Replica-Based High-Performance Tuple Space Computing

Marina Andrić[1], Rocco De Nicola[1], and Alberto Lluch Lafuente[2](✉)

[1] IMT Institute for Advanced Studies Lucca, Lucca, Italy
[2] DTU Compute, Technical University of Denmark, Kgs. Lyngby, Denmark
{marina.andric,rocco.denicola}@imtlucca.it, albl@dtu.dk

Abstract. We present the tuple-based coordination language RepliKlaim, which enriches Klaim with primitives for replica-aware coordination. Our overall goal is to offer suitable solutions to the challenging problems of data distribution and locality in large-scale high performance computing. In particular, RepliKlaim allows the programmer to specify and coordinate the replication of shared data items and the desired consistency properties. The programmer can hence exploit such flexible mechanisms to adapt data distribution and locality to the needs of the application, so to improve performance in terms of concurrency and data access. We investigate issues related to replica consistency, provide an operational semantics that guides the implementation of the language, and discuss the main synchronization mechanisms of our prototypical run-time framework. Finally, we provide a performance analysis, which includes scenarios where replica-based specifications and relaxed consistency provide significant performance gains.

1 Introduction

The scale of parallel and distributed computing systems is growing fast to meet the computational needs of our society, ranging from (big) data-driven analyses to massively distributed services. One of the key points in parallel and distributed computing is the division and communication of data between computational entities. Better performances are achieved with increased data locality and minimized data communication. Increasing data locality can be easily achieved by replicating data, but this comes of course at a high price in terms of synchronization if replicated data need to be kept consistent. As a matter of fact the trade-off between consistency and performance is one of the big dilemmas in distributed and parallel computing.

The recent years have seen the advent of technologies that provide software engineers and programmers with flexible mechanisms to conveniently specify data locality, communication and consistency to the benefit of their applications. A pragmatical example for large-scale distributed services is the Google Cloud Storage[1] service, that allows users to geographically specify data locality (to

Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

[1] https://cloud.google.com/storage/

© IFIP International Federation for Information Processing 2015
T. Holvoet and M. Viroli (Eds.): COORDINATION 2015, LNCS 9037, pp. 3–18, 2015.
DOI: 10.1007/978-3-319-19282-6_1

reduce cost and speed up access) and provides different consistency levels (e.g. strong and eventual consistency) for different operations (e.g. single data and list operations).

In the realm of parallel computing, one can find several high performance computing languages that offer similar support for designing efficient applications. De-facto standards such as OpenMP[2] (for shared memory multiprocessing) and MPI[3] (for message-passing large-scale distributed computing) are being challenged by new languages and programming models that try to address concerns such as the *memory address to physical location* problem. This is a general concern that needs to be solved when programming scalable systems with a large number of computational nodes. In languages X10[4], UPC[5] and Titanium [19], this problem is solved via the *partitioned global address space* (PGAS) model. This model is a middle way approach between shared-memory (OpenMP) and distributed-memory (MPI) programming models, as it combines performance and data locality (partitioning) of distributed-memory and global address space of a shared-memory model. In the PGAS model, variables and arrays are either *shared* or *local*. Each processor has private memory for local data and shared memory for globally shared data.

Summarizing, two key aspects in the design of distributed and parallel systems and software are *data locality* and *data consistency*. A proper design of those aspects can bring significant performance advantages, e.g. in terms of minimization of communication between computational entities.

Contribution. We believe that those two aspects cannot be hidden to the programmer of the high performance applications of the future. Instead, we believe that programmers should be equipped with suitable primitives to deal with those aspects in a natural and flexible way. This paper instantiates such philosophy in the coordination language RepliKlaim, a variant of Klaim [12] with first-class features to deal with data locality and consistency. In particular, the idea is to let the programmer specify and coordinate data replicas and operate on them with different levels of consistency. The programmer can hence exploit such flexible mechanisms to adapt data distribution and locality to the needs of the application, so to improve performance in terms of concurrency and data access. We investigate issues related to replica consistency, provide an operational semantics that guides the implementation of the language, and discuss the main synchronisation mechanisms of our implementation. Finally, we provide a performance evaluation study in our prototype run-time system. Our experiments include scenarios where replica-based specifications and relaxed consistency provide significant performance gains.

Structure of the Paper. This paper is organised as follows. Section 2 presents RepliKlaim and discusses some examples that illustrate its semantics. Section 3

[2] www.openmp.org/
[3] http://www.open-mpi.org/
[4] x10-lang.org
[5] upc.lbl.gov

$$
\begin{array}{llll}
N & ::= & \mathbf{0} \mid l :: [K,P] \mid N \parallel N & \text{(networks)} \\
K & ::= & \emptyset \mid \langle et_i, L\rangle \mid K, K & \text{(repositories)} \\
P & ::= & \mathsf{nil} \mid A.P \mid P+P \mid P \mid P & \text{(processes)} \\
A & ::= & \mathsf{out_s}(t_i)@L \mid \mathsf{in_s}(T_\iota)@\ell \mid \mathsf{read}(T_\iota)@\ell & \text{(strong actions)} \\
 & & \mathsf{out_w}(t_i)@L \mid \mathsf{in_w}(T_\iota)@\ell \mid & \text{(weak actions)} \\
 & & \mathsf{in_u}(T_\iota, L)@\ell \mid \mathsf{out_u}(et_i, L)@\ell & \text{(unsafe actions)} \\
L & ::= & \epsilon \mid \ell \mid \underline{\ell} \mid L \bullet L & \text{(locations)}
\end{array}
$$

Fig. 1. Syntax of RepliKlaim

provides some details about our prototype implementation and presents a set of performance experiments. Section 4 discusses related works. Section 5 concludes the paper and identifies possible future works.

