# Automated Replication of Tuple Spaces via Static Analysis

Aline Uwimbabazi, Omar Inverso, Rocco de Nicola

▶ **To cite this version:**

Aline Uwimbabazi, Omar Inverso, Rocco de Nicola. Automated Replication of Tuple Spaces via Static Analysis. 9th International Conference on Fundamentals of Software Engineering (FSEN), May 2021, Virtual, Iran. pp.18-34, 10.1007/978-3-030-89247-0_2 . hal-04074522

**HAL Id: hal-04074522**
**https://inria.hal.science/hal-04074522**

Submitted on 19 Apr 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

# Automated Replication of Tuple Spaces via Static Analysis

Aline Uwimbabazi[2], Omar Inverso[2], and Rocco De Nicola[1]

[1] IMT School for Advanced Studies, Piazza S. Francesco, 19, 55100 Lucca, Italy
rocco.denicola@imtlucca.it
[2] Gran Sasso Science Institute, Viale F. Crispi, 7, 67100 L'Aquila, Italy
{aline.uwimbabazi, omar.inverso}@gssi.it

**Abstract.** Coordination languages for tuple spaces can offer significant advantages in the specification and implementation of distributed systems, but often do require manual programming effort to ensure consistency. We propose an experimental technique for automated replication of tuple spaces in distributed systems. The system of interest is modelled as a concurrent Go program where different threads represent the behaviour of the separate components, each owning its own local tuple repository. We automatically transform the initial program by combining program transformation and static analysis, so that tuples are replicated depending on the components' read-write access patterns. In this way, we turn the initial system into a replicated one where the replication of tuples is automatically achieved, while avoiding unnecessary replication overhead. Custom static analyses may be plugged in easily in our prototype implementation. We see this as a first step towards developing a fully-fledged framework to support designers to quickly evaluate many classes of replication-based systems under different consistency levels.

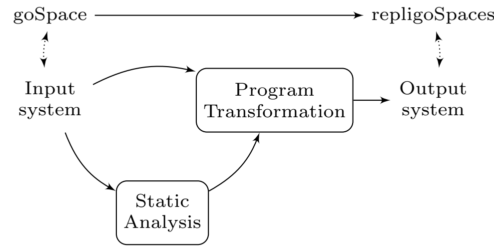
## 1 Introduction

When designing a distributed system, adopting a suitable coordination model can be of fundamental importance. To facilitate the specification of inter-process communication patterns, some coordination languages provide explicit data-access primitives. In Linda [13], processes can concurrently access an associative data store referred to as *tuple space*, where *tuples*, i.e., sequences of typed data atoms, can be stored to or fetched from. Processes synchronise and communicate in this way. Klaim [8] extends this approach to multiple tuple spaces with explicit localities for greater flexibility.

On large, data-intensive distributed systems, techniques to optimise data distribution and locality may significantly improve efficiency. One such technique, *replication*, fits very well within the coordination languages framework. The idea is quite simple: on a store operation, tuples are deployed to a set of target spaces rather than just to a single one. This increases locality and thus reduces latency, but brings along the problem of consistency: once a specific copy of a given tuple is modified, how are the remaining copies to be affected?

RepliKlaim [1] addresses such tension between performance and consistency by extending Klaim's operational semantics with replica-aware data manipulation primitives. The programmer can use these primitives to control the distribution of the data as well as the consistency level. Yet, doing so requires programming ingenuity to specify and coordinate the replicas. Such manual reasoning can be particularly cumbersome because of process interleaving, and hardly feasible in the presence of a large number of complex processes. For the same reasons, evaluating different replication strategies with respect to the intended performance-reliability trade-off can be rather tricky.

In this paper, we address the above shortcomings by proposing an experimental approach to support the design of replication policies in distributed systems that use tuple spaces for process coordination and data storage. More concretely, we present an automated technique to transform the specifications of any such given system into an equivalent version where the tuples are replicated. The overall approach is sketched in the following diagram.




The system of interest is modelled as a concurrent Go [22] program. The behaviour of each system component of the system is defined by a separate thread of the program. Coordination takes place via goSpace [15], a recent Go implementation of Klaim.

To attain automated replication, we first work at the programming interface level, by implementing extended primitives for replica-aware manipulation of tuples. Taking inspiration from the way Klaim's operational semantics was extended in RepliKlaim, we extend goSpace's programming interface to obtain what we call RepligoSpaces. The extended primitives make it possible to target multiple tuple spaces for a single store operation. In addition, an embedded tracking mechanism allows to consistently remove the replicated data at need.

At this point one could immediately obtain full consistency by naively using the extended primitives to replicate every tuple to every shared space in the system. This could be automatically obtained via program transformation, by replacing the tuple manipulation operations with their replica-aware versions, but would likely result in unnecessary overhead. For this reason, between the replica-aware data-handling layer and the program transformation part, we introduce a static analysis pass to refine the target spaces for each store operation.

This simple workflow is easily extensible, given the modularity between the data-handling layer, the program transformation schema, and the static analysis procedure. Different static analysis techniques may be plugged in effortlessly. At the same time, alternative consistency models can be quickly prototyped

by altering the existing replica-aware primitives. We see this as a first step towards developing an integrated framework to experiment with data replication in distributed systems with tuple spaces.

The rest of the paper is organised as follows. Section 2 provides a preliminary introduction to Klaim, RepliKlaim, and pSpaces. Section 3 presents our RepligoSpaces prototype that implements the replica-aware tuple manipulation routines. Section 4 presents our automated replication schema based on static analysis and program transformation. Section 5 provides some details of our prototype implementation and an experimental evaluation of our approach. Sections 6 and 7 discuss related work, conclusion, and future work.

