IMT School for Advanced Studies, Lucca

Lucca, Italy

Improving the efficiency of tuple spaces

PhD Program in Computer Science

XXX Cycle

By

Vitaly Buravlev

2018

The dissertation of Vitaly Buravlev is approved.

Program Coordinator: Prof. Mirco Tribastone, IMT Institute for Advanced Studies, Lucca

Supervisor: Prof. Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

Supervisor: Dr. Claudio Antares Mezzina, IMT Institute for Advanced Studies, Lucca

The dissertation of Vitaly Buravlev has been reviewed by:

eva Kühn, TU Wien, Vienna, Austria

Emilio Tuosto, University of Leicester, Leicester, United Kingdom

IMT School for Advanced Studies, Lucca

2018

To my friend

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I thank my parents who always support and inspire me for all my accomplishment.

Undoubtedly, I am very grateful to my supervisors, Prof. Rocco De Nicola and Dr. Claudio Antares Mezzina, for providing me the opportunity of this work. Their continuous support, knowledge, and ideas guide me from the very beginning. It was always a pleasure to talk to them and discuss during our joint meetings that I always enjoyed.

Part of the work has been done at the Technical University of Denmark under the supervision of Prof. Alberto Lluch Lafuente. His ideas and guidance helped me to accomplish an important part of this work. Being a part of the Formal Methods Group was great and they made my visiting period unforgettable.

I highly appreciate the time and efforts of the reviewers, Prof. eva Kühn and Prof. Emilio Tuosto. Their comments and suggestions were very useful and undoubtedly has improved the thesis.

I am thankful to the members of the SysMA group at IMT School for their discussions and suggestions and the IMT administration for their support and help.

Declaration

Most of the material in this thesis has been published. In particular: most of Chapter 2 and Chapter 4 are based on [1] and [2], coauthored with Rocco De Nicola and Claudio Antares Mezzina, IMT School for Advanced Studies, Lucca. Chapter 5 is inspired by collaboration with Alberto Lluch Lafuente, Technical University of Denmark, Lyngby, and is based on [3], coauthored with Rocco De Nicola, Alberto Lluch Lafuente and Claudio Antares Mezzina.

Vita

March 12, 1988	Born, Moscow, Russia
2005-2011	Degree in Computer Science Average grade: 4.96/5 with honors Informatics and Control Systems, BMSTU, Moscow, Russia
2014-2018	PhD in Computer Science IMT School for Advances Studies, Lucca.
2017	Visiting Research Student DTU Compute - Section for Formal Methods, Technical University of Denmark.

Publications

1. V. Buravlev, R. De Nicola, C.A. Mezzina, "Tuple Spaces Implementations and Their Efficiency" in *COORDINATION 2016*, pp. 51-66, 2016.
2. R. Barbi, V. Buravlev, C.A. Mezzina, V. Schiavoni "Block Placement Strategies for Fault-Resilient Distributed Tuple Spaces: An Experimental Study - (Practical Experience Report)" in *DAIS 2017*, pp. 67-82, 2017.
3. V. Buravlev, R. De Nicola, C.A. Mezzina, "Evaluating the efficiency of Linda implementations" in *Concurrency and Computation: Practice and Experience*, vol. 30, issue 8, 2018.
4. V. Buravlev, R. De Nicola, A.L. Lafuente, C.A. Mezzina, "Improving availability in distributed tuple spaces via sharing abstractions and replication strategies" in *PDP 2018*, pp. 302-305, 2018.

Presentations

1. V. Buravlev, “Tuple space implementation and their efficiency” at *COORDINATION 2016*, Heraklion, Greece, 2016.
2. V. Buravlev, “Improving availability in distributed tuple spaces via sharing abstractions and replication strategies” at *PDP 2018*, Cambridge, United Kingdom, 2018.

Abstract

Linda provides high-level linguistic abstractions for concurrent programming with operations for synchronization and exchange of values between different programs that share information by accessing common repositories named tuple spaces. Despite their expressive power and their simplicity, there are several challenges in implementing tuple space systems, which prevent the Linda model to be widespread.

The goal of this work is to provide an efficient implementation of the Linda coordination model. As a starting point, we take KLAIM and its Java implementation and we improve on it after evaluating the performances of selected implementations of tuple spaces and discussing their different implementation choices concerned with data structures and querying techniques. Our KLAIM implementation is also extended with abstractions for data replication based on automatic data placement strategies.

