). https://doi.org/10.1016/j.compeleceng.2021.107580
4. Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M.: Functional programming for distributed systems with XC. In: 36th European Conference on Object-Oriented Programming, ECOOP 2022. LIPIcs, vol. 222, pp. 20:1–20:28. Schloss Dagstuhl (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.20
5. Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M.: The exchange calculus (XC): A functional programming language design for distributed collective systems. J. Syst. Softw. **210**, 111976 (2024). https://doi.org/10.1016/J.JSS.2024.111976
6. Audrito, G., Casadei, R., Torta, G.: On the dynamic evolution of distributed computational aggregates. In: 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). pp. 37–42 (2022). https://doi.org/10.1109/ACSOSC56246.2022.00024
7. Audrito, G., Rapetta, L., Torta, G.: Extensible 3d simulation of aggregated systems with FCPP. In: Coordination Models and Languages - 24th International Conference, COORDINATION 2022 Proceedings. LNCS, vol. 13271, pp. 55–71. Springer (2022). https://doi.org/10.1007/978-3-031-08143-9_4
8. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Logic **20**(1), 5:1–5:55 (2019). https://doi.org/10.1145/3285956
9. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, chap. 16, pp. 436–501. IGI Global (2013). https://doi.org/10.4018/978-1-4666-2092-6.ch016
10. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer **48**(9) (2015). https://doi.org/10.1109/MC.2015.261
11. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. TAAS **1**(2), 223–259 (2006)
12. Lluch-Lafuente, A., Loreti, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. Log. Methods Comput. Sci. **13**(1) (2017). https://doi.org/10.23638/LMCS-13(1:13)2017
13. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The tota approach. ACM Trans. on Software Engineering Methodologies **18**(4), 1–56 (2009). https://doi.org/10.1145/1538942.1538945
14. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems. In: Petta, P., Tolksdorf, R., Zam-

bonelli, F. (eds.) Engineering Societies in the Agents World III, Third International Workshop, ESAW 2002, Madrid, Spain, September 16-17, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2577, pp. 68–81. Springer (2002). https://doi.org/10.1007/3-540-39173-8_6
15. Menezes, R., Tolksdorf, R.: Adaptiveness in linda-based coordination models. In: Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering, LNAI, vol. 2977, pp. 212–232. Springer (January 2004)
16. Omicini, A., Viroli, M.: Coordination models and languages: From parallel computing to self-organisation. The Knowledge Engineering Review **26**(1), 53–59 (2011)
17. Torrent-Moreno, M., Corroy, S., Schmidt-Eisenlohr, F., Hartenstein, H.: IEEE 802.11-based one-hop broadcast communications: understanding transmission success and failure under different radio propagation environments. In: Alba, E., Chiasserini, C., Abu-Ghazaleh, N.B., Cigno, R.L. (eds.) Proceedings of the 9th International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems, MSWiM 2006, Terromolinos, Spain, October 2-6, 2006. pp. 68–77. ACM (2006). https://doi.org/10.1145/1164717.1164731
18. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2), 16:1–16:28 (2018). https://doi.org/10.1145/3177774
19. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/j.jlamp.2019.100486
20. Viroli, M., Pianini, D., Beal, J.: Linda in space-time: An adaptive coordination model for mobile ad-hoc environments. In: Sirjani, M. (ed.) Coordination Models and Languages - 14th International Conference, COORDINATION 2012, Stockholm, Sweden, June 14-15, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7274, pp. 212–229. Springer (2012). https://doi.org/10.1007/978-3-642-30829-1_15