2 RepliKlaim: Klaim with Replicas

We present our language RepliKlaim in this section. We start with the definition of the syntax in Section 2.1 and proceed then with the description of the operational semantics in Section 2.2. Section 2.3 discusses some examples aimed at providing some insights on semantics, implementation and performance aspects, later detailed in Section 3.

2.1 RepliKlaim: Syntax

The syntax of RepliKlaim is based on Klaim [12]. The main differences are the absence of mobility features (i.e. the eval primitive and allocation environments) and the extension of communication primitives to explicitly deal with replicas.

Definition 1 (RepliKlaim Syntax). *The syntax of RepliKlaim is defined by the grammar of Fig. 1, where $\mathcal{L}$ is a set of locations (ranged over by $\ell, \ell', \dots$), $\mathcal{U}$ is a set of values (ranged over by $u, v, \dots$), $\mathcal{V}$ is a set of variables (ranged over by $x, y, \dots$), $!\mathcal{V}$ denotes the set binders over variables in $\mathcal{V}$ (i.e. $!x, !y, \dots$), $\mathcal{I}$ is a set of tuple identifiers (ranged over by i, i', j, j'), $\mathcal{T} \subseteq (\mathcal{U} \cup \mathcal{V})^*$ is a set of $\mathcal{I}$-indexed tuples (ranged over by $t_i, t'_{i'}, \dots$), $\mathcal{ET} \subseteq (\mathcal{U}^*$ is a set of $\mathcal{I}$-indexed evaluated tuples (ranged over by $et_i, et'_{i'}, \dots$), and $\mathcal{TT} \subseteq (\mathcal{U} \cup \mathcal{V} \cup !\mathcal{V})^*$ is a set of templates (ranged over by $T_\iota, T'_{\iota'}, \dots$, with $\iota \in \mathcal{I} \cup !\mathcal{V}$).*

Networks. A RepliKlaim specification is a *network* N, i.e. a possibly empty set of *components* or *nodes*.

Components. A component $\ell :: [K, P]$ has a locality name ℓ which is unique (cf. well-formedness in Def. 2), a *data repository* K, and set of processes P. Components may model a data-coherent unit in a large scale system, where each unit has dedicated memory and computational resources, e.g. an SMP node in a many-core machine.

Repositories. A data repository K is a set of data items, which are pairs of identifier-indexed tuples and their replication information. In particular a data item is a pair $\langle et_i, L\rangle$, where t_i is a tuple, i is a unique identifier of the tuple, and L is a list of localities where the tuple is replicated. For a data item $\langle et_i, L\rangle$ with $|L| > 1$ we say that t_i is *shared* or *replicated*. We use indexed tuples in place of ordinary anonymous tuples to better represent long-living data items such as variables and objects that can be created and updated. We require the replication information to be *consistent* (cf. well-formedness in Def. 2). This property is preserved by our semantics, as we shall see.

It is worth to note that a locality ℓ in L can appear as ℓ or as $\underline{\ell}$. The latter case denotes a sort of *ownership* of the tuple. We require each replicated tuple to have exactly one owner (cf. well-formedness in Def. 2). This is fundamental to avoid inconsistencies due to concurrent *weak* (asynchronous) retrievals or updates of a replicated tuple. This issue will be explained in detail later.

Processes. Processes are the main computational units and can be executed concurrently either at the same locality or at different localities. Each process is created from the nil process, using the constructs for *action prefixing* $(A.P)$, *non-deterministic choice* $(P_1 + P_2)$ and *parallel execution* $(P_1 \mid P_2)$.

Actions and Targets. The actions of RepliKlaim are based on standard primitives for tuple spaces, here extended to suitably enable replica-aware programming. Some actions are exactly as in Klaim. For instance, $\mathsf{read}(t_i)@\ell$ is the standard non-destructive read of Klaim.

The standard output operation is enriched here to allow a list of localities L as target. RepliKlaim features two variants of the output operation: a *strong* (i.e. atomic) one and a *weak* (i.e. *asynchronous*) one. In particular, $\mathsf{out}_\alpha(\mathsf{t_i})@\mathsf{L}$ is used to place the shared tuple t_i at the data repositories located on sites $l \in L$ atomically or asynchronously (resp. for $\alpha = \mathsf{s}, \mathsf{w}$). In this way the shared tuple is replicated on the set of sites designated with L. In RepliKlaim output operations are blocking: an operation $\mathsf{out}_\alpha(\mathsf{t_i})@\mathsf{L}$ cannot be enacted if an i-indexed tuple exists at L. This is necessary to avoid inconsistent versions of the same data item in the same location to co-exist. Hence, before placing a new version of a data item, the previous one needs to be removed. However, we will see that weak consistency operations still allow inconsistent versions of the same data item to co-exist but in *different* locations.

As in the case of output operations, RepliKlaim features two variants of the standard destructive operation in: a *strong* input $\mathsf{in_s}$ and a *weak* input $\mathsf{in_w}$. A strong input $\mathsf{in_s}(T_\iota)@\ell$ retrieves a tuple et_i matching T_ι at ℓ and atomically removes all replicas of et_i. A weak input $\mathsf{in_w}(T_\iota)@\ell$ tries to asynchronously remove all replicas of a tuple et_i matching T_ι residing in ℓ. This means that replicas are not removed simultaneously. Replicas in the process of being removed are called *ghost* replicas, since they are reminiscent of the *ghost* tuples of [25,14] (cf. the discussion in Section 4).