Chapter 1

Introduction

1.1 Context and overview

Nowadays, people are surrounded by distributed systems: they use them in their work by relying on clouds storages, by taking advantage of online services, and when searching information on the Internet; they chat with friends via Facebook, Telegram, and Skype; they use them on their laptops and mobile devices. Distributed systems are also used for sharing resources and for delivering services to high-performance computing. Distributed systems work not only on the Internet but also in intranets: banks use them to store and process money transactions, IT companies create specialized clusters for Big data processing, etc.

A distributed system is a network that consists of autonomous computational units: computers, servers, mobile devices. All its components are connected using a middleware and can work over different networks.

When we talk about components of distributed systems, an important problem arises: how to connect them in such a way they constitute a coherent ensemble? Communication and coordination of components play a crucial role in distributed systems since they determine how components of systems interact with each other, exchange data, and synchronize their work. According to the classification in Chapter 2 of [4], three communication paradigms can be singled out: *interprocess communication*, *remote* *invocation*, and *indirect communication*. The interprocess paradigm refers to low-level techniques of direct and explicit communication. This means that a process, in order to communicate with another one, has to know its address (see Figure 1). Remote invocation is common in *client-server* interactions and used when one of the two processes provides an operation or a service and the other one accesses to it (see Figure 2). Indirect communication is based on the idea of the absence of space- and time- coupling when senders and receivers do not know each other (see Figure 3).

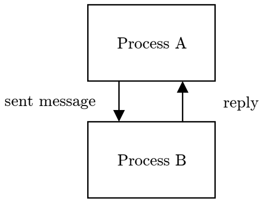

Figure 1: Interpr. communic.

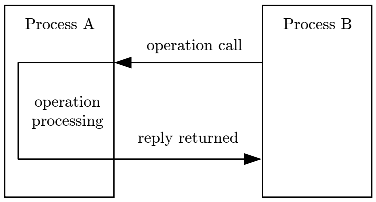

Figure 2: Remote invocation

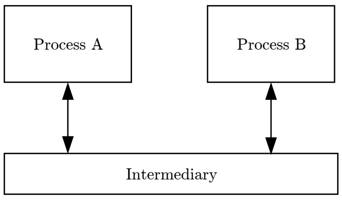

Figure 3: Indirect communication

Each of the above-mentioned communication paradigms has a different level of abstraction. Usually, modeling communication via message passing is more complicated than using primitives based on shared memory, see Chapter 4 of [5].

Indeed, describing and building a communication protocol between two processes requires more efforts than relying on a common memory which processes can asynchronously access.

One can single out two approaches on how to deal with shared memory: when it is maintained by one process and when it is split into parts and each part is controlled by a different process, i.e. *distributed shared memory* (DSM) is used. The first approach leads the process bearing the memory to be a bottleneck since all the communications have to be mediated through it. For DSM instead, the common problems are which data consistency model to use and how to maintain it. The need to solve these problems comes with an additional implementation complexity of the middleware that constitutes and maintains the shared memory. Anyway, the programmer trades the system efficiency with the flexibility and simplicity of programming interactions between components based on DSM. The situation is similar when we compare higher level languages with assembly languages, where any overhead is usually outweighed by the benefits of easier programming and portability.

In this work, we focus on *tuple spaces*, an associative shared memory implementing the Linda coordination model. In this model, data are retrieved not using their address but mentioning (part of) their content, by relying on pattern matching. The processes of a system do interact by accessing a common memory that might be centralized or distributed. The Linda model provides a small set of operations to write, read, withdraw data from the memory and to spawn a process. Since the paradigm consists of just four operations, it can be easily added to any programming language either as an external library or as an extension of the target language. Being a high-level model of communication, tuple spaces allow expressing in a neat and compact way some mechanisms which are typical of modern distributed systems and used for load balancing, fault tolerance, synchronization.

As we said, the simplicity of the tuple space model comes with the price of efficiency. In this document, we focus on the problem of how to make tuple spaces efficient. We start from surveying techniques used for storing and querying data which are typically used in well-known tuple space implementations. Then, we evaluate several implementations of tuple spaces highlighting their pros and cons. Afterwards, by taking advantage of this evaluation phase, we propose a new efficient implementation of KLAIM, a process algebra based languages that relies on the

Linda model and on a distributed shared memory. Finally, we extend KLAIM with constructs for modeling data replication. The description of the new constructs, as well as their implementation details, is accompanied by the experimental evaluation of the proposed middleware.

