Automated Replication of Tuple Spaces via Static Analysis*

Rocco De Nicola[a], Luca Di Stefano[b,c,*], Omar Inverso[d], Aline Uwimbabazi[d]

[a] *IMT School for Advanced Studies, Lucca, Italy*
[b] *Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP,† LIG, Grenoble, France*
[c] *University of Gothenburg, Gothenburg, Sweden*
[d] *Gran Sasso Science Institute (GSSI), L'Aquila, Italy*

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.

Keywords: Concurrency, Tuple spaces, Replication, Program transformation, Static analysis, Golang, Linda.

---

*Work partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems), and by ERC consolidator grant no. 772459 *D-SynMA* (Distributed Synthesis: from Single to Multiple Agents).

*Corresponding author.

Email address: `luca.di.stefano@gu.se` (Luca Di Stefano)

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 [1], 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. Introducing multiple tuple spaces is a natural way to extend Linda [2]: this concept has been further explored in several related languages such as X10 [3] and Klaim [4]. 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 one copy of a given tuple is consumed, how are the remaining copies to be affected?

RepliKlaim [5] 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 Figure 1. The system of interest is modelled as a concurrent Go [6] program. The behaviour of each component of the system is defined by a separate thread of the program.

†Institute of Engineering Univ. Grenoble Alpes

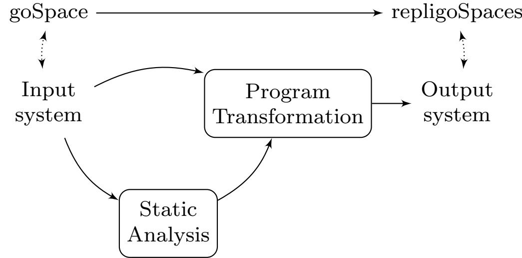


Figure 1: An overview of our proposed approach.

Coordination takes place via goSpace [7], a recent Go library which is part of a family of Klaim implementations called pSpaces.[1]

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/update the replicated data at need.

At this point one could immediately obtain full consistency by 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 relatively effortlessly. At the same time, alternative replication policies and 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

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

tuple spaces. We would like to emphasise that our goals do not yet include an efficient implementation of RepligoSpaces. Rather, we aim at providing a methodology and an experimental framework to analyse the costs and benefits of different replication schemes, supporting system designers in choosing the most appropriate solution for their application. The case studies and the proposed replication schemata are reasonably simple as their main purpose is to show the applicability of our approach.

This paper extends [8] in different ways. To show the extensibility of our methodology, we gradually build more sophisticated replication techniques on top of the basic replication mechanism proposed in [8]; in particular, we add the possibility of limiting the overall number of elements in a tuple space as well as different well-known replacement policies, which allows to model more realistic scenarios with limited local memory. We expand the experimental evaluation of the case study considered in [8] to include the alternative replication mechanisms. We also sketch another case study, a network of work-stealing cooperating service providers, and discuss how to use our experimental methodology to assess the effects of the different replacement policies; in particular, we show how to use our technique to evaluate the tradeoff between local memory size and quality of service.

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 procedure 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. For conciseness, we only describe the language features used in the rest of the paper. We refer the interested reader to the cited references for further details.

Klaim. Klaim (Kernel Language for Agents Interaction and Mobility) [4] 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 [4, 1], that enables asynchronous communication via a set of operations used 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* or *patterns*. `("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 have 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 ℓ. The $\texttt{read}(T)@\ell$ operation selects via pattern matching one of the tuples at locality ℓ that matches template T; this operation blocks until a tuple matching T is found at ℓ. 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 $\texttt{eval(-)}@\ell$. We do not consider process spawning here as it is not currently implemented in RepliKlaim and pSpaces, and it is not relevant for our analysis.

RepliKlaim. RepliKlaim [5] 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 $\texttt{outRK}(t, \ell_1 \ldots \ell_n)$ 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,ℓ`)` reads a tuple space. It uses a pattern T to retrieve a matching tuple (if any) from locality ℓ, 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 `readRK`$_{nb}$`(`T,ℓ`)` is similar to `readRK(`T,ℓ`)` except that it is non-blocking, and returns an empty tuple when no matching tuple is found.

The input operation `inRK(`T,ℓ`)` retrieves a tuple matching the pattern T at ℓ and removes all replicas of that tuple. This removal is assumed to happen atomically, meaning that no process may access any tuple space until all replicas have been removed. Operation `inRK`$_w$`(`T,ℓ`)` also removes a tuple matching T along with all of its replicas, but the removal of replicas happens asynchronously. That is, the removal is interleaved with the operations of other processes, and therefore some tuple spaces may end up in an *inconsistent* state in the sense that they store a replica of an already-deleted tuple. The other operations seen so far never introduce such states, and thus we say that they preserve *strong* consistency. Instead, `inRK`$_w$ only preserves *weak* consistency, meaning that it only guarantees that all tuple spaces will be consistent after it completes. In this paper we focus on replication under strong consistency.

pSpaces. The pSpaces framework 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 [7].

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 simi-

lar 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 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. 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 $s_1 \ldots s_n$ via an `MPut(`$t, s_1 \ldots s_n$`)` 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. Lastly, operations `MGetP(`s,T`)` and `MGet(`s,T`)` return a tuple matching T, and remove it from space s and from any other space where it was previously replicated. The latter blocks until such a tuple is found, while the former simply returns an empty tuple if T cannot be matched.

We now provide a first possible implementation of the replica-aware tuple manipulation routines. We then show how to extend this implementation to model slightly more involved replication mechanisms. 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).

3.1. 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 centralised tracking of the storage locations [5] 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.

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 23). Note that the operation `MQuery` is similar to `MQueryP`, except that it

```
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     // Return the matching tuple without the last field
15     if e == nil {
16         var u = CreateTuple(t1.Fields[:len(t1.Fields)-1]...)
17         Sp.mux.Unlock()
18         return u
19     }
20
21     // Return an empty tuple when no tuple is available
22     Sp.mux.Unlock()
23     return CreateTuple()
24 }
```

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

blocks until a tuple is found, and does not lock access to the tuple space while waiting, which would result in a deadlock.