## 2 Preliminaries

In this section, we introduce the main languages representing the starting points of our work. Due to space limits, we limit our description to the minimum necessary. We refer to the cited references for further details.

**Klaim.** Klaim (Kernel Language for Agents Interaction and Mobility) [8] is a coordination language to describe distributed systems and that supports a programming paradigm where both data and processes can be moved from one computing environment to another. In this paper, we focus on the data aspects and refer the reader to the above reference for a general description of Klaim.

Klaim's communication model is based on Linda [8,13], that enables asynchronous communication via a set of operations that allow to exchange information through a shared environment referred to as *tuple space*. A *tuple space* is a collection of tuples. A *tuple* is a finite sequence of *actual* fields (e.g., expressions, values, processes) or *formal* fields (i.e., variables). Tuples that contain variables are also called *templates*. `("Journalist","Sport",2018)` is an example of a tuple, while `("Journalist",category,year)` is a template with two variables `category` and `year`. In Linda and its variants, tuples are retrieved by *pattern matching*. Two tuples match if they have the same number of fields, and all the pairs of fields at the same position *match*: two actual fields match if their values are equal; an actual field and a variable match if they are of the same *type*.

Klaim extends Linda by allowing multiple tuple spaces and offering communication primitives with explicit *localities*. Processes and tuple spaces can be located on different *nodes*, and localities represent unique identifiers for such nodes. Explicit localities allow to distribute and to retrieve data to and from the localities, and to structure the tuple space. In fact, the data manipulation operations of Klaim are based on the standard Linda primitives for tuple spaces but, in addition, they explicitly require the target tuple space as a reference ($@\ell$) to the intended locality.

The non-blocking output operation $\texttt{out}(t)@\ell$ places a tuple $t$ in the tuple space at location $\ell$. The $\texttt{read}(T)@\ell$ operation selects via pattern matching one of the tuples at locality $\ell$ that matches template $T$; this operation blocks until a tuple matching $T$ is found at $\ell$. The $\texttt{in}(T)@\ell$ input operation is similar to `read` but it also removes the matched tuple from the tuple space.

In both Linda and Klaim it is also possible to spawn new processes respectively via `eval(-)` and `eval(-)`$@\ell$. We do not consider process spawning here as it is not currently implemented in RepliKlaim nor pSpaces, and it is not relevant for our analysis.

**RepliKlaim.** RepliKlaim [1] adds to Klaim extended tuple manipulation primitives for replica-aware programming. Similarly to Klaim, RepliKlaim provides a set of blocking and non-blocking operations that add, search and remove tuples from or to tuple spaces. Tuples in RepliKlaim, i.e., *replicated tuples*, have the same format as Klaim's tuples.

The non-blocking output operation `outRK(`$t$`,`$\ell_1 \dots \ell_n$`)` permits to add the shared tuple $t$ to the data repositories located at all localities $\ell \epsilon L$ (L= $\ell_1 \dots \ell_n$) atomically (when strong consistency is required) or asynchronously (in case of weak consistency). Thus, the shared tuple is replicated to every locality in $L$.

The input operation `readRK(`$T$`,`$\ell$`)` reads a tuple space. It uses a pattern $T$ to retrieve a matching tuple (if any) from locality $\ell$, but it does not remove the matching tuple. In case no matching tuple is found, the operation blocks until a matching tuple becomes available. The operation $\texttt{readRK}_{nb}$`(`$T$`,`$\ell$`)` is similar to `readRK(`$T$`,`$\ell$`)` except that it is non-blocking, and returns an empty tuple when no matching tuple is found.

The input operation `inRK(`$T$`,`$\ell$`)` retrieves a tuple matching the pattern $T$ at $\ell$ and atomically removes all replicas of that tuple, thus preserving strong consistency. Operation $\texttt{inRK}_{nb}$`(`$T$`,`$\ell$`)` can also be performed on tuple spaces asynchronously in order to remove all replicas of a tuple that match $T$. This operation only preserves weak consistency. In the rest of the paper, we focus on replication under strong consistency.

**pSpaces.** pSpaces[3] is a family of implementations based on Klaim's formal semantics and targeted at different modern development platforms, such as Go, Java, and Swift. In this paper we focus on the Go implementation, goSpace.

In pSpaces, a *space* is a collection of tuples. Spaces can be either local or remote, in the sense that they can be possibly located on another device. A remote space supports the same operations as for local spaces, but it needs slightly different operations to be created and connected with. Every space is associated with a unique uniform resource identifier (URI) encoded as a string, i.e., the *space identifier*. In the rest of the paper, we make no explicit distinction between local and remote spaces: each component manipulating the tuple spaces is also associated its own URI, which makes it possible to figure out whether a space is local or not.

The implementation of pSpaces relies on communication primitives similar to those of Klaim, essentially a set of blocking and non-blocking actions to add, search, and remove tuples to or from a space. A new tuple $t$ is added to a space $s$ by invoking the non-blocking operation `s.Put(`$t$`)`. The operation `s.Query(`$T$`)` scans a tuple space using pattern matching, blocking until a tuple is found. The

[3] https://github.com/pSpaces/

non-blocking version `s.QueryP(`$T$`)` instead looks for a tuple in the space and returns the tuple, if any, and a boolean value indicating whether the operation was successful. The non-blocking operation `s.GetP(`$T$`)` is similar to `s.QueryP(`$T$`)`, but it also removes the matching tuple, if any, from the space.