RepliKlaim features two additional (possibly) *unsafe* operations: $\mathsf{out_u}(et_i, L)@\ell$ puts a data item $\langle et_i, L\rangle$ at all locations in L, while $\mathsf{in_u}(T_\iota, L)@\ell$ retrieves a tuple

$$
\begin{array}{cc}
P + (Q + R) \equiv (P + Q) + R & P \mid Q \equiv Q \mid P \\
P + \mathsf{nil} \equiv P & N \parallel (M \parallel W) \equiv (N \parallel M) \parallel W \\
P + Q \equiv Q + P & N \parallel \mathbf{0} \equiv N \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & N \parallel M \equiv M \parallel M \\
P \mid \mathsf{nil} \equiv P & \ell :: [K, P] \equiv \ell :: [K, \mathsf{nil}] \parallel \ell :: [\emptyset, P]
\end{array}
$$

Fig. 2. Structural congruence for RepliKlaim

et_i matching T_ι at ℓ and does *not* remove the replicas of et_i. These operations are instrumental for the semantics and are not meant to appear in user specifications.

As we have seen, the syntax of RepliKlaim admits some terms that we would like to rule out. We therefore define a simple notion of well-formed network.

Definition 2 (Well-formedness). *Let N be a network. We say that N is well formed if:*

1. *Localities are unique, i.e. no two distinct components $\ell :: [K, P]$, $\ell :: [K', P']$ can occur in N;*
2. *Replication is consistent, i.e. for every occurrence of $\ell :: [(K, \langle et_i, L \rangle), P]$ in a network N it holds that $\ell \in L$ and for all (and only) localities $\ell' \in L$ we have that component ℓ' is of the form $\ell' :: [(K', \langle et'_i, L \rangle), P']$. Note that t' is not required to be t since we allow relaxed consistency of replicas.*
3. *Each replica has exactly one owner, i.e. every occurrence of L has at most one owner location $\underline{\ell}$.*
4. *Tuple identifiers are unique, i.e. there is no K containing two data items $\langle et_i, L \rangle$, $\langle et'_i, L' \rangle$. Note that this guarantees local uniqueness; global uniqueness is implied by condition (2).*

Well-formedness is preserved by the semantics, but as usual we admit some intermediate bad-formed terms which ease the definition of the semantics.

We assume the standard notions of free and bound variables, respectively denoted by $fn(\cdot)$ and $bn(\cdot)$, as well as the existence of a suitable operation for matching tuples against templates, denoted $match(T_\iota, t_i)$ which yields a substitution for the bound variables of T_ι. Note that ι may be a bound variable to record the identifier of the tuple.

2.2 RepliKlaim: Semantics

RepliKlaim terms are to be intended up to the structural congruence induced by the axioms in Fig 2 and closed under reflexivity, transitivity and symmetry.

$$\frac{}{A.P \xrightarrow{A} P}\ (\textsc{ActP}) \qquad \frac{P \xrightarrow{A} P'}{P+Q \xrightarrow{A} P'}\ (\textsc{Choice}) \qquad \frac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q}\ (\textsc{Par})$$

$$\frac{P \xrightarrow{\mathsf{out_s}(t_i)@L} P' \qquad \forall \ell' \in L . \nexists et', L' . \langle et'_i, L' \rangle \in K_{\ell'}}{N \| \ell::[K,P] \| \Pi_{\ell' \in L} \ell'::[K_{\ell'}, P_{\ell'}] \longrightarrow N \| \ell::[K,P'] \| \Pi_{\ell' \in L} \ell'::[(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}]}\ (\textsc{OutS})$$

$$\frac{P \xrightarrow{\mathsf{out_w}(t_i)@L} P' \qquad \underline{\ell''} \in L \qquad \nexists et', L' . \langle et'_i, L' \rangle \in K_{\ell''}}{N \| \ell::[K,P] \| \ell''::[K_{\ell''}, P_{\ell''}] \longrightarrow N \| \ell::[K,P'] \| \ell''::[(K_{\ell''}, \langle et_i, L \rangle), P_{\ell''} | \Pi_{\ell' \in (L \setminus \ell'')} \mathsf{out_u}(et_i, L)@\ell']}\ (\textsc{OutW})$$

$$\frac{P \xrightarrow{\mathsf{out_u}(et_i, L)@\ell} P' \qquad \nexists et', L' . \langle et'_i, L' \rangle \in K_{\ell}}{N \| \ell::[K,P] \longrightarrow N \| \ell::[(K, \langle et_i, L \rangle), P']}\ (\textsc{OutU})$$

$$\frac{P \xrightarrow{\mathsf{in_s}(T_\iota)@\ell''} P' \qquad \ell'' \in L \qquad \sigma = match(T_\iota, et_i)}{N \| \ell::[K,P] \| \Pi_{\ell' \in L} \ell'::[(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell::[K, P'\sigma] \| \Pi_{\ell' \in L} \ell'::[K_{\ell'}, P_{\ell'}]}\ (\textsc{InS})$$

$$\frac{P \xrightarrow{\mathsf{in_w}(T_\iota)@\ell''} P' \qquad \ell'' \in L \qquad \underline{\ell'} \in L \qquad \sigma = match(T_\iota, et_i)}{N \| \ell::[K,P] \| \ell'::[(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell::[K, P'\sigma] \| \ell'::[K_{\ell'}, P_{\ell'} | \prod_{\ell''' \in (L \setminus \ell')} \mathsf{in_u}(et_i, L)@\ell''']}\ (\textsc{InW})$$

$$\frac{P \xrightarrow{\mathsf{in_u}(T_\iota, L)@\ell'} P' \qquad \sigma = match(T_\iota, et_i)}{N \| \ell::[K,P] \| \ell'::[(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell::[K, P'\sigma] \| \ell'::[K_{\ell'}, P_{\ell'}]}\ (\textsc{InU})$$

$$\frac{P \xrightarrow{\mathsf{read}(T_\iota)@\ell'} P' \qquad \sigma = match(T_\iota, et_i)}{N \| \ell::[K,P] \| \ell'::[(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell::[K, P'\sigma] \| \ell'::[(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}]}\ (\textsc{Read})$$

Fig. 3. Operational semantics of RepliKlaim

As usual, besides axiomatising the essential structure of RepliKlaim systems, the structural congruence allows us to provide a more compact and simple semantics. The axioms of the structural congruence are standard. We just remark the presence of a *clone* axiom (bottom right) which is similar to the one used in early works on Klaim. In our case, this clone axiom allows us to avoid cumbersome semantic rules for dealing with multiparty synchronisations where the subject component is also an object of the synchronisation (e.g. when a component ℓ removes a shared tuple t_i that has a replica in ℓ itself). The clone axiom allows a component to participate in those interactions, by separating the processes (the subject) from the repository (the object). It is worth to note that this axiom does not preserve well-formedness (uniqueness of localities is violated).