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

```
 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(t1.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()   // Return an empty tuple when no tuple is available
39 }
```

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

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). Similarly to what has been said before about `MQuery` and `MQueryP`, we obtain the blocking operation `MGet` simply by blocking until a matching tuple t is found. When this happens, we remove t and all of its replicas. During the removal phase, we forbid processes from performing other operations on the tuple spaces so as to preserve strong consistency.

3.2. Limited Memory and Tuple Replacement Policies

So far we have considered arbitrarily large tuple spaces (Sect. 3.1). However, this is too simplistic for many systems of interest such as multi-core processors with local caches, wireless sensor networks with resource-constrained

nodes, and so on. In order to experiment more realistically with replication in the presence of limited local memory, we introduce *memory-limited* tuple spaces, which can store only up to a given number of tuples. This obviously also begs the question of what should be done when a process attempts to store a tuple into a space that is already full. We thus provide different *replacement policies*, where transparently a tuple is selected according to a set of rules and removed from an otherwise full space (possibly along with any replicas) to make room for the new tuple being inserted.

A possible way to extend the `MPut` operation of Listing 1 is shown in Listing 4. We iterate through every space in `S` (lines 6–14) to store a replica of tuple `t` into every space in the set of targets `S`. Differently from Listing 1, for every such space `s` we check (line 8) if the memory limit is non-zero and `s` is already full (note the variable `ReplLimit` that we add to the `RepliSpace` object to represent the maximum number of tuples that each space can hold, the default value 0 meaning unlimited). If so, we must first remove one of the existing tuples, by calling `evictTuple`, which selects and removes a tuple from a full space, according to the chosen policy (more details on this functions later on). This guarantees that, when the execution reaches line 12, the number of tuples in `s` is strictly lower than the limit. Notice that we still aim at preserving strong consistency, therefore any replica of the tuple selected for replacement is removed as well. At this point, we store the new tuple in `s`, and call an internal function `afterPut()` that updates the space's `counter` and perform additional bookkeeping required by the policy (lines 12–13). Notice that the creation timestamp `now` is computed only once, before the loop (line 4), to guarantee that all replicas of the tuple have the same timestamp. More specifically, time information is recorded in a `TimeRecord` object storing the actual timestamp as well as the tuple being inserted. This is only needed for internal bookkeeping. When these operations have been performed for all target spaces, we can finish by returning the inserted tuple (as seen in Listing 1).

Replacement policies are sets of rules that guide the selection of the tuple to be removed to make room for a new one. In RepligoSpace we consider the following policies:

– *Random replacement* (RR): this policy simply removes one random tuple from the tuple space. When prototyping more sophisticated replication mechanisms, it may be used as a baseline against which to compare. If the access patterns are quite random, RR may be just as effec-

```
1 func MPut(t Tuple, Sp Replispace, S []string) Tuple {
2     ... // Lock mutex, create tuple t' = (t, S)
3
4     now := TimeRecord{tuple:t', time:time.Now()} // Get timestamp for t'
5
6     for i := 0; i < len(S); i++ {
7         s := *Sp.Sp[S[i]]
8         if Sp.ReplLimit > 0 && counter[*s] == Sp.ReplLimit {
9             evictTuple(Sp, *s)      // Remove a tuple, based on the chosen policy
10        }
11
12        s.Put(t1.Fields...)
13        afterPut(t1, Sp, *s, now)     // Update counter etc., depending on policy
14    }
15
16    ... // Unlock mutex, return t'
17 }
```

Listing 4: The `MPut` operation with cache replacement.

tive as the other policies while enjoying a much lower computational cost [9].

- *First in, first out* (FIFO): this policy associates to each tuple a timestamp that records when it has been stored in the `RepliSpace`, and removes the tuple with the lowest timestamp when space is needed. Notice that our implementation guarantees that replicas of the same tuple have the same timestamp.

- *Least recently used* (LRU): similarly to FIFO, this policy removes the tuple with the lowest associated timestamp. The timestamp initially represents the moment when the tuple was stored, but is updated every time the tuple is returned as the result of a `Query()` or `QueryP()` operation. Thus, different replicas of the same tuple may end up having different timestamps.

Both the memory limit (`ReplLimit`) and the chosen policy (`ReplPolicy`) of the replicated space may be changed at runtime: thus, the current prototype allows in principle to handle dynamic memory limits and policies, which however is beyond the scope of this work.

A replacement policy relies on four functions: `afterPut()`, `afterQuery()`, and `afterGet()` that are called from within the corresponding primitive operations and perform the required bookkeeping after a tuple is successfully inserted, retrieved, or withdrawn from a space, respectively; and `evictTuple()` that selects and removes a tuple from a full space, as mentioned earlier.

In Listing 5 we show a simplified version of these functions for the LRU policy. Essentially, for every function, the logic for each policy is implemented as a case within a `switch` statement. This means that new policies can be plugged in by simply adding an appropriate `case` to each function. The only policy-independent operations are those that allow counting the number of tuples in each tuple space: namely, we increase `counter` within `afterPut()` (line 2) and decrease it within `afterGet()` (line 18).

In the case of the LRU policy, we maintain a dictionary `lastAccessTime` that tracks the time when the tuple was last retrieved. Whenever we insert a tuple `t` into a space `s`, we initialize `lastAccessTime[s][t]` to the creation time of the tuple, `now`. This is done in the `afterPut()` function (line 4). Each time a tuple is retrieved via a `Query` operation, we capture the current timestamp and update `lastAccessTime[s][t]` accordingly (line 12). Moreover, when a tuple is withdrawn, we delete its timestamp from the dictionary (line 21). Lastly, whenever `evictTuple()` is invoked to make room for a new tuple in a full space `s`, we call a function `minByTime()` that iterates through `lastAccessTime[s]` and returns the tuple with the lowest associated timestamp (line 30). Then, we remove this tuple (along with all its replicas) by means of an `MGetP` operation (line 35).