## 3 Programming Interface Extension for Replication

We now present RepligoSpaces, our replica-aware extension of goSpace [15]. Both pSpaces and goSpace (Sect. 2) allow to manipulate tuples within a single space. With RepligoSpaces, we instead allow to manipulate tuples across multiple spaces. Our extension follows a similar approach to RepliKlaim with Klaim (Sect. 2). As with RepliKlaim, a store operation takes as an argument the set of targeted spaces. A tuple $t$ is added to spaces $s_1 \dots s_n$ via an `MPut(`$t$`,`$s_1 \dots s_n$`)` operation. The operation `MQueryP(`$s$`,`$T$`)` queries a specific space $s$ for tuples matching pattern $T$. It returns the found tuple, if any, or an empty tuple. The operation `MQuery(`$s$`,`$T$`)` is similar, but blocks until a matching tuple is found. The operation `MGetP(`$s$`,`$T$`)` returns a tuple matching $T$ and removes it from space $s$ and from any other space where it was previously replicated. In the rest of the section, we provide further details about `MPut(`$t$`,`$s_1 \dots s_n$`)`, `MQueryP(`$s$`,`$T$`)` and `MGetP(`$s$`,`$T$`)`. We omit the details for `MQuery(`$s$`,`$T$`)` due to space limits. Note that we are currently only considering strong consistency, i.e., atomic operations on tuples; Also note that in this paper, we are concerned with efficiency (and thus data locality) rather than robustness (redundancy).

**Extended Operations.** The `MPut` operation in Listing 1 adds a tuple to a set of spaces. It takes as input a tuple `t` and a set `S` of space identifiers, in the form of strings that encode their URIs.

```
1  func MPut(t Tuple, Sp Replispace, S []string) Tuple {
2      Sp.mux.Lock()
3
4      // create tuple t' = {t,S}
5      var data []interface{}
6      data = append(data, t.Fields...)
7      data = append(data, S)
8      var t1 Tuple = CreateTuple(data...)
9
10     // add t' to each space in S
11     for i := 0; i < len(S); i++ {
12         Sp.Sp[S[i]].Put(t1.Fields...)
13     }
14
15     Sp.mux.Unlock()
16     return CreateTuple(t1)
17 }
```

Listing 1: The `MPut` operation replicates a tuple over a set of spaces

The idea is then to simply perform a normal goSpace `Put` operation for every space in `S` (lines 10–13). To do so, we need a reference to the space object identified by the URI at any given position of the set `S`. For this, we use a global map `Sp` from URIs to references to `space` objects. Note that this is not a limiting factor as our source transformation procedure will automatically populate `Sp` for us (Sect. 4). Note that the actual tuple being stored is not exactly `t`, but an

extended tuple obtained by appending S to t (lines 4–7). This avoids centralized tracking of the storage locations [1] and simplifies the implementation. We are interested in strong consistency, thus the sequence of Put operations is enclosed in a critical section (lines 2 and 15) to enforce atomicity.

```
1  func MQueryP(p Tuple, Sp Replispace, s Space) Tuple {
2      Sp.mux.Lock()
3
4      // create template p' = {t,S}
5      var y []string // <--- extra field to match the space list S
6      var data []interface{}
7      data = append(data, p.Fields...)
8      data = append(data, &y)
9      var p1 Tuple = CreateTuple(data...)
10
11     // query a tuple via a pattern matching from a specific space
12     t1, e := s.QueryP(p1.Fields...)
13
14     if e == nil {
15         // no error: return the matching tuple without the last field
16         var u = CreateTuple(t1.Fields[:len(t1.Fields)-1]...)
17         Sp.mux.Unlock()
18         return u
19     }
20
21     Sp.mux.Unlock()
22     return CreateTuple()   // returns an empty tuple when no tuple is available
23 }
```

Listing 2: The MQueryP operation to search for a replicated tuple

The MQueryP operation illustrated in Listing 2 searches the given space for tuples matching the given pattern. It takes as input a tuple p (i.e., a pattern) and a space identifier s, and returns as output a tuple, if any. As the result of the previous MPut operation as described above, every stored tuple is extended with an extra field that contains the set of target spaces. Therefore, our search pattern p will need to be adapted accordingly by appending to p an extra field to be used as a placeholder to match the set of targeted spaces in the last field of any stored tuple (line 5 and lines 6–9). The modified pattern p1 so obtained is used instead of p to retrieve matching tuples at space s (line 12). On a successful search (lines 14–19), the last field of the returned tuple is removed as no longer relevant (line 16), and the tuple originally stored is returned. Otherwise, an empty tuple is returned (line 22). Note that the operation MQuery is similar to MQueryP, except that it blocks until a tuple is found.