1.2 Background and motivations

Tuple spaces have been considered in many works. Some of them study how to adapt them to program context-aware applications [6, 7, 8] or how to query tuple spaces via a SQL-like language [9]. In what follows, we talk about the works that aim at offering efficient implementations of tuple spaces. In particular, we discuss works devoted to the problem of boosting pattern matching and of optimizing the distribution of tuple spaces.

There are two approaches to make the Linda model efficient: to optimize tuple spaces management at compile-time and at run-time. The former can be applied only on compiled languages, whereas the latter does not have this restriction.

The compile-time optimization implies the use of a preprocessor to reduce or eliminate run-time searching. Usually, this approach requires two steps: code analysis and code generation. In the first phase, the preprocessor analyzes a Linda program to determine patterns of how tuples are used and how processes communicate [10, 11]. Then, in the second phase, a new program code is generated that optimizes calls to operations on tuple spaces. As examples, we can consider the following cases considered in [12]:

- constant tuples that are used as semaphore-like signals can be substituted by variables-counters meaning that no search is required;
- hash tables can be used for tuples needed to store parameters (with a key-value structure) providing a constant search time;
- a data update operation, that consists of a sequential call of withdrawing and writing operations, can be substituted by a single operation that reduces the number of interactions between processes.

However, it is not always possible to use techniques based on a compile-time analysis. In fact, this approach does not appear to be that appealing nowadays since there are no recent works on it.

The second approach is more widespread. There exist several techniques used to boost pattern matching. Since sequential reading is rarely required and linearly depends on the number of tuples in a tuple space, an important problem is how to avoid the linear search time. A general approach is to extract a specific information from tuples and then use this information while performing the pattern matching. The most common techniques rely on hash tables as a base data structures to store tuples where the key can be computed using tuple content [13, 14, 15]. It is worth to note that using hash tables is not a universal solution but a trade-off between performance and expressiveness. On one hand, it provides a constant search time when a tuple space is used as a key-value store. On the other hand, it limits expressiveness of the paradigm since there are restrictions on templates that can be correctly matched. This is why some implementations, such as [14] and [15], restrict the key to a single field.

A good balance between performance and flexibility of using any templates can be achieved when *indexing* techniques are used [9, 16]. Precomputed indexes provide more flexible search and acceptable performances. The drawback is the necessity to maintain an index structure which implies memory overhead.

Other techniques worth mentioning are *partitioning*, *prefetching*, and *concurrent data structures*. Partitioning utilizes a specific information about tuples to split a tuple space into several smaller ones. For example, in [17] the specific information is the structure of tuples. The technique called prefetching [17] allows processes to send an asynchronous request for a tuple and to continue their work while the search is performed. When the requested tuple is needed, if found, it is used without waiting. Using parallel or concurrent data structures is another way to improve the throughput of the tuple spaces since several operations can be performed in parallel.

Replication is used in tuple spaces either to guarantee fault tolerance or to increase data availability. According to the CAP theorem (Consistency-

Availability-Partition tolerance) [18], strong consistency of data, data availability and partition tolerance cannot be guaranteed simultaneously, but only any two out of the three. Usually, network partitioning has to be tolerated meaning that either consistency or availability requirements have to be softened. When talking about making data more available, we mean that the average access time for these data should decrease. Fault tolerance prevents data loss by keeping different copies of the data. Tuple space systems that aim at fault tolerance take care of how to prevent data loss and do not consider performance an important issue[19, 20, 21, 22] .

There are several approaches to improve the time of data reading and data manipulation. One of them is to relax consistency requirements. Among many consistency models [23], one can distinguish *strong consistency* which is the strictest of all consistency models and other *relaxed memory models*. By specifying a certain consistency model for certain classes of data, we can manage the performance of reading and writing operations. For instance, for data that simulate synchronization primitives, such as *barriers*, which require atomic operations on them, it makes sense to use strong consistency. While for data that do not require atomic operations, weaker consistency models can be used.