The operational semantics in Fig. 3 mixes an SOS style for collecting the process actions (cf. rules ActP, Choice and Par) and reductions for the evolution of nets. The standard congruence rules are not included for simplicity.

It is worth to remark that the replicas located at the owner are used in some of the rules as a sort of tokens to avoid undesirable race conditions. The role of such tokens in inputs and outputs is dual: the replica must *not* exist for outputs to be enacted, while the replica *must* exist for inputs to be enacted.

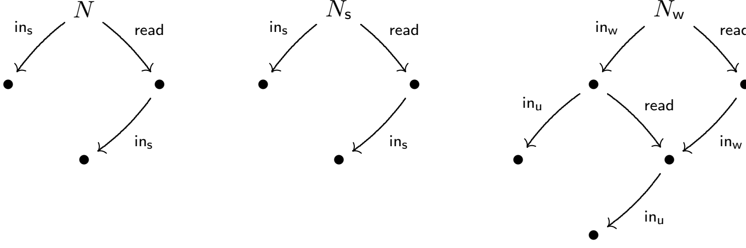

Fig. 4. Concurrent reads and inputs with no replicas (left), replicas and strong input (center) and weak input (right)

Rule OUTS deals with a strong output $\mathsf{out}(et_i)@L$ by putting the tuple et_i in all localities in L. However, the premise of the rule requires a version of data item i (i.e. a tuple et_i') to *not* exist in the repository of the owner of et_i (ℓ''). Rule OUTW governs weak outputs of the form $\mathsf{out}(et_i)@L$ by requiring the absence of a version of data item i. The difference with respect to the strong output is that the effect of the rule is that of creating a set of processes that will take care of placing the replicas in parallel, through the unsafe output operation. Such operation is handled by rule OUTU which is very much like a standard Klaim rule for ordinary outputs, except that the operation is blocking to avoid overwriting existing data items.

Rule INS deals with actions $\mathsf{in}(T_\iota)@\ell$ by retrieving a tuple et_i matching T_ι from locality ℓ, and from all localities containing a replica of it. Rule INW retrieves a tuple et_i from an owner ℓ' of a tuple that has a replica in the target ℓ. As a result, processes are installed at ℓ' that deal with the removal of the remaining replicas in parallel (thus allowing the interleaving of read operations). As in the case of weak outputs, weak inputs resort to unsafe inputs. Those are handled by rule INU, which is like a standard input rule in Klaim.

Finally, rule READ is a standard rule for dealing with non-destructive reads.

2.3 RepliKlaim: Examples

We provide here a couple of illustrative examples aimed at providing insights on semantics, implementation and performance aspects.

Concurrent Reads and Inputs. The following example illustrates three ways of sharing and accessing a tuple and is meant to exemplify the benefit of replicas and weak inputs. The example consists of the networks

$$N \doteq \ell_1 :: [\langle et_i, \ell_1 \rangle, \mathsf{in_s}(et_j)@\ell_1] \quad \| \; \ell_2 :: [\emptyset, \mathsf{read}(et_j)@\ell_1]$$
$$N_\alpha \doteq \ell_1 :: [\langle et_i, \{\underline{\ell_1}, \ell_2\} \rangle, \mathsf{in}_\alpha(et_j)@\ell_1] \; \| \; \ell_2 :: [\langle et_i, \{\underline{\ell_1}, \ell_2\} \rangle, \mathsf{read}(et_\mathrm{j})@\ell_2]$$

with $\alpha \in \{\mathsf{s}, \mathsf{w}\}$. The idea is that in N a tuple has to be accessed by both ℓ_1 and ℓ_2 is shared in the traditional Klaim way: it is only stored in one location

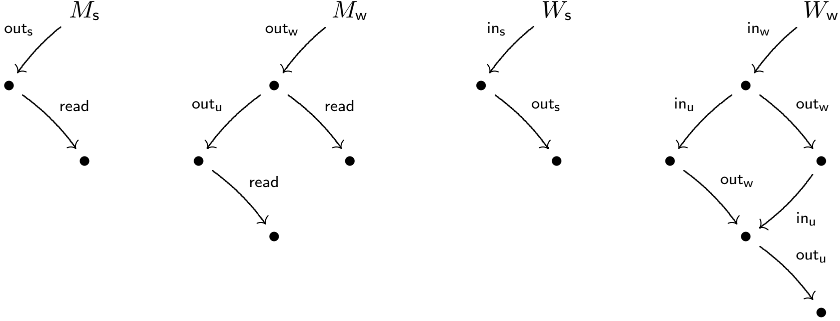

Fig. 5. Transitions for M_{s} (concurrent read and strong output), M_{w} (concurrent read and weak output), W_{s} (concurrent strong input and strong output) and W_{w} (concurrent weak input and weak output)

(namely, ℓ_1) with no replicas. To the contrary, N_α models the same scenario with explicit replicas. The tuple et_i is replicated at both ℓ_1 and ℓ_2, possibly after some process executed $\mathsf{out}(et_i)@\{\underline{\ell_1}, \ell_2\}$. Networks N_{s} and N_{w} differ in the way the tuple et_i is retrieved by ℓ_1: using strong or weak input, respectively. Fig. 4 depicts the transition systems for the three networks, where the actual description of the reachable states is not provided due to lack of space and due to the simplicity of the example. The transition systems of N and N_{s} are similar but differ in the way the transitions are computed. In N, the input is local to ℓ_1, but the read is remote (from ℓ_2 to ℓ_1), while in N_{s} the input is global (requires a synchronization of ℓ_1 and ℓ_2 to atomically retrieve all replicas of et_i), and the read is local to ℓ_2. The main point in N_{w} is that the process in ℓ_2 can keep reading the ghost replicas of et_i even after ℓ_1 started retrieving it.