3.3. Reactive Replication

In both variants of the replica-aware primitives presented so far (Sect. 3.1 and 3.2) a tuple is immediately replicated to all target spaces that might need it in the future. Provided that the target spaces are soundly over-approximated, this yields strong consistency. However, the actual costs of overhead will heavily depend on the specific application. One might thus want to evaluate alternative replication mechanisms where the number of replicas is limited, but occasionally remote lookups may be needed. We thus propose another variation named *reactive* replication, as opposed to the *proactive* replication seen so far. Under the reactive scheme, we still rely on an (over-approximated) set of target spaces for each tuple in a store operation, but only store the tuple into one of the target spaces. When a process attempts at querying the tuple later, it may have to perform a remote lookup if the tuple has not been replicated to its local space yet. If the lookup succeeds, the found tuple is then replicated to the space of the querying process, to avoid further remote lookups.

We have implemented reactive replication as an optional transformation in RepligoSpaces. Listing 6 gives an overview of the implementation. Notice

```
func afterPut(t1 Tuple, Sp Replispace, s Space, now TimeRecord) {
  counter[s] += 1
  switch Sp.ReplPolicy {
     case "lru": lastAccessTime[s][t1.String()] = now
     ... // other policies
  }
}

func afterQuery(t1 Tuple, Sp Replispace, s Space) {
  switch Sp.ReplPolicy {
     case "lru":
        lastAccessTime[s][t1.String()] = TimeRecord{tuple:t1, time:time.Now()}
     ... // other policies
  }
}

func afterGet(t1 Tuple, Sp Replispace, s Space, now TimeRecord) {
  counter[s] -= 1

  switch Sp.ReplPolicy {
     case "lru": delete(lastAccessTime[s], t1.String())
     ... // other policies
  }
}

func evictTuple(Sp Replispace, s Space) {
  var t Tuple

  switch Sp.ReplPolicy {
     case "lru": t = MinByTime(lastAccessTime, s)
     ... // other policies
  }

  // Create template p = {t,S}
  MGetP(p, Sp, s) // Remove {t,S} along with its replicas
}
```

Listing 5: Implementation of the LRU policy.

that the API is identical to that of Listings 1, 2, and 3, to allow users of RepligoSpaces to effortlessly switch between the two schemes.

The `MPut` operation is implemented similarly to Listing 1, except that the extended tuple (`t,S`) is only stored into the first space of `S` (line 5). Now, assume that a given process attempts to retrieve this tuple from its local space by performing an `MQuery` or `MQueryP` operation. Under the proactive scheme and assuming a sound static analysis this query would surely succeed, as the tuple would already have been replicated to every target space (and the local space would be one of the targets, by soundness of the static analysis). With reactive replication, instead, the local lookup (line 14) may fail. If so, we initiate a sequence of remote lookups across the whole replicated space, until we find a matching tuple. Then, we replicate the tuple locally to prevent further remote lookups, and return the found tuple (lines 18–29). Finally, if the template passed to `MQueryP` does not match any tuple, the operation

returns an empty tuple (line 31).

The modified `MGetP` procedure is almost the same as `MQueryP`. We perform lookups (first locally, then remotely) to find a tuple that matches the given template (lines 38–44). If one such tuple is found, we extract from it the set of spaces `S` where it may have been replicated. Then, we iterate through `S` and remove all replicas of the tuple (lines 46–48). Obviously, no replication happens here, since the found tuple is going to be removed from every space anyway. As usual, if no matching tuple is found, an empty tuple is returned instead.

Notice that this replication scheme still preserves strong consistency, and makes `Put` operations lighter: initially, every tuple is stored in a single space, whereas the proactive scheme would always entail one store operation for every target space. This comes at a cost of potentially heavier queries, as the required tuple may not be locally available at first.

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 7) 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

```
1  func MPut(t Tuple, Sp Replispace, S []string) Tuple {
2       ... // Lock mutex, create tuple p' = (t, S)
3
4       if len(S) > 0 {
5           Sp.Sp[S[0]].Put(t1.Fields...) )
6       }
7
8       ... // Unlock mutex, return t'
9  }
10
11 func MQueryP(p Tuple, Sp Replispace, s Space) Tuple {
12      ... // Lock mutex, create template p1 = {p, S}
13
14      t1, e := s.QueryP(p1.Fields...)
15
16      if e == nil {
17          ... // Tuple found locally: unlock mutex, return t1
18      } else { // Remote lookup
19          for _, remote := range Sp.Sp {
20              t2, err := remote.QueryP(p1.Fields...)
21
22              if err == nil {
23                  // Replicate tuple locally and return it
24                  s.Put(t2)
25                  Sp.mux.Unlock()
26                  return t2
27              }
28          }
29      }
30
31      ... // No tuple matches p: Unlock mutex, return empty tuple
32 }
33
34 func MGetP(p Tuple, Sp Replispace, s Space) Tuple {
35      ... // Lock mutex, create tuple p1 = (p, S)
36
37      var S []string = make([]string, 0)
38      t1, e := s.QueryP(p1.Fields...)
39
40      if e == nil {    // Local lookup succeeds
41          S = ...      // Extract S from t1
42      } else {         // Remote lookup on every space of Sp
43          ...          // If a matching tuple t1 is found, extract S from it
44      }
45
46      if len(S) > 0 {
47          ... // Remove t1 from every space in S
48      }
49
50      ... // Unlock mutex, return t1 (or empty tuple)
51 }
```

Listing 6: Primitives for reactive replication.

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 8) 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 8). 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 7 and Listing 8). 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. This assumption does not affect the generality of our approach. Fields that match the result of an arbitrary expression may be handled by first storing the result in a dedicated variable and then using a reference to that variable in the tuple. What matters here is the type of the expression (and therefore of the dedicated variable), which may be statically determined in languages with strong, static typing such as Go.