The MGetP operation illustrated in Listing 3 uses a pattern p to search and remove a matching tuple from space s and any other space where it was replicated. It returns as output the tuple, if any. As for the other operations, the pattern p needs to be adapted with an extra placeholder to match the set of target spaces appended to the stored tuples by an MPut operation. We can then use the modified pattern p1 to scan space s for matching tuples (line 12). On a successful search (lines 14–35), we extract from the matched tuple, the set $S$ of spaces holding a replica of the tuple (line 16). To perform a standard goSpace GetP operation on every space in S, we use the map Sp to retrieve a reference to the relevant space object identified by the URI in S, similarly to the procedure used to implement the MPut operation. Thus, upon searching for the matching tuples from space s (line 12), the list S of all spaces containing a replica of the matching tuple is extracted and transformed in the form of strings of spaces identifiers

(lines 16–19). The loop (lines 22–34) performs a `GetP` operation for every space in the set `S` of space identifiers (line 24) using the map `Sp` and the modified pattern `p1`. On a successful search (lines 26–34), at the last iteration, the tuple is stripped from the extra field containing the target URIs and returned. Note that, since we are assuming only strong operations, it should not be possible for the search to be unsuccessful after passing the first check (line 14). Eventually the operation returns an `empty` tuple in case none is found (line 38).

```
1  func MGetP(p Tuple, Sp Replispace, s Space) Tuple {
2      Sp.mux.Lock()
3
4      // create template p' = {t,S}
5      var y []string // <--- extra field to match the space list S
6      var data []interface{}
7      data = append(data, p.Fields...)
8      data = append(data, &y)
9      var p1 Tuple = CreateTuple(data...)
10
11     //  search the tuple from space s
12     t1, e := s.QueryP(p1.Fields...)
13
14     if e == nil {
15         // extract the list of all spaces
16         var S = (t1.Fields[len(t1.Fields)-1])
17         // transform the interface type of spaces into the string type
18         var v []string
19         v = S.([]string)
20
21         // for each space in the set S of space identifiers
22         for s := range v {
23             // remove the tuple from the relevant spaces
24             u, e1 := Sp.Sp[v[s]].GetP(p1.Fields...)
25
26             if e1 == nil {
27                 if s == len(v)-1 {
28                     // no error: tuple successfully removed from the space
29                     u = CreateTuple(u.Fields[:len(u.Fields)-1]...)
30                     Sp.mux.Unlock()
31                     return u
32                 }
33             }
34         }
35     }
36
37     Sp.mux.Unlock()
38     return CreateTuple()   // returns an empty tuple when no tuple is available
39 }
```

Listing 3: The `MGetP` operation for removing a replicated tuple

## 4 Static Analysis and Program Transformation

We now discuss our approach to automatically transform an initial Go program that uses goSpace for data manipulation into an equivalent program that uses RepligoSpaces (Sect. 3). Intuitively, a fully-consistent but inefficient replicated system may be easily obtained by atomically re-applying every output operation to every shared space (regardless of the originally intended target) using the extended programming interface of Sect. 3. We aim at reducing unnecessary overhead by automatically inspecting the program to refine the set of target spaces. To that end, we rely on static analysis to extract from the initial program the data access patterns, and then use this information during a program transformation phase.

**Input Structure.** Our initial program (Listing 4) is composed of a set $P$ of $n$ parallel processes performing concurrent computations over a set $S$ of $n$ shared tuple spaces. It is worth to observe that the input program may represent an *abstract model* of a more complex system whose computations that do not directly involve tuples are simply abstracted away.

We assume that each process is defined by a separate and unique *process definition function*, and that all such functions are collected into the input program. We denote the process definition functions with $P = p_1, \ldots, p_n$. We also assume that the input program additionally contains a main section where all the shared tuple spaces are created beforehand and associated to unique space identifiers, and all the processes are spawned as separate threads. Finally, we denote with $S = s_1, \ldots, s_n$ the set of spaces shared among the processes, and associate to each process $p_i$ a local tuple space $s_i$; we consider every tuple manipulation operation performed within a process to be a *local* operation if it refers to that space, and a *remote* operation otherwise.

**Output Structure.** The output program (Listing 5) retains the same structure as the input program. The global section of the initial program is extended with auxiliary data structures, such as the map `sp` from space identifiers to concrete references to space objects (line 12) and the map `uri` from space objects to space identifiers (line 13) (used for example in Listing 5). An additional package with the definitions of the extended tuple manipulation routines (Listings 1, 2, etc.), is added to the import section at the beginning of the output program (line 3).

In the process definition functions $p_1, \ldots, p_n$ every call to a goSpace routine is transformed into a call to the corresponding extended primitive (Sect. 3) to achieve replication accordingly. For `MPut` operations, the set of target spaces for replication is added as an argument (e.g., cf. line 21 of Listing 4 and Listing 5). Each such set is over-approximated by the procedure described in the following section. Any other access operation, such as `GetP`, `Query` or `QueryP` (lines 31 and 40) is instead changed to always refer to the local space.

**Overapproximating the Sets of Target Spaces.** It is worth noticing that the extended tuple manipulation routines (Sect. 3) are independent from the specific technique used for reducing the set of target spaces for data replication. In the following, we simply describe a lightweight static analysis technique for overapproximating such sets of target spaces. The goal of our static analysis procedure is to work out a refined set of target spaces, i.e., the *data-access tables*, for replicating the tuples while preserving strong consistency.

Let us consider a tuple $t$ and a process $p_i$ performing an output operation of $t$ into a specific space $s_j$. The key idea of our approach consists in determining the set of processes $P' \subseteq P$ that can potentially perform a subsequent read operation on that tuple. We identify such processes by looking at the patterns used in the input operations within the corresponding definition functions, approximating the actual pattern matching mechanism of the normal tuple manipulation routines. In practice, given on the one hand an output operation and on the other hand an input operation, we check for a potential match between the tuple being

stored and the given search tuple or template. We repeat this for every process except $p_i$ and for every input operation in the corresponding process definition function, obtaining $P'$ by progressively excluding from $P$ any process that is *definitely* not involved in an input operation matching the tuple $t$. Eventually, the data-access table for replicating $t$ will be the set $S' \subseteq S$ induced by $P'$ on $S$.