Concurrent Reads and Outputs. The next example illustrates (see also Fig. 5) the interplay of reads with strong and weak outputs.

$$M_\alpha \doteq \ell_1 :: [\emptyset, \mathsf{out}_\alpha(et_i)@\{\underline{\ell_1}, \ell_2\}] \parallel \ell_2 :: [\emptyset, \mathsf{read}(et_j)@\ell_1]$$

with $\alpha \in \{\mathsf{s}, \mathsf{w}\}$. The idea is that component ℓ_1 can output a tuple with replicas in ℓ_1 and ℓ_2 in a strong or weak manner, while ℓ_2 is trying to read the tuple from ℓ_1. In the strong case, the read can happen only after all replicas have been created. In the weak case, the read can be interleaved with the unsafe output.

Concurrent inputs and outputs. The last example (see also Fig. 5) illustrates the update of a data item using strong and weak operations.

$$W_\alpha \doteq \ell_1 :: [\emptyset, \mathsf{in}_\alpha(et_i)@\{\underline{\ell_1}, \ell_2\}.\mathsf{out}_\alpha(f(et)_i)@\{\underline{\ell_1}, \ell_2\}] \parallel \ell_2 :: [\emptyset, \mathsf{nil}]$$

with $\alpha \in \{\mathsf{s}, \mathsf{w}\}$. The idea is that component ℓ_1 retrieves a tuple and then outputs an updated version of it (after applying function f). Relaxing consistency from s to w increases the number of interleavings.

3 Performance Evaluation

We describe in this section our prototype implementation and present a set of experiments aimed at showing that an explicit use of replicas in combination with weakly consistent operations then provide significant performance advantages.

Implementing RepliKlaim in KLAVA. Our prototype run-time framework is based on KLAVA, a Java package used for implementing distributed applications based on Klaim. KLAVA is a suitable framework for testing our hypothesis as it provides a set of process executing engines (nodes) connected in a network via one of the three communication protocols (TCP, UDP, local pipes). The current implementation of RepliKlaim is based on an encoding of RepliKlaim into standard Klaim primitives. We recall the main Klaim primitives we use in the encoding: $\mathsf{in}(T)@\ell$ destructively retrieves a tuple matching T in location ℓ. The operation is blocking until a matching tuple is found; $\mathsf{read}(T)@\ell$: non-destructive variant of in; $\mathsf{out}(t)@\ell$: inserts a tuple t into the tuple space located at ℓ. The actual encoding is based on the operational semantics presented in Fig. 3, which already uses some operations that are close to those of Klaim, namely the unsafe operations in_u and out_u. The rest of the machinery (atomicity, etc.) is based on standard synchronisation techniques.

Experiments: Hypothesis. The main hypothesis of our experiments is that better performances are achieved with improved data locality and data communication minimized through the use of replicated tuples and weak operations. Indeed, minimizing data locality can be easily done by replicating data, however it comes at a cost in terms of synchronization if replicated data need to be kept consistent (e.g. when using strong inputs and outputs). As we shall see, our experimental results show how the ratio between the frequencies of read and update (i.e. sequences of inputs and outputs on the same data item) operations affects the performance of three different versions of a program: a *traditional* one that does not use replicas, and two versions using replicas: one using strong (consistent) operations and another one using weak (weak consistent) operations. We would only like to remark that we had to deviate in one thing from the semantics: while spawning parallel processes in rules InW and OutW to deal with the asynchronous/parallel actions on replicas seems very appealing, in practice performing such operations in sequence showed to be more efficient. Of course in general the choice between parallel and sequential composition of such actions depends on several aspects, like the number of available processors, the number of processes already running in the system and the size of the data being replicated.

Experiments: Configuration of the Scenario.[6] The general idea of the scenario we have tested is that multiple nodes are concurrently working (i.e. performing

[6] The source code and Klava library are available online at http://sysma.imtlucca.it/wp-content/uploads/2015/03/RepliKlaim-test-examples.rar

inputs, reads and outputs) on a list whose elements can be scattered on various nodes. A single element (i.e. the counter) is required to indicate the number of the next element that can be added. In order to add an element to the list, the counter is removed using an input, the value of the counter is increased and the tuple is re-inserted, and then a new list element is inserted. We call such a sequence of input and output operations on the same data item (i.e. the counter) an *update* operation.

Each of the nodes is running processes that perform read or update operations. Both reader and updater processes run in loops. We fix the number of updates to 10, but vary the number of read accesses (20, 30, 50, 100, 150, 200). We consider two variants of the scenario. The first variant has 3 nodes: one node containing just one reader process, another node containing just one updater process and a last one containing both a reader and an updater process. The second variant has 9 nodes, each containing process as in the previous case, i.e. this scenario is just obtained by triplicating the nodes of the previous scenario. The main point for considering these two variants is that we run the experiment in a dual core machine, so that in the first case one would ideally have all processes running in parallel, while this is not the case in the second variant.