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 noticing 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;
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 7, 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 8. Unparsing 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 7 to the one in Listing 8. The graphs that represent the data distribution for the initial program (see Listing 7) and the transformed program (see Listing 8) are shown in Figures 2a and 2c, respectively. Figure 2b 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. Actual tuple fields, i.e., literals or variable names, are typeset normally, while placeholders, i.e., type names, are typeset in italics.

Let us first 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

```
1  import (
2    . "github.com/pspaces/gospace"
3    ...
4  )
5
6
7
8
9
10 func main() {
11   s1 := NewSpace("tcp://host:12/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 7: 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:12/s1")
12   sp["tcp://localhost:12/s1"] = &s1
13   uri[s1] = "tcp://host:12/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 ...
```

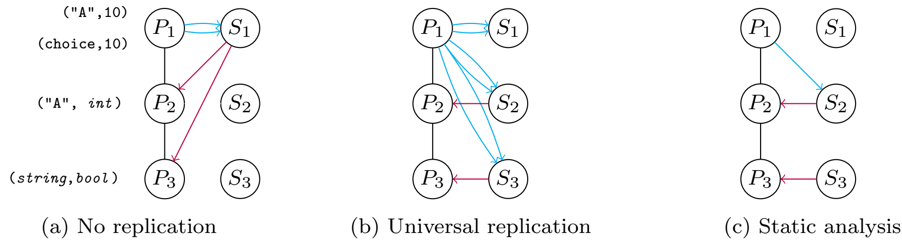
Listing 8: Transformed program

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`, nor at `s3` (again, because the pattern at line 40 does not match).

Indeed, in the transformed program (Figure 2c) 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 (e.g., the set containing `s2` for tuple `("A",10)`, called `targets0` in Listing 8).



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

Figure 2: 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 3).

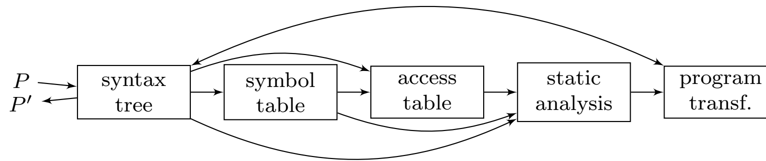


Figure 3: Static analysis and source transformation for automated replication

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/lou1306/Replication/releases/tag/v2.0`. The prototype is written in Go, partly because of the availability of the actively maintained goSpace library, and partly because the standard libraries for Go provide built-in facilities for static analysis and transformation of abstract syntax trees, which has greatly reduced our overall implementation effort.

5.1. Distributed Lookup

Let us now consider a distributed system composed of n computational nodes, each executing a separate program, and interacting through a decentralised data store equipped with m memory locations. Following a similar schema to those used in distributed lookup protocols (e.g., Chord [10]), 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 scheme 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

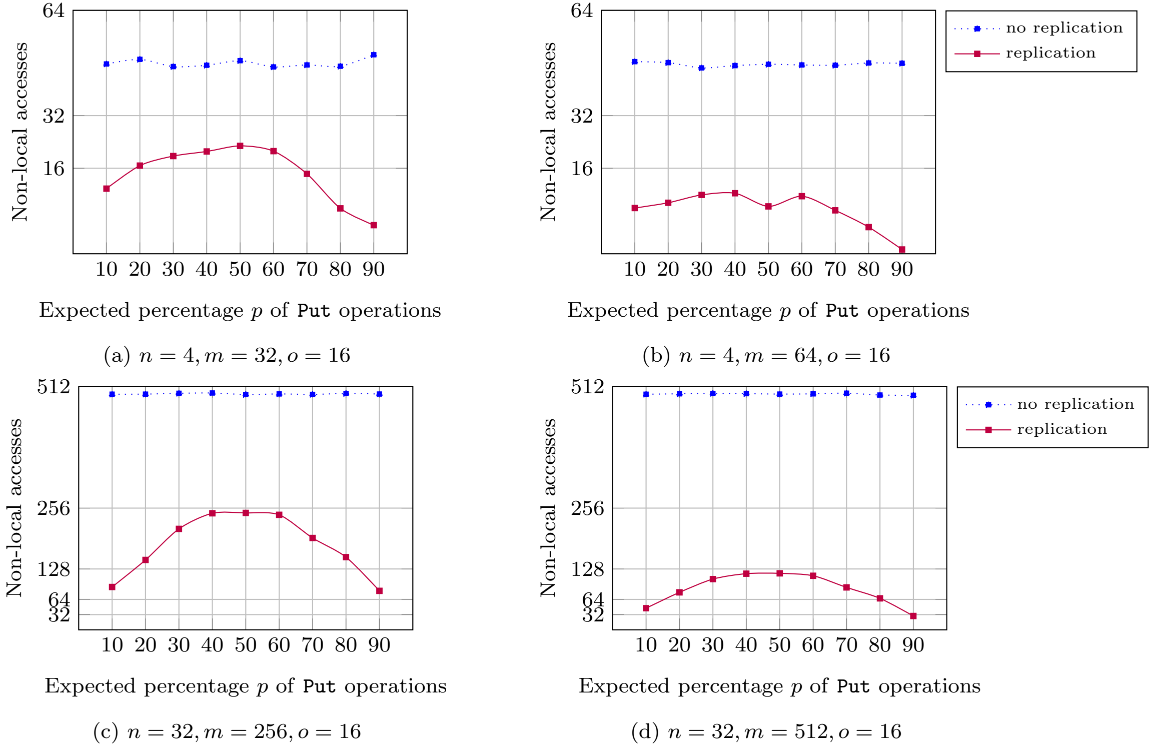
($address, value$) representing values held at different memory addresses. The structure of the program follows that of Listing 7. For simplicity, we assume that all read operations are `QueryP`. Write operations are of course `Put`.

Experiment 1. To evaluate the effect of replication on the system, we conduct the following experiments. We initially consider 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 are summarised in Figures 4a, 4b, 4c and 4d 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 4a–4d, 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).