Replica placement is another important part of replication mechanisms. Placement algorithms can utilize different information: network information [20, 24, 25], characteristics of the network or of its nodes [26], the activity of network nodes based on Hot-Spot [27] or data popularity [28]. Usually, fast placement computing and high precision of placement cannot be achieved simultaneously and it is required to choose which characteristic is preferable for the system. Replication is a standard technique [29] in Content Delivery Networks (CDN), however, for CDN the speed of the algorithm for replica placement is not very important. For real-time systems, it is acceptable to sacrifice high precision to be able to work over bigger networks. Among techniques to decrease the complexity of placement's computing, it is worth mentioning the work in [30] where a decentralized summary of recent accesses is used to achieve near-optimal performance, and the work in [25] that applies a two-steps procedure of choosing places where data have to be stored.

We believe that it is possible to enjoy the flexibility and the easiness of programming with tuple spaces and, at the same time, be sure that communication between components of the distributed system performs efficiently. In our view, while designing distributed systems, it is difficult to determine in advance where to place data and this problem should be solved automatically by the middleware of tuple spaces. At the same time, programmers of a distributed system know the types of data to be transferred and their characteristics (allowed consistency and how they can be stored). By dividing responsibility between programmers and the middleware, distributed systems can gain better performances.

1.3 Contribution and organization

1.3.1 Contribution

In this work, we answer the following questions:

– what makes tuple space efficient?

– what is the appropriate abstraction for sharing data via tuple spaces?

Specifically, the contributions of our work are the following ones:

- an experimental evaluation of a number of tuple space implementations. Starting from the opinion that practical evaluation gives more information about the way certain techniques and approaches affect performances of tuple spaces and which of them are more important than others, we perform experiments with several preselected tuple spaces using case studies that test different aspects of implementations, such as pattern matching, communication or computation speed.

- an implementation of customizable tuple spaces in which programmers can choose the underlying data structure where each node will store its tuples. In this way, it is possible to create a system such that certain nodes offer fast data retrieval time while other, for example, fast writing time.

- a programming abstraction for sharing data via tuple spaces. Sharing primitives that can be used by the programmer to explicitly specify the localities where data has to be replicated is beneficial in static networks that do not change over time and when the programmer knows the characteristics of localities and network. When using dynamic networks, it is more difficult to determine where to put data. We propose to use a sharing abstraction where the programmer instead of directly specifying the localities can specify a "group" for which a particular tuple has to be replicated. A group is a dynamic set of nodes, since at run-time a node may join or leave. By using this approach, it is possible to optimize data placement according to certain network characteristics.

1.3.2 Structure of the thesis

The rest of the work is organized as follows:

- Chapter 2 provides a deep and wide survey of the state of art of tuple space's implementations, replication and replica placement. We start this by reviewing some of the main implementations of general purposes tuple spaces. Then, we consider replication and review tuple spaces that exploit it and describe techniques used for replica placement.

- Chapter 3 is devoted to the programming language KLAIM and its implementation in Java, named KLAVA. First, we formally define KLAIM presenting its syntax and semantics. Then, we describe the application interface (API) of KLAVA and show how to use this framework to program distributed applications. Finally, we report on the change that we did to have a more efficient KLAIM implementation.

- Chapter 4 presents an experimental evaluation of different tuple space's implementations. Two series of experiments are performed. First, the old implementation of KLAIM is compared with the new one in order to show the obtained improvements. Then, the new implementation of KLAIM is compared with other preselected implementations.

- Chapter 5 presents a tuple space with programming abstraction for data sharing with the purpose of increasing data availability. First, we describe the sharing abstraction and how it is applied to KLAIM providing details about consistency model and replication strategies. Then, we report results of the experimental evaluation of this tuple space.
- Chapter 6 summarizes the main results of the work and draws possible directions for further research.

Chapter 2

Background on tuple spaces

A distributed system consists of different components located on different nodes and possibly in different networks all over the Internet. Communication among its nodes is crucial for its functioning. Communication paradigms are groups of techniques that describe how processes can communicate with each other to exchange data and to synchronize their actions.

This chapter is mainly focused on the tuple space paradigm as a mean of communication among components. In Section 2.1, we will overview different communication paradigms and then we will focus on the Linda model which is based on guaranteed a controlled access to the shared memory. In Section 2.2, we discuss the main data structures and speed-up techniques that can be used while building a tuple space. Section 2.3 reviews a number of current implementations of tuple spaces highlighting their main features. Section 2.4 discusses replication mechanisms for tuple spaces and reviews some of the implementations of tuple spaces which make use of them.