Formally, the RepliKlaim nets N we use in our experiments are specified as follows

$$N \doteq \prod_{i=1}^{n} \left\{ \ell_{i,1} :: [\emptyset, \mathsf{P}_1(\ell_{i,1})] \parallel \ell_{i,2} :: [\emptyset, \mathsf{P}_2(\ell_{i,2})] \parallel \ell_{i,3} :: [\emptyset, \mathsf{P}_1(\ell_{i,3}) \mid \mathsf{P}_2(\ell_{i,3})] \right\}$$

where P_1 is an updater process and P_2 is a reader process, both parametric with respect to the locality they reside on. P_1 is responsible for incrementing the counter and adding a new list element, while P_2 only reads the current number of list elements. For the scalability evaluation we compare results for nets obtained when $n = 1$ and $n = 3$, meaning that corresponding nets have 3 and 9 nodes respectively. Our aim is to compare the following three alternative implementations of processes P_1 and P_2 which offer the same functionality, but exhibit different performances:

Program $\mathsf{no} - \mathsf{replicas}$: this implementation follows a standard approach that does not make use of replica-based primitives. The idea here is that the shared tuple is stored only in one location, with no replicas. The consistency of such model is obviously strong, as there are no replicas. Local access to the shared tuple is granted only to processes running on the specified location, while other processes access remotely. In the begining we assume that one of the sites has executed $\mathsf{out_s}(\mathit{counter}_a)@\ell_1$ which places the counter tuple $\mathit{counter}_a$ at place ℓ_1, with a being a unique identifier. Then processes P_1 and P_2 can be expressed as follows:

$$\begin{aligned}
\mathsf{P}_1(\mathit{self}) &\equiv \mathsf{in_s}(\mathit{counter}_a)@\ell_1.\mathsf{out_s}(f(\mathit{counter}_a))@\ell_1.\mathsf{out_s}(\mathit{lt}_{a_{counter}})@\mathit{self}.\mathsf{P}_1\\
\mathsf{P}_2(\mathit{self}) &\equiv \mathsf{read}(T_a)@\ell_1.\mathsf{P}_2
\end{aligned}$$

where $f(\cdot)$ refers to the operation of incrementing the counter and lt refers to the new list element which is added locally after the shared counter had

been incremented. Note that we use a as unique identifier for the counter and $a_{counter}$ as unique identifier for the new elements being inserted.

Program $\mathsf{strong-replicas}$: The difference between this model and the non-replicated one is the presence of replicas on each node, while this model also guarantees strong consistency. Concretely, each update of replicated data items is done via operations $\mathsf{in_s}$ and $\mathsf{out_s}$. The formalisation is presented below, after the description of the weak variant of this implementation.

Program $\mathsf{weak-replicas}$: In this variant, the replicas are present on each node, but the level of consistency is weak. This means that interleavings of actions over replicas are allowed. However, to make this program closer to the functionality offered by the above ones, we forbid the co-existence of different versions of the same data item. Such co-existence is certainly allowed in sequences of operations like $\mathsf{in_w}(t_i)@\ell.\mathsf{out_w}(t'_i)@L$ as we have seen in the examples of Section 2.3. To avoid such co-existence, but still allow concurrent reads we use an additional tuple that the updaters used as sort of lock to ensure that outputs (reps. inputs) are only enacted once inputs (resp. outputs) on the same data item are completed on all replicas. Of course, this makes this program less efficient than it could be but it seems a more fair choice for comparison and still our results show its superiority in terms of performance.

In the above two replication-based implementations we assume that the counter is replicated on all nodes by executing $\mathsf{out}_\alpha(counter_a)@\{\underline{\ell_1}, \ell_2, \ell_3\}$ with $\alpha \in \{\mathsf{s}, \mathsf{w}\}$. In this case the processes are specified as:

$$\begin{aligned} \mathsf{P}_1(self) &\equiv \mathsf{in}_\alpha(counter_a)@self.\mathsf{out}_\alpha(f(counter_a))@\{\ell_1, \ell_2, \ell_3\}.\\ &\quad\ \mathsf{out_s}(a_{counter})@self.\mathsf{P}_1 \\ \mathsf{P}_2(self) &\equiv \mathsf{read}(T_a)@self.\mathsf{P}_2 \end{aligned}$$

where the strong and weak variants are obtained by letting α be s and w, respectively.

Experiments: Data and Interpretation. The results of our experiments are depicted in Fig. 6 and 7. The x axis corresponds to the ratio of reads and updates performed by all processes, while the y axis corresponds to the time needed by the processes to complete their computation. We measure the relation between average running time and the ratio between access frequencies. Time is expressed in seconds and presents the average of 15 executions, while the ratio is a number (2, 3, 5, 10, 15, 20). The results obtained for programs $\mathsf{no-replicas}$, $\mathsf{strong-replicas}$ and $\mathsf{weak-replicas}$ are respectively depicted in blue, green and red.

It can be easily observed that when increasing the ratio the $\mathsf{weak-replicas}$ program is the most efficient. This program improves over program $\mathsf{no-replicas}$ only after the ratio of reading operations reaches a certain level that varies from the two variants used (3 and 9 nodes). The variant with 9 nodes requires a higher ratio to show this improvement, mainly due to the fact that the 12 processes of the scenario cannot run in parallel in the dual-core machine we used. Note that $\mathsf{strong-replicas}$ offers the worst performance. Indeed, preserving strong