Experiment 2. To assess the difference between proactive and reactive replication (Sects. 3.1 and 3.2, respectively), we compare their memory usage in terms of tuples stored in the replicated space.

We consider proactive and reactive replication for the cases $\{n=4, m=8\}$ and $\{n=4, m=16\}$, fixing $o=128$, and varying p in $\{10, 20, \ldots, 90\}$, ending up with 36 configurations. For every configuration, we generate 10 test cases with randomized access pattern, and run each test case 10 times. The entire experiment thus requires 3600 runs. In every run, we measure the maximum



(a) $n = 4, m = 32, o = 16$ (b) $n = 4, m = 64, o = 16$

(c) $n = 32, m = 256, o = 16$ (d) $n = 32, m = 512, o = 16$

Figure 4: Non-local read or write operations with and without replication

number of tuples stored simultaneously in the replicated space, and then compute the average of these measurements for each given configuration. Notice that this number may easily exceed m, since it includes tuple replicas, and also because processes never remove tuples (thus, multiple tuples may refer to the same memory location).

The experimental results are summarised in Figure 5. Reactive replication always uses less space than proactive replication, and savings increase with p, until we reach $p = 80\%$. From there on, the number of tuples stored by proactive replication appears to decrease, likely because queries become so scarce that there is no longer any need to create replicas; nevertheless, the savings brought about by the reactive scheme are still significant. Note that, since both approaches enforce strong consistency, the success rate of the single read or write operations is the same regardless of the chosen scheme. Therefore, reactive replication seems beneficial in systems where storage is expensive but slightly longer execution time for query operations can be

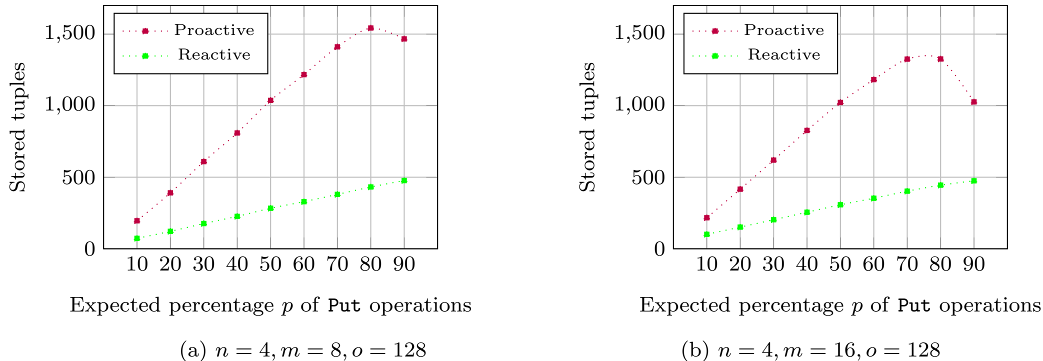


Figure 5: Tuple space usage under proactive and reactive replication.

afforded.

5.2. Work-stealing Peers

The following scenario is inspired by work stealing scheduling (e.g., [11]), a collaborative strategy for allocating tasks among processing units in the attempt to increase the overall efficiency of a concurrent system.

Let us consider a network of peers processing incoming service requests from external clients. The network provides multiple services. Any peer (i.e., the *receiver*) can receive requests for any of the services. By default, a specific peer is the *default target* for each service, i.e., it is in charge of handling all the requests for that service. For simplicity, we assume that the number of peers and services is the same, and that the i-th peer is the default target for the i-th service.

Upon an incoming service request, if the default target peer is particularly busy, the receiver will take over, in a work-stealing fashion. Keeping in mind that the network has no central control, such policy can be enacted as follows. Each peer keeps track of its own system load locally, by increasing or decreasing a simple counter when receiving or after processing a service request, respectively. Therefore, the counter represents the number of requests that the peer needs to handle. At the same time, depending on a predetermined threshold on the system load, the peer becomes *busy* or *non-busy*. A peer that receives a service request for a busy default target can decide to *steal* it, i.e., handle it locally. If the default target is not busy, instead, the peer will simply forward the request to it.

The above mechanism requires exchange of information across peers. We wish to use our approach to evaluate the effect of replication, and possibly

storage limitations, on the efficiency of the network.

We model the system described above as a concurrent Go program where n is the number of peers, T_b is the busy threshold, T_L is the upper load threshold, and p_a is a peer's probability of accepting (vs. handling) a request. In addition to the n peers, the system contains one request generator. The request generator randomly creates $100n$ service requests and stores them in the tuple space of their receiving peers. We model a service request as a triple (i, j, τ) where i is the id of the peer receiving the request, j the service id, and τ a timestamp recording the creation time of the request.[2]

Each peer i repeatedly tries to either *accept* a new request, or *handle* an already-accepted request. The request accepting behaviour works as follows. First, i tries to withdraw a request tuple from its own space with a `GetP` operation. If the operation is successful, it returns a triple (x, y, t) where x is a peer identifier, y a service identifier, and t the time when the request was created. If $y = i$, then the peer accepts it and increments a system load counter l. When the new value of l reaches the busy threshold T_b, i inserts into its own space a *busy tuple* $(i,$ `"busy"`$)$. If $y \neq i$, then the peer checks whether the default target y is busy by looking for the busy tuple in y's space. If y is busy, i accepts the request and increments l, possibly becoming busy by doing so. Otherwise, i simply forwards the request to y by performing a `Put`(y, y, t) operation.

The handling behaviour is as follows. The peer first checks whether $l \geq 0$. If this is the case, then there is (at least) one request to be processed, so the peer can decrease l (meaning that the request has been served). When l becomes less than T_b, the peer removes the busy tuple from its own space to signal that it is no longer busy.

The choice between the two behaviours is regulated by the parameters p_a and T_L. At every iteration, a peer accepts or handles a request with probabilities p_a or $1 - p_a$, respectively. However, if a peer's load reaches the threshold T_L, then it will no longer accept any request until its load falls back below the threshold. This models the fact that a peer has limited resources and cannot accept an arbitrarily large number of requests without handling them.