For simplicity, let us assume that a field of a tuple $t$ given as input to an `MPut` operation can be either a constant or a variable identifier, while a field of a pattern $p$ taken by `MGetP` or `MQueryP` can be either a constant value or a formal field, namely a typed variable reference. Due to space limitations, we only give an informal description.

The matching mechanism initially compares the number of fields of $t$ and $p$: if they are different, then certainly there is no match; otherwise, there is still the possibility for $t$ and $p$ to match. The matching is then refined based on the actual fields of the tuple and the pattern, ignoring any formal fields or placeholders. A difference of any actual field at the same position of $t$ and $p$ indicates a mismatch. The matching is eventually refined again by taking into account the type of the formal fields of $p$. A type mismatch between an actual field of $t$ (either a constant or a variable) and the corresponding formal field of $p$ means no match.

It is worth to notice that combining the matching mechanism described above with the replica-aware routines from Section 3 preserves consistency, because

1. the matching algorithm only avoids replication for those spaces where a tuple is definitely never going to be accessed (i.e., no matching input operations for that tuple exist in the whole process definition function corresponding to that space), and therefore safely over-approximates the set of target spaces for replication, and
2. the tracking mechanism embedded within the replica-aware tuple manipulation routines guarantees that, when one copy of a tuple is removed, all its replicas are atomically removed as well.

**Program Transformation.** We can now transform the initial program to automatically achieve replication, by converting all the operations to goSpace into calls to the new RepligoSpaces routines introduced in Section 3, and using as target locations for write operations the sets computed by the matching technique above.

The program transformation procedure takes as input the initial program and the data-access tables built via the static analysis pass described above, and generates a program where each tuple is replicated as indicated by the corresponding access lists. This can be done by parsing the input program into an abstract syntax tree, and then performing a series of pattern-based transformations on (parts of) this tree.

To see how pattern-based syntax tree transformations work, let us now consider the function call at line 21 of Listing 4, where `process1` performs a `Put` operation of the tuple `("A",10)` into the local tuple space `s1`. This fragment of code will trigger transformation because the referenced object (`s1`) is a tuple space (which is detected via a symbol table lookup) and the `Put` method is

among the relevant ones. In the syntax tree, the corresponding subtree for the whole expression is therefore changed into a call to `MPut` (see Listing 1 from Section 3); new child nodes are appended to the function call node in the syntax tree for the extra parameters as shown in Listing 5. Un-parsing the syntax tree modified in this way will produce the transformed program.

**Example.** We now show how the procedure described above leads from the program of Listing 4 to the one in Listing 5.

```
1  import (
2    . "github.com/pspaces/gospace"
3    ...
4  )
5
6
7
8
9
10 func main() {
11   s1 := NewSpace("tcp://host:123/s1")
12   go Process1(&s1)
13   ...
14 }
15
16
17
18 func process1() {
19   var choice bool
20   ...
21   s1.Put("A",10)
22   ...
23   s1.Put(choice,10)
24   ...
25 }
26
27 func process2() {
28   ...
29   if check {
30     var key int
31     s1.GetP("A",&key)
32   }
33   ...
34 }
35
36 func process3() {
37   ...
38   var choice bool
39   var desc string
40   s1.GetP(&desc,&choice)
41   ...
42 }
43
44 ...
```

Listing 4: Initial program

```
1  import (
2    . "github.com/pspaces/gospace"
3    . "repligospaces"
4    ...
5  )
6
7  var uri = make(map[space]string)
8  var sp = make(map[string]*Space)
9
10 func main() {
11   s1 := NewSpace("tcp://host:123/s1")
12   sp["tcp://localhost:123/s1"] = &s1
13   uri[s1] = "tcp://host:123/s1"
14   go Process1()
15   ...
16 }
17
18 func process1() {
19   var choice bool
20   ...
21   MPut("A",10,targets0)
22   ...
23   MPut(choice,10,targets1)
24   ...
25 }
26
27 func process2() {
28   ...
29   if check {
30     var key int
31     MGetP("A",&key,uri[s2])
32   }
33   ...
34 }
35
36 func process3() {
37   ...
38   var choice bool
39   var desc string
40   MGetP(&desc,&choice,uri[s3])
41   ...
42 }
43
44 ...
```

Listing 5: Transformed program

The graphs that represent the data distribution for the initial program (see Listing 4) and the transformed program (see Listing 5) are shown in Figures 1a and 1c, respectively. Figure 1b represents universal replication and is included for comparison. In the figures, arrows from left to right indicate write operations; arrows from right to left read operations.

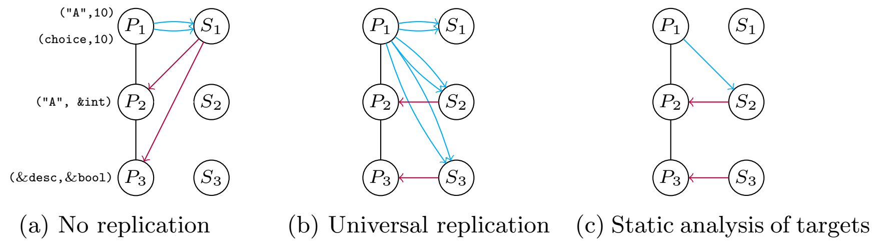
Let us consider the tuple `("A",10)` stored by `process1` at line 21. The `GetP` operation at line 31 `process2` uses as the pattern a string constant and a formal field of integer type. Therefore the local tuple space `s2` is included in the set

of spaces for replication of `("A",10)`. Note that the analysis is control-flow insensitive, as the branch condition at line 29 is ignored.