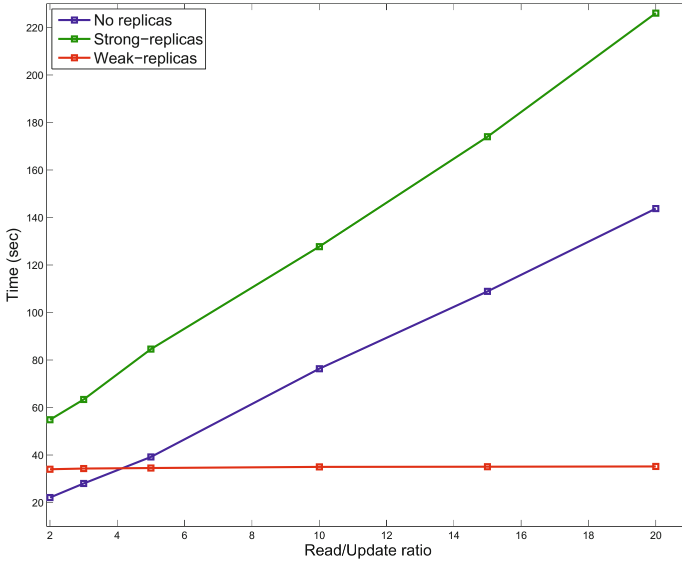

Fig. 6. Comparing three strategies in a scenario with 3 nodes

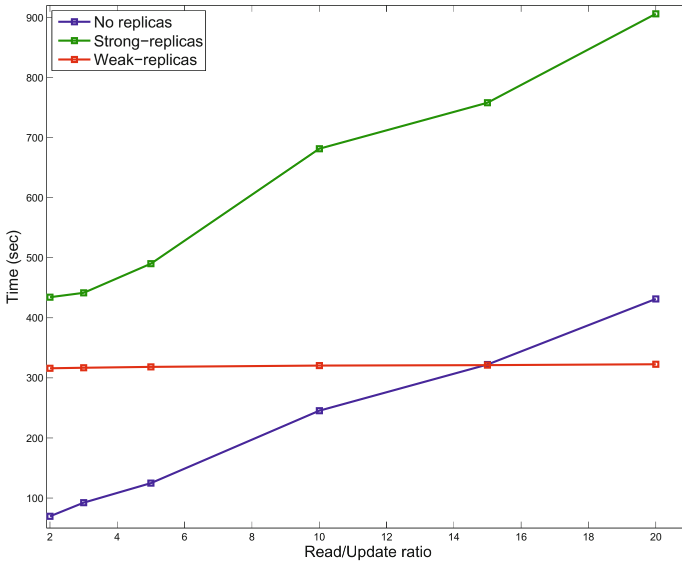

Fig. 7. Comparing three strategies in a scenario with 9 nodes

consistency in presence of replicas is unfeasible in practice because it requires a great deal of synchronization.

4 Related Works

Many authors have investigated issues related to the performance of tuple space implementations and applications of tuple space coordination to large-scale distributed and concurrent systems (cloud computing, high-performance computing, services, etc.). We discuss here some representative approaches that are closely related to our work and, in part, served as inspiration.

One of the first performance improvements for tuple-space implementations was the *ghost* tuple technique, originally presented in [25] and later proven to be correct in [14]. The technique applies to Linda-like languages used in a distributed setting where local tuple replicas are used to improve local operations. Ghost tuple is a local replica of a tuple being destructively read (by a Linda in) operation. The ghost tuple technique allows the non-destructive read of those local replicas (by a Linda read operation). This technique is very similar to our idea of relaxing consistency in tuple space operations. In particular, our local replicas can be seen as ghost tuples as we have mentioned in several occasions in the paper. As a matter of fact, the ghost tuple technique is one of our main sources of inspiration.

Another seminal work considering performance issues in tuple space coordination was the introduction of asynchronous tuple space primitives in Bonita (asynchronous Linda) [24]. This work provided a practical implementation and an illustrative case study to show the performance advantages of asynchronous variants of tuple space primitives for coordinating distributed agents. A thorough theoretical study of possible variants of tuple space operations was later presented in [8]. In particular, the authors study three variants for the output operation: an instantaneous output (where an output can be considered as instantaneous creation of the tuple), and ordered output (where a tuple is placed in the tuple space as one atomic action) and an unordered output (where the tuple is passed to the tuple space handler and the process will continue, the tuple space handler will then place the tuple in the tuple space, not necessarily respecting order of outputs). A clear understanding of (true) concurrency of tuple space operations was developed in [7], where the authors provide a contextual P/T nets semantics of Linda. All these works have inspired the introduction of the asynchronous weak operations in RepliKlaim.

Performance issues have been also considered in tuple space implementations. Besides Klaim implementations [5,4], we mention GigaSpaces [1], a commercial tuple space implementation, Blossom [15], a C++ high performance distributed tuple space implementation, Lime [23], a tuple space implementation tailored for ad-hoc networks, TOTA [22], a middleware for tuple-based coordination in multi-agent systems, and PeerSpace [9] a P2P based tuple space implementation. Moreover, tuple space coordination has been applied and optimised for a large variety of systems where large-scale distribution and concurrency are

key aspects. Among other, we mention large-scale infrastructures [10], cluster computing environments [2], cloud computing systems [17], grid computing systems [21], context-aware applications [3], multi-core Java programs [16], and high performance computing systems [18]. As far as we know, none of the above mentioned implementations treats replicas as first-class programming citizens.

Another set of works that are worth considering are recent technologies for high performance computing. Among them we mention *non-uniform cluster computing* systems, which are built out of multi-core SMP chips with non-uniform memory hierarchies, and interconnected in horizontally scalable cluster configurations such as blade servers. The programming language X10, currently under development, is intended as object-oriented language for programing such systems. A recent formalization of some X10 features can be found in [11]. The main concept of X10 is a notion of *place* which is a collection of threads (activities) and data, and it maps to a data-coherent unit of a large system (e.g. SMP node). In X10 the programmer makes the initial distribution of shared data which is not changed throughout the program execution. Each piece of shared data maps to a single place, and all remote accesses are achieved by spawning (asynchronous) activities. In our language, such concept of place would correspond to a single node. We believe that the concept of replicas introduced in RepliKlaim, can be suitable for modeling high-performance programming using X10-like programming languages.