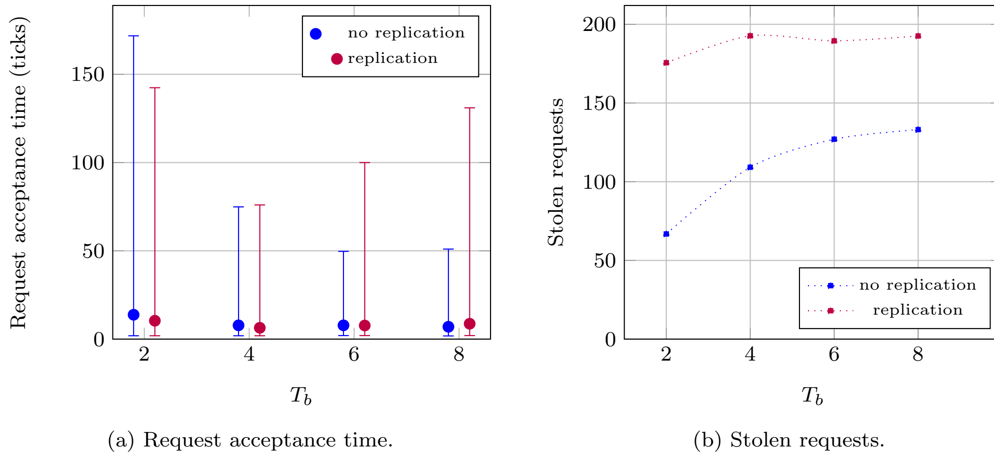
[2]The timestamp is only used to measure performance, and has no impact on the behaviour of the peers.

Experiment 3. First, we are interested in assessing how replication affects the time taken by the system to accept a request, as well as the effectiveness of the work-stealing mechanism itself. To do so, we measure request acceptance time in four configurations, with T_b in $\{2, 4, 6, 8\}$, and fix $n = 3, p_a = 75\%$, and $T_L = 1.5T_b$, under a load of 300 requests. We run each configuration 20 times, with and without replication, for a total of 160 test cases. We then collect the average number of stolen requests, as well as the minimum, maximum, and average request acceptance time.

The experimental results are shown in Figure 6. Notice that wall time was unsuitable for these measurements, since the replicated program must perform more expensive operations (e.g., replicating a tuple to multiple spaces) than the non-replicated one. Therefore, all times are measured using logical timestamps ("ticks"), taken from a counter that increases every time one of the processes (i.e., the peers and the request generator) executes an iteration, and thus are not affected by our program transformation.

Figure 6b shows how the number of stolen requests in the replicated program plateaus at nearly 2/3 of the total number of request. This is the probabilistic optimum, since 1/3 of requests, on average, are addressed to the default peer and thus cannot be stolen. In Figure 6a, we can see that increasing the busy threshold T_b reduces the maximum acceptance time of requests in the non-replicated program. This makes sense, as the individual peers are able to accept more request before becoming busy. In the replicated program, however, we observe the best performance with $T_b = 4$, while higher values lead to an increase in the maximum time. This may be related to the fact that peers are more likely to steal requests in the replicated system. Stealing more requests would make peers more likely to hit the load threshold T_L, forcing them to handle some accepted requests before they can accept new ones. This could negatively affect acceptance times. Furthermore, some unlucky requests may be initially addressed to a non-default peer, and then forwarded to their default one right before it becomes unable to accept requests. Notice that replication barely affects the average acceptance time, which is quite close to the minimum both with and without replication. This suggests that most requests are accepted in a much shorter time than the maximum, otherwise the average time would be noticeably higher.

Experiment 4. We now would like to check how memory-limited tuple spaces impact the quality of service provided by the peer network. Intuitively, by enforcing a limit r on the number of tuples that each server can store, we



(a) Request acceptance time.

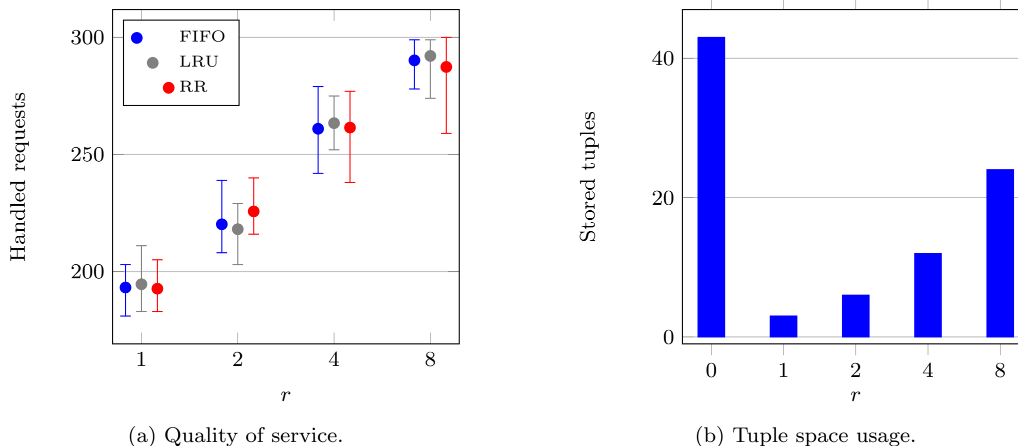
(b) Stolen requests.

Figure 6: Impact of replication on the work-stealing case study.

introduce the possibility of *service degradation*, i.e., that the network drops some requests without handling them. We are interested in measuring the relation between r and service degradation under different replacement policies. Thus, for each policy, we set up a round of 20 experiments with $n = 3, T_b = 2, T_L = 3, p_a = 75\%$, and r in $\{1, 2, 4, 8\}$, for a total of 240 test cases. Then, we measure the minimum, maximum, and average number of handled requests under each configuration, as well as the average maximum number of tuples stored in the whole replicated space.