Now let us consider `process3`. The size of the pattern given at line 40 and of the tuple `("A",10)` under consideration match. The types of the last fields respectively of the tuple and of the pattern do not match (`bool` vs `integer`). Therefore, the tuple is not replicated to `s3`.

Let us now focus on the tuple `(choice,10)` stored by `process1` at line 23. The type of the first field of the tuple is known, but its value depends on previous computations. The pattern used in the input operation in `process2` at line 31 does not match this type. The tuple is thus not replicated at `s2` or at `s3`.

Indeed in the transformed program (Fig. 1c) the only replicated tuple is `("A",10)`, which is replicated to $s2$ as it can potentially be accessed by `process2`. Note that there is no need to store this tuple to `s1`, as no subsequent matching read operation within `process1` occurs. It is worth to observe that in general this program transformation does not depend on the specific static analysis technique to work out the set of target locations (i.e., shown as `targets0` and `targets1` in Fig. 1c).




(a) No replication (b) Universal replication (c) Static analysis of targets

Fig. 1: Example replication strategies

## 5 Implementation and Experimental Evaluation

In this section, we describe the implementation of our prototype and provide an experimental evaluation on our technique.

**Overall Workflow and Technical Details.** Having defined the structure of the input program $P$ and of the output program $P'$ (Sect. 4), we can now describe more precisely the overall workflow of our approach (Figure 2).

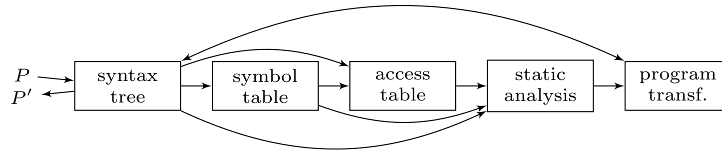



Fig. 2: Static analysis and source transformation for automated replication

The program $P$ is initially parsed to generate an *abstract syntax tree*. The syntax tree is traversed to generate the *symbol table*. During this process we start visiting the body of the process definition functions, and then of the nested blocks recursively. As we go along, we assign blocks unique identifiers, so that as soon a new variable declaration occurs in the syntax tree, that variable is added to the set of symbols for the current block; the *type* of the variable is also extracted from the syntax tree and stored in the symbol table.

The next step consists in building the *access table* by extracting information from the syntax tree and the symbol table. In particular, we visit the syntax tree again to detect all the operations on tuples. At the same time, we search the symbol table to figure out the type of fields for each tuple occurring as an argument for any of such operations. Eventually, we obtain for each tuple operation the actual tuple along with the type of each field of the tuple.

We can now perform *static analysis* by visiting the syntax tree a third time and combining information from the symbol table and the access table in order to overapproximate the set of target spaces for replication.

A program transformation module alters the syntax tree to replace the tuple manipulation operations occurring in the initial program with their replica-aware counterparts, where the set of target spaces for each `Put` operation has been determined by the static analyser. We finally obtain the modified program $P'$ by un-parsing the modified syntax tree. The code of our open-source prototype is available at `https://github.com/Uwimbabazi/Replication/releases/tag/v1`.

**Experimental Evaluation.** Let us now consider a distributed system composed of $n$ computational nodes, each executing a separate program, and interacting through a decentralised data store of capacity $m$ elements. Following a similar schema to those used in distributed lookup protocols (e.g., Chord [26]), memory entries are represented as key-value pairs, with a partitioned address space among the nodes. Each node is responsible for storing $m/n$ memory entries. A node reads from and writes to either its own local memory, or that of another node, depending on the source or target memory address. Each node performs $o$ operations, with $p$ denoting the expected percentage of write operations.

For such a system, one might consider adopting a replication schema in the attempt to reduce non-local access (at the cost of additional local storage, plus some overhead for replication to non-local storage). To experiment with this idea, we model the nodes as separate processes, and the local memory of a node as the local tuple space of the corresponding process, with tuples $(address, value)$ representing values held at different memory addresses. The structure of the program follows that of Listing 4. For simplicity, we assume that all read operations are `QueryP`. Write operations are of course `Put`.

To evaluate the effect of replication on the system, we conduct the following experiments. We initially considered a system with $\{n=4, m=32\}$, then one with twice as much memory $\{n=4, m=64\}$, then a larger system $\{n=32, m=256\}$, and eventually a larger system with twice as much memory $\{n=32, m=512\}$. For all these systems, we set $o=16$, while varying $p$ in $\{10, 20, \ldots, 90\}$. For each combination of the chosen values for $n$, $m$, and $p$, we generate 10 test cases (i.e.,

programs) with random data access patterns. We run each test case 10 times, leading to rounds of 100 runs each. We repeat each such round twice: once on the initial program, and once on the replicated program obtained with our tool (Sect. 4), for an overall number of 1800 runs for each of the four considered systems. We eventually compare the average number of local and remote read and write operations. The experiments were conducted on a standard laptop. The experiments are summarised in Figures 3a, 3b, 3c and 3d where we compare the average count of non-local memory accesses with and without replication, for each configuration.

Without replication, both read and write access can be non-local, depending on the address being accessed. With replication, read operations are always local, because tuples are always replicated where they can be potentially accessed. However, this comes at the cost of extra non-local write access to replicate the data. If the system tends to read from the shared memory more often than writing to it, our approach can be beneficial. In Figures 3a–3d, the number of non-local accesses with replication is maximised when the read and write operations occur with the same probability. Replication seems to be more beneficial with larger memory size (from 32 to 64, or from 256 to 512).