2.1 The Linda model

In this section, we discuss communication paradigms focusing on indirect communication and the Linda model. Then, we describe tuple spaces stressing their weak and strong points.

2.1.1 Communication paradigms

There are three communication paradigms (cf. [4], Chapter 2) used in programming parallel and distributed computing: *interprocess communication*, *remote invocation* and *indirect communication*. Communication can occur between processes located either on the same machine or on different ones.

Interprocess communication. Interprocess communication refers to techniques that provide an intuitively simple abstraction of communication between processes. It is represented by low-level socket programming, e.g., network sockets [4] and Unix domain sockets [31], and more advanced message passing. Often, interprocess communication is used to build higher level programming abstractions. For instance, message passing provides primitives for sending messages and receiving messages from other processes. It requires programmers to specify the addresses of senders and recipients, data and metadata to be sent. Message passing is a popular technique that was proposed for the first time in the formal language *Communicating Sequential Processes* [32] and has a number of implementations such as *Actor model* [33] and *Message Passing Interface* (MPI) [34]. Most recently, the language *Go* [35] defined at Google has message passing as the basic communication primitive.

Remote invocation. Remote invocation is considered the most common paradigm of communication used in distributed systems. It provides a higher abstraction of communication that allows executing remote operations, methods or procedures in different address spaces. Procedure calls are coded as they were local ones, hiding from the software developers the details about the interaction with remote processes. Instances of remote invocation paradigm are request-reply protocols, *Remote Procedure call* (RPC) [36] and *Remote Method Invocation* (RMI) [37].

Indirect communication. Indirect communication is a high-level abstraction that refers to techniques where senders and receivers are time- and space-decoupled and that is characterized by the presence of an intermediary component placed between the communicating processes. The paradigm is represented by techniques of group communication (the communication is performed via a group abstraction and senders are unaware of receivers), publish-subscribe systems, message queue systems, and shared memory.

In our work, we focus on the approach of distributed shared memory (DSM). This is a memory which is distributed over processes (or nodes of the network) but from the access point of view is considered as a centralized one. DSM allows programmers to access remote data as they were stored locally. Since memory is distributed, an important issue is how to preserve consistency of data while reading, writing and updating them. Thus, each DSM's implementation has to define a consistency model. Most recognized types of DSM are *shared objects*, e.g. *Orca* [38] and *Shasta* [39], and structured DSM, e.g., *Linda model* [40], *Global Arrays* [41] and *PastSet* [42].

2.1.2 Linda model and tuple space

The Linda model [40], proposed by David Gelernter, is a coordination model for parallel and distributed systems. The model is based on high-level operations for synchronization and for exchange of information among different processes sharing data via common repositories named *tuple spaces*. In Linda, a programming model consists of two separate parts: the *computation* model and the *coordination* model. The latter takes on the role of a glue that binds separate activities into an ensemble [43] to obtain a single computational unit.

The Linda model is based on the use of associative access to data meaning that data are accessed not by their addresses but by their content. This paradigm is appropriate for those applications based on a data driven approach.

The Linda interaction model provides time and space decoupling [44], since tuple producers and consumers do not need to know each other. It is beneficial for many scenarios because processes can interact without knowing their names and physical addresses. Indeed, avoiding explicit or fixed names and addresses makes programming more flexible.

The simplicity of this coordination model makes it very intuitive and easy to use. Some synchronization primitives, e.g. semaphores or barrier synchronization, can be implemented easily in Linda (cf. [45], Chapter 3). Because the model is based on shared memory paradigm where all data are assumed and managed as a single data space, the model facilitates implementing some techniques improving data security [46], e.g., access to the tuple space and the monitoring of performed operations.

Nevertheless, there are several limitations that prevent tuple spaces to be widespread as, for instance, message passing. Some of the limitations, like security vulnerability and absence of the mechanism for fault tolerance, are common to other paradigms and they are provided just on implementations of tuple spaces that are focused on addressing these particular issues. The scalability problem is a more fundamental one and difficult to be solved [47, 48]. One of the main reasons is the atomicity of Linda's operations and, specifically, the presence of the operation that removes data. This operation introduces a problem of data consistency and prevents tuple spaces to be easily scalable.