The measurements are shown in Table 7. Predictably, a very low memory limit r leads to severe degradation, regardless of the policy. However, a value of $r = 8$ allows every policy to handle at least 95% of requests on average. Specifically, FIFO handles 290 requests out of a possible 300 (96.7%); LRU handles 292 (97.3%); RR handles 287 (95.7%). LRU displays the best average performance in all settings except $r = 2$. While the average quality of service shows little difference across all policies, the interval between the minimum and maximum number of handled requests over all runs appears to worsen under FIFO and RR as r increases from 1 to 4. When $r = 8$, both policies show improvement: this is probably because there are at most 300 requests to process, so the maximum cannot exceed this value. LRU, on the other hand, displays an approximately constant range across all values of r. Finally, we observe that, for low values of r, the much lighter RR policy behaves on par with, if not better than, the others.

The effects of memory-limited tuple spaces on storage requirements are



(a) Quality of service.

(b) Tuple space usage.

Figure 7: Service degradation and memory usage.

apparent in Figure 7b. Naturally, low values of r result in a much lower storage footprint, but also lead to severe service degradation, as shown earlier. Still, with $r = 8$, the system requires 44% less space than in the unlimited-memory case ($r = 0$): we store at most 24 tuples, as opposed to 43. Given that such a system will drop less than 5% of all requests, the memory-limited approach may be well suited in a setting where such levels of degradation are acceptable while storage space comes at a premium.

6. Related work

In addition to the pSpaces family of implementations, tuple space systems have been embedded in a number of different programming languages. Java implementations include Klava [12] for Klaim, and jRESP[3] for SCEL [13]. 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 [5], tuple replication has also been implemented in X10 [14], a general-purpose language for large-scale distributed systems [3]. An extension of Lime, a distributed tuple space for mobile ad-hoc networks, relies on replication to increase availability [15]. While not explicitly aimed at replication, LogOp [16] also extends Linda operators with *scopes*, which dynamically resolve to one or more tuple spaces upon which the operation

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

will be performed. 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 [17]. Spatial distribution of tuples is a rather different approach to ours where tuples contain both content and replication rules [18]; 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 [19], 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 in [20]. Consistency models for replicated data are covered in [21]. Dynamic replication has been considered in [22, 23].

Several program transformation frameworks for different languages are available. A popular framework for C and C++ is ROSE [24], 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 [25]. 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. This was another reason for choosing the Go implementation among the available ones of pSpaces.

Static verification of concurrent Go programs for bounded liveness and safety has been considered in [26, 27]. Bounded analysis of concurrent programs for safe replication has been proposed in [28].

7. Conclusion and Future Work

We have presented RepligoSpaces, a replica-aware extension of goSpace, and the implementation of Klaim (pSpaces) in Go. We have also discussed how RepligoSpaces fits within a fully-mechanisable procedure for automated replication of programs 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. To realistically model scenarios where memory is constrained, we have enriched the approach with memory-limited tuple spaces implementing several well-known replacement policies. Finally, we have introduced an alternative, reactive replication scheme that replicates tuples on-demand. This reactive mechanism still preserves strong consistency, and for some applications may require less storage space than the proactive one at the cost of additional computations. We implemented a prototype that can automatically apply these replication schemes to a replica-unaware Go program, allowing us to evaluate their impact empirically. We then demonstrated this prototype on a selection of synthetic examples, where replication provides significant benefits. We do not claim these results to be general: replication may be less effective or even detrimental to performance, depending on the case study. Our experiments instead aim at demonstrating our tool-assisted methodology to evaluate the costs and benefits of replication in a distributed system.

In the near future, we 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. The analysis presented in this paper is simple and lightweight, as it only requires simple visits of the abstract syntax tree of the program and was mainly aimed at demonstrating the usage of our experimental framework for quick prototyping of replication schemes. We plan to initially focus on simple and efficient techniques to complement the existing analysis with limited effort. To name a few, constant propagation [29] can reduce the overall number of formal fields or restrict the possible values of a given formal field collecting them over different branching paths; while abstract interpretation [30] can overapproximate the interval ranges of the integer variables used as formal fields. Dynamic techniques may also complement this work in interesting ways. For instance, one might extract a probabilistic model of a running program via a dynamic analysis, generate a goSpace program that captures this model, and finally use our workflow to analyse whether replication could guarantee any performance benefits. Dynamic analysis may also become necessary if we shift our focus towards more open systems, where processes may join and leave as they please. This would likely require extending the programming interface to register and de-register spaces and adjust the replication mechanism accordingly. To evaluate these extensions, we should consider further scenarios where replication has successfully been applied to

other contexts, such as database systems [31], and cloud computing [32], as well as other consistency models [21, 33].

Clearly, tuple eviction may alter the runtime behaviour of processes that coordinate or synchronize by means of tuples. This currently limits the applicability of our limited-memory transformation. To address this limitation, we may let the user specify a set of patterns to identify *critical* tuples that should never get evicted. This feature, in turn, would introduce the risk of some tuple space getting filled with critical tuples and being unable to accept new ones. Still, we could formally check the absence of such risks by means of automated verification tools such as Gomela [34].

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 to provide different analyses and consistency models to choose from, 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.

References

[1] D. Gelernter, Generative communication in Linda, ACM (TOPLAS) 7 (1) (1985) 80–112.

[2] D. Gelernter, Multiple tuple spaces in Linda, in: E. Odijk, M. Rem, J.-C. Syre (Eds.), Parallel Architectures and Languages Europe, Volume II: Parallel Languages (PARLE), Vol. 366 of LNCS, Springer, Eindhoven, The Netherlands, 1989, pp. 20–27. doi:10.1007/3-540-51285-3_30.

[3] V. A. Saraswat, R. Jagadeesan, Concurrent clustered programming, in: CONCUR, Vol. 3653 of LNCS, Springer, 2005, pp. 353–367.

[4] R. De Nicola, G. L. Ferrari, R. Pugliese, KLAIM: A kernel language for agents interaction and mobility, IEEE Trans. Software Eng. 24 (5) (1998) 315–330.