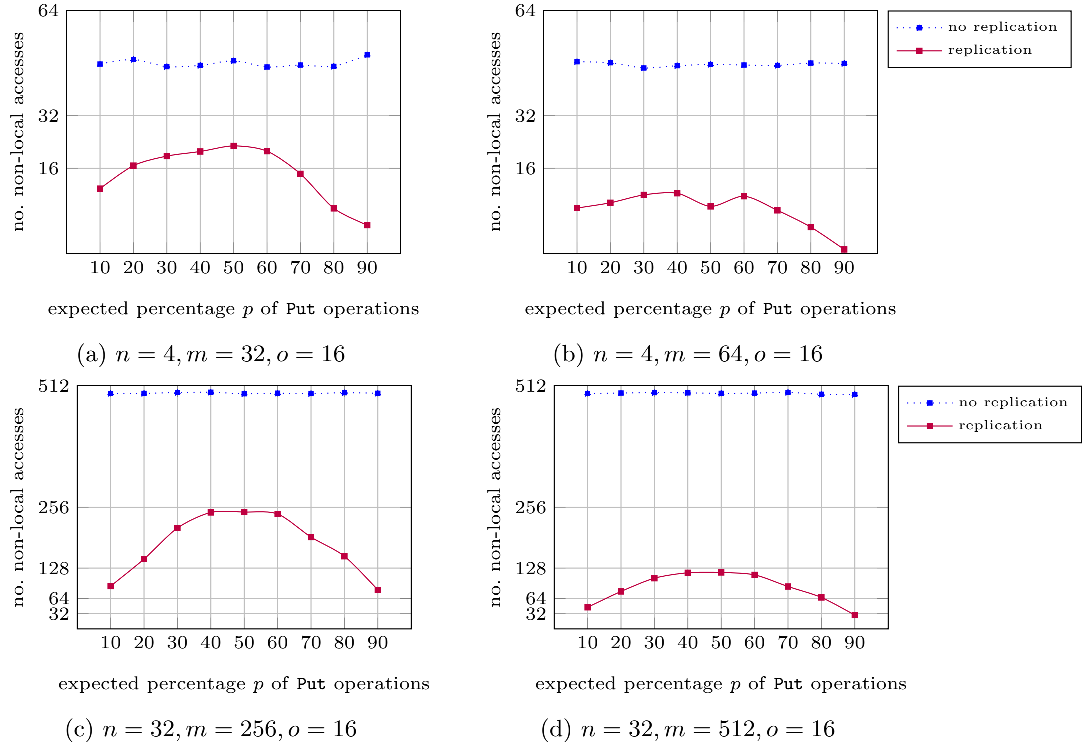



Fig. 3: Non-local read or write operations with and without replication

## 6 Related work

In addition to the pSpaces family of implementations, tuple space systems have been proposed in different programming languages. Java implementations include Klava [4] for Klaim, and jRESP[4] for SCEL [9]. jSpace, the implementation of pSpaces in Java, was initially based on a fork of jRESP. We chose to work on top of pSpaces because it is actively maintained.

Besides RepliKlaim [1], tuple replication has also been implemented in X10 [2], a general-purpose language for large-scale distributed systems [25]. An extension of Lime, a distributed tuple space for mobile ad-hoc networks, relies on replication to increase availability [21]. In all these approaches, the responsibility to control replication is left to the programmer.

In the attempt to increase scalability, a hierarchical tuple space model with partial replication has been proposed in [6]. Spatial distribution of tuples is a rather different approach to ours where tuples contain both content and replication rules [20]; in this model the propagation of the tuples is asynchronous and thus strong consistency has to be explicitly programmed. Alternative distribution mechanisms for tuple spaces based on the concept of ghost tuples have been proposed in [10], where it is the system that may decide not to eliminate tuples for using them later.

Tuple-based coordination models focusing on fault tolerance have been proposed [3]. Consistency models for replicated data are covered in [12]. Dynamic replication has been considered in [5,24].

Several program transformation frameworks for different languages are available. A popular framework for C and C++ is ROSE [23], where the syntax tree can be directly modified and then un-parsed to obtain the modified program. Another transformation framework for C and C++, widely adopted in software verification, is the Clang compiler framework [19]. As Clang does not allow to modify the abstract syntax tree, program transformation is obtained by directly altering the relevant fragments of the initial source code. In our approach the model of the system to be replicated is expressed as a Go program, and standard Go packages support either of the above techniques. For this reason, among the available implementations of pSpaces, we found it particularly convenient to focus on the Go implementation.

Static verification of concurrent Go programs for bounded liveness and safety has been considered in [17,18]. Bounded analysis of concurrent programs for safe replication has been proposed in [14].

## 7 Conclusion and Future Work

We have presented RepligoSpaces, a replica-aware extension of goSpace, an implementation of Klaim (pSpaces) in Go. We have also discussed how RepligoSpaces fits within a fully-mechanisable procedure for automated replication of programs

[4] http://jresp.sourceforge.net/

over tuple spaces that relies on combining static analysis and program transformation. A lightweight static analysis pass on the initial program computes the sets of target spaces for replication, so that the standard tuple manipulation routines can be replaced by equivalent replica-aware versions. The combined approach preserves strong consistency, thanks to a tracking mechanism embedded in the tuple manipulation routines and to the fact that the set of target spaces is safely over-approximated.