5 Conclusion

We have presented the tuple-based coordination language RepliKlaim, which enriches Klaim with primitives for replica-aware coordination. RepliKlaim allows the programmer to specify and coordinate the replication of shared data items and the desired consistency properties so to obtain better performances in large-scale high performance computing applications. We have provided an operational semantics to formalise our proposal as well as to guide the implementation of the language, which has been encoded into Klava [5], a Java-based implementation of Klaim. We have also discussed issues related to replica consistency and the main synchronization mechanisms of our implementation. Finally, we have provided a performance evaluation study in our prototype run-time system. Our experiments include scenarios where replica-based specifications and relaxed consistency provide significant performance gains.

We plan to enrich our performance evaluation to consider large-scale distributed systems since our focus so far has been on local concurrent systems. Moreover, we would like to compare our implementation against existing tuple space implementations (cf. the discussion in Section 4). We may also consider other forms of consistency beyond strong and weak, as advocated e.g. in [26,6], and to understand if there are automatic ways to help the programmer decide when to use which form of consistency as done, e.g. in [20]. Another future work we plan to pursue is to apply our approach to the Scel language [13]. One characteristic difference between Scel and Klaim is that the target of tuple operations can be specified by a predicate on the attributes of components.

This provides a great flexibility as it allows to use group-cast operations without explicitly creating groups, called *ensembles* in SCEL. In many applications creating replicas would be a convenient mechanism to share information among groups. However, the dynamicity of ensembles, since components change attributes at run-time and those join and leave ensembles arbitrarily, poses some challenges on the semantics and implementation of shared data items that need to be investigated.

References

1. Gigaspaces technologies ltd, www.gigaspaces.com
2. Atkinson, A.K.: Development and Execution of Array-based Applications in a Cluster Computing Environment. Ph.D. thesis, University of Tasmania (2010)
3. Balzarotti, D., Costa, P., Picco, G.P.: The lights tuple space framework and its customization for context-aware applications. Web Intelligence and Agent Systems 5(2), 215–231 (2007), http://iospress.metapress.com/content/v16153407085177x/
4. Bettini, L., De Nicola, R., Loreti, M.: Implementing mobile and distributed applications in x-klaim. Scalable Computing: Practice and Experience 7(4) (2006), http://www.scpe.org/index.php/scpe/article/view/384
5. Bettini, L., De Nicola, R., Pugliese, R.: Klava: a java package for distributed and mobile applications. Softw., Pract. Exper. 32(14), 1365–1394 (2002), http://dx.doi.org/10.1002/spe.486
6. Brewer, E.: CAP twelve years later: How the "rules" have changed. Computer 45(2), 23–29 (2012)
7. Busi, N., Gorrieri, R., Zavattaro, G.: A truly concurrent view of linda interprocess communication. Tech. rep., University of Bologna (1997)
8. Busi, N., Gorrieri, R., Zavattaro, G.: Comparing three semantics for linda-like languages. Theor. Comput. Sci. 240(1), 49–90 (2000), http://dx.doi.org/10.1016/S0304-3975(99)00227-3
9. Busi, N., Montresor, A., Zavattaro, G.: Data-driven coordination in peer-to-peer information systems. Int. J. Cooperative Inf. Syst. 13(1), 63–89 (2004)
10. Capizzi, S.: A tuple space implementation for large-scale infrastructures. Ph.D. thesis, University of Bologna (2008)
11. Crafa, S., Cunningham, D., Saraswat, V., Shinnar, A., Tardieu, O.: Semantics of (resilient) X10. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 670–696. Springer, Heidelberg (2014)
12. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: a kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering 24(5), 315–330 (1998)
13. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. TAAS 9(2), 7 (2014), http://doi.acm.org/10.1145/2619998
14. De Nicola, R., Pugliese, R., Rowstron, A.: Proving the correctness of optimising destructive and non-destructive reads over tuple spaces. In: Porto, A., Roman, G.-C. (eds.) COORDINATION 2000. LNCS, vol. 1906, pp. 66–80. Springer, Heidelberg (2000)
15. van der Goot, R.: High Performance Linda using a Class Library. Ph.D. thesis, Erasmus University Rotterdam (2001)

16. Gudenkauf, S., Hasselbring, W.: Space-based multi-core programming in java. In: Probst, C.W., Wimmer, C. (eds.) Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, pp. 41–50. ACM (2011), http://doi.acm.org/10.1145/2093157.2093164
17. Hari, H.: Tuple Space in the Cloud. Ph.D. thesis, Uppsala Universitet (2012)
18. Jiang, Y., Xue, G., Li, M., You, J.-y.: Dtupleshpc: Distributed tuple space for desktop high performance computing. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006. LNCS, vol. 4186, pp. 394–400. Springer, Heidelberg (2006)
19. Krishnamurthy, A., Aiken, A., Colella, P., Gay, D., Graham, S.L., Hilfinger, P.N., Liblit, B., Miyamoto, C., Pike, G., Semenzato, L., Yelick, K.A.: Titanium: A high performance java dialect. In: PPSC (1999)
20. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N.M., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: Thekkath, C., Vahdat, A. (eds.) 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012), pp. 265–278. USENIX Association (2012)
21. Li, Z., Parashar, M.: Comet: a scalable coordination space for decentralized distributed environments. In: Second International Workshop on Hot Topics in Peer-to-Peer Systems, HOT-P2P 2005, pp. 104–111. IEEE Computer Society (2005)
22. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. Softw. Eng. Methodol. 18(4) (2009)
23. Murphy, A.L., Picco, G.P., Roman, G.: LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15(3), 279–328 (2006)
24. Rowstron, A.: Using asynchronous tuple-space access primitives (BONITA primitives) for process co-ordination. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 426–429. Springer, Heidelberg (1997)
25. Rowstron, A., Wood, A.: An efficient distributed tuple space implementation for networks of workstations. In: Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y. (eds.) Euro-Par 1996. LNCS, vol. 1123, pp. 510–513. Springer, Heidelberg (1996)
26. Terry, D.: Replicated data consistency explained through baseball. Commun. ACM 56(12), 82–89 (2013), http://doi.acm.org/10.1145/2500500