[5] M. Andrić, R. De Nicola, A. Lluch-Lafuente, Replica-based high-performance tuple space computing, in: COORDINATION, LNCS, Springer, 2015, pp. 3–18.

[6] R. Pike, Go at google, in: SPLASH, ACM, 2012, pp. 5–6.

[7] L. Kaminskas, A. Lluch-Lafuente, Aggregation policies for tuple spaces, in: COORDINATION, Vol. 10852 of LNCS, Springer, 2018, pp. 181–199. doi:10.1007/978-3-319-92408-3_8.

[8] A. Uwimbabazi, O. Inverso, R. De Nicola, Automated replication of tuple spaces via static analysis, in: 9th International Conference on Fundamentals of Software Engineering (FSEN), Vol. 12818 of LNCS, Springer, 2021, pp. 18–34. doi:10.1007/978-3-030-89247-0_2.

[9] P. Benedicte, C. Hernández, J. Abella, F. J. Cazorla, RPR: A random replacement policy with limited pathological replacements, in: 33rd Annual ACM Symposium on Applied Computing (SAC), ACM, 2018, pp. 593–600. doi:10.1145/3167132.3167197.

[10] I. Stoica, R. T. Morris, D. R. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: SIGCOMM, ACM, 2001, pp. 149–160.

[11] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, C. Wilkerson, Scheduling threads for constructive cache sharing on CMPs, in: 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM, 2007, pp. 105–115. doi:10.1145/1248377.1248396.

[12] L. Bettini, R. De Nicola, R. Pugliese, Klava: a Java package for distributed and mobile applications, Softw. Pract. Exp. 32 (14) (2002) 1365–1394.

[13] R. De Nicola, D. Latella, A. Lluch-Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, A. Vandin, The SCEL language: Design, implementation, verification, in: The ASCENS Approach, Vol. 8998 of LNCS, Springer, 2015, pp. 3–71.

[14] M. Andrić, R. De Nicola, A. Lluch-Lafuente, Replicating data for better performances in X10, in: Semantics, Logics, and Calculi, Vol. 9560 of LNCS, Springer, 2016, pp. 236–251.

[15] A. L. Murphy, G. P. Picco, Using Lime to support replication for availability in mobile ad hoc networks, in: COORDINATION, Springer, 2006, pp. 194–211.

[16] R. Menezes, A. Omicini, M. Viroli, Dynamic composition of coordination abstractions for pervasive systems: The case of LogOp, in: S. Ossowski, P. Lecca (Eds.), 27th Symposium on Applied Computing (SAC), ACM, Riva del Garda, Italy, 2012, pp. 1557–1559. doi:10.1145/2245276.2232025.

[17] A. Corradi, L. Leonardi, F. Zambonelli, Distributed tuple spaces in highly parallel systems, Tech. Rep. DEISLIA-96-005, University of Bologna, Italy (1996).

[18] M. Mamei, F. Zambonelli, L. Leonardi, *T*uples *O*n *T*he *A*ir: a middleware for context-aware multi-agent systems, in: WOA, PEB, 2002, pp. 108–116.

[19] R. De Nicola, R. Pugliese, A. I. T. Rowstron, Proving the correctness of optimising destructive and non-destructive reads over tuple spaces, in: COORDINATION, Vol. 1906 of LNCS, Springer, 2000, pp. 66–80.

[20] A. N. Bessani, E. A. P. Alchieri, M. Correia, J. da Silva Fraga, DepSpace: A byzantine fault-tolerant coordination service, in: EuroSys, ACM, 2008, pp. 163–176.

[21] A. D. Fekete, K. Ramamritham, Consistency models for replicated data, in: Replication: Theory and Practice, Vol. 5959 of LNCS, Springer, 2010, pp. 1–17. doi:10.1007/978-3-642-11294-2_1.

[22] M. Casadei, M. Viroli, L. Gardelli, On the collective sort problem for distributed tuple spaces, Sci. Comput. Program. 74 (9) (2009) 702–722.

[23] G. Russello, M. R. V. Chaudron, M. van Steen, Dynamically adapting tuple replication for managing availability in a shared data space, in: COORDINATION, Vol. 3454 of LNCS, Springer, 2005, pp. 109–124.

[24] D. Quinlan, C. Liao, The ROSE Source-to-Source Compiler Infrastructure, in: Cetus Users and Compiler Infrastructure Workshop, in conj. with PACT (2011), 2011.

[25] C. Lattner, Llvm and Clang: Next generation compiler technology., in: The BSD Conference, 2008.

[26] J. Lange, N. Ng, B. Toninho, N. Yoshida, Fencing off go: liveness and safety for channel-based programming, in: POPL, ACM, 2017, pp. 748–761.

[27] J. Lange, N. Ng, B. Toninho, N. Yoshida, A static verification framework for message passing in Go using behavioural types, in: ICSE, ACM, 2018, pp. 1137–1148.

[28] G. Kaki, K. Earanky, K. C. Sivaramakrishnan, S. Jagannathan, Safe replication through bounded concurrency verification, Proc. ACM Program. Lang. 2 (OOPSLA) (2018) 164:1–164:27.

[29] M. N. Wegman, F. K. Zadeck, Constant propagation with conditional branches, ACM Trans. Program. Lang. Syst. 13 (2) (1991) 181–210.

[30] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: POPL, ACM Press, 1978, pp. 84–96.

[31] S. Elnikety, S. G. Dropsho, W. Zwaenepoel, Tashkent+: Memory-aware load balancing and update filtering in replicated databases, in: EuroSys, ACM, 2007, pp. 399–412.

[32] R. R. Karandikar, M. B. Gudadhe, Comparative analysis of dynamic replication strategies in cloud, IJCA. TACIT2016 (1) (2016) pp. 26–32.

[33] D. Terry, Replicated data consistency explained through baseball, Commun. ACM 56 (12) (2013) 82–89.

[34] N. Dilley, J. Lange, Automated verification of Go programs via bounded model checking, in: 36th International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 1016–1027. doi:10.1109/ASE51524.2021.9678571.