In the near future, we plan to consider further scenarios where replication has successfully been applied to other contexts, such as database systems [11], and cloud computing [16], as well as other consistency models [12,27]. We also plan further work on the static analysis procedure to improve the accuracy of the over-approximation in the presence of formal fields, i.e., placeholders in the pattern or variables in the tuple to be stored. We plan to initially focus on simple and efficient techniques to complement the existing analysis with limited effort. To name a few, constant propagation [28] can reduce the overall number of formal fields or restrict the possible values of a given formal field collecting them over different branching paths; abstract interpretation [7] can overapproximate the interval ranges of the integer variables used as formal fields.

We consider our contribution to be a first step towards developing an integrated framework to experiment with data replication in distributed systems with tuple spaces. We aim at providing different analyses and consistency models to choose from, in order to appreciate the effect of different consistency levels on many interesting classes of more or less complex distributed systems where data replication is heavily used. This would allow, for instance, to evaluate under different consistency levels many interesting classes of systems, such as models of hardware cache or complex interaction models, where replication is heavily used and performance is particularly sensitive to variations in the data distribution.

**Acknowledgements.** Work partially funded by MIUR project PRIN 2017FTXR7S IT MATTERS (Methods and Tools for Trustworthy Smart Systems).

## References

1. Andrić, M., De Nicola, R., Lluch-Lafuente, A.: Replica-based high-performance tuple space computing. In: COORDINATION. pp. 3–18. LNCS, Springer (2015)
2. Andrić, M., De Nicola, R., Lluch-Lafuente, A.: Replicating data for better performances in X10. In: Semantics, Logics, and Calculi. LNCS, vol. 9560, pp. 236–251. Springer (2016)
3. Bessani, A.N., Alchieri, E.A.P., Correia, M., da Silva Fraga, J.: DepSpace: a byzantine fault-tolerant coordination service. In: EuroSys. pp. 163–176. ACM (2008)
4. Bettini, L., De Nicola, R., Pugliese, R.: Klava: a Java package for distributed and mobile applications. Softw. Pract. Exp. **32**(14), 1365–1394 (2002)
5. Casadei, M., Viroli, M., Gardelli, L.: On the collective sort problem for distributed tuple spaces. Sci. Comput. Program. **74**(9), 702–722 (2009)
6. Corradi, A., Leonardi, L., Zambonelli, F.: Distributed tuple spaces in highly parallel systems. Tech. rep., DEISLIA-96-005, UNIBO(Italy) (1996)

7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. pp. 84–96. ACM Press (1978)
8. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. **24**(5), 315–330 (1998)
9. De Nicola, R., Latella, D., Lluch-Lafuente, A., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL language: Design, implementation, verification. In: The ASCENS Approach, LNCS, vol. 8998, pp. 3–71. Springer (2015)
10. De Nicola, R., Pugliese, R., Rowstron, A.I.T.: Proving the correctness of optimising destructive and non-destructive reads over tuple spaces. In: COORDINATION. LNCS, vol. 1906, pp. 66–80. Springer (2000)
11. Elnikety, S., Dropsho, S.G., Zwaenepoel, W.: Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In: EuroSys. pp. 399–412. ACM (2007)
12. Fekete, A.D., Ramamritham, K.: Consistency models for replicated data. In: Replication, pp. 1–17. Springer (2010)
13. Gelernter, D.: Generative communication in Linda. ACM (TOPLAS) **7**(1), 80–112 (1985)
14. Kaki, G., Earanky, K., Sivaramakrishnan, K.C., Jagannathan, S.: Safe replication through bounded concurrency verification. Proc. ACM Program. Lang. **2**(OOPSLA), 164:1–164:27 (2018)
15. Kaminskas, L., Lluch-Lafuente, A.: Aggregation policies for tuple spaces. In: COORDINATION. LNCS, vol. 10852, pp. 181–199. Springer (2018)
16. Karandikar, R.R., Gudadhe, M.B.: Comparative analysis of dynamic replication strategies in cloud. IJCA. **TACIT2016**(1), pp. 26–32 (2016)
17. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: liveness and safety for channel-based programming. In: POPL. pp. 748–761. ACM (2017)
18. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in Go using behavioural types. In: ICSE. pp. 1137–1148. ACM (2018)
19. Lattner, C.: Llvm and Clang: Next generation compiler technology. In: The BSD Conference (2008)
20. Mamei, M., Zambonelli, F., Leonardi, L.: *T*uples *O*n *T*he *A*ir: a middleware for context-aware multi-agent systems. In: WOA. pp. 108–116. PEB (2002)
21. Murphy, A.L., Picco, G.P.: Using Lime to support replication for availability in mobile ad hoc networks. In: COORDINATION. pp. 194–211. Springer (2006)
22. Pike, R.: Go at google. In: SPLASH. pp. 5–6. ACM (2012)
23. Quinlan, D., Liao, C.: The ROSE Source-to-Source Compiler Infrastructure. In: Cetus Users and Compiler Infrastructure Workshop, in conj. with PACT (2011)
24. Russello, G., Chaudron, M.R.V., van Steen, M.: Dynamically adapting tuple replication for managing availability in a shared data space. In: COORDINATION. LNCS, vol. 3454, pp. 109–124. Springer (2005)
25. Saraswat, V.A., Jagadeesan, R.: Concurrent clustered programming. In: CONCUR. LNCS, vol. 3653, pp. 353–367. Springer (2005)
26. Stoica, I., Morris, R.T., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM. pp. 149–160. ACM (2001)
27. Terry, D.: Replicated data consistency explained through baseball. Commun. ACM **56**(12), 82–89 (2013)
28. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. **13**(2), 181–210 (1991)