Operation on tuple space. Linda model deals with data in forms of tuples. A tuple is a finite ordered sequence of elements. The length of tuples can vary and fields of tuples can be either *formal* (referred also as *wildcard*) or *actual*. Actual fields contain values of different data types. Formal fields are used to define patterns (patterns are sometimes called *templates*) meaning that certain values of these fields are not specified. Tuples to be inserted can in the tuple space have only actual fields, whereas templates can contain both actual fields and formal fields. For instance, the tuple ⟨"lamp", "status", "on"⟩ consists of three actual fields (strings); the template ⟨"lamp", "status", _⟩ consists of three elements such that the values of the first two of them are "lamp" and "status" respectively and the third field can have any value.

The typical pattern matching algorithm matches tuples and templates if:

1. both elements have the same number of fields;
2. the actual field in the template is equal to the values in the tuple;
3. the types of all formal fields equals the type of the proper values.

The operations provided by the Linda model are:

- **out**($tuple$), inserts a tuple into the tuple space;
- **in**($template$), withdraws from the tuple space a tuple matching the given template while keeping note of the matched values. If there is no matching tuple the process executing the operation blocks until such a tuple is detected.
- **rd**($template$), behaves like **in** but it does not remove the matched tuple from the tuple space;
- **eval**($expression$), spawns a new process to evaluate the expression and inserts the result of the evaluation into the tuple space.

As mentioned above, **in** and **rd** are blocking operations meaning that they keep waiting for the wanted tuple.

Figure 4 illustrates an example of a tuple space that contains different structured values. The tuple $\langle$"lamp", "status", "on"$\rangle$ is produced by the operation `out`($\langle$"lamp", "status", "on"$\rangle$), and it can be read with the operation `rd`($\langle$"lamp", "status", $x\rangle$) after pattern-matching. In this case, the reading process checks all tuples in the tuple space consisting of three elements containing the strings "lamp" and "status" in the first two fields and assigns the third value of one of the matching tuples to variable x. The withdrawal operation `in`($\langle$"room", _, _$\rangle$) consumes (atomically retracts) one of two tuples that has the string "room" as the first field and leaves the other one in the tuple space. This `in` operation can consume either tuple $\langle$"room", 1, "lamp"$\rangle$ or $\langle$"room", 2, "tv"$\rangle$ since both are matched by the template. The choice of the tuple to be "consumed" is nondeterministic and would depend on the specific implementation of pattern matching.

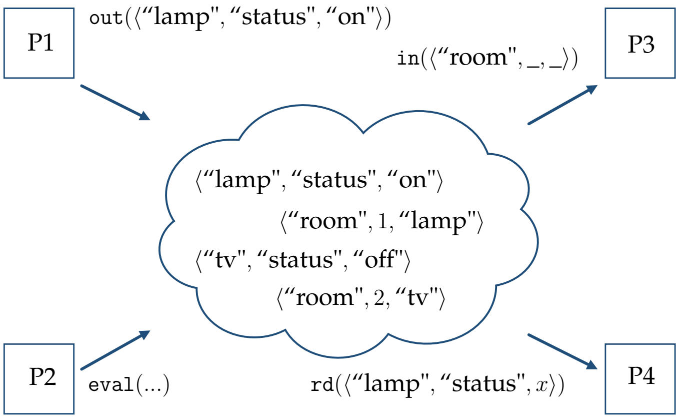

Figure 4: An example of tuple space

Some implementations extend the original set of operations on the tuple space with non-blocking variants of querying operations `rdp` and `inp` that are similar to the blocking operations `rd` and `in` but do not block the executing process [45]. These operations look up a tuple in the tuple space and, if the tuple is found, assign values to variables in the template and return a positive result. If no tuples are found, these operations return a negative result.

Distributed tuple spaces. From the application point of view, tuple spaces can be either managed by a single process or split among different processes [49]. The first case is when a centralized tuple space (see Figure 5) is used that is when one process maintains a tuple space and other processes make requests to it. The second case is when each node maintains its own tuple space (see Figure 6).

In this last case, we distinguish between *local* and *remote* tuple spaces. Regardless of the fact that processes run on a single or on different machines, we consider remote a tuple space that is instantiated/managed by processes different from the one that performs operations (read, write, withdraw) on it. We instead consider local a tuple space that is accessed by the same process that instantiated it. As shown in Figure 6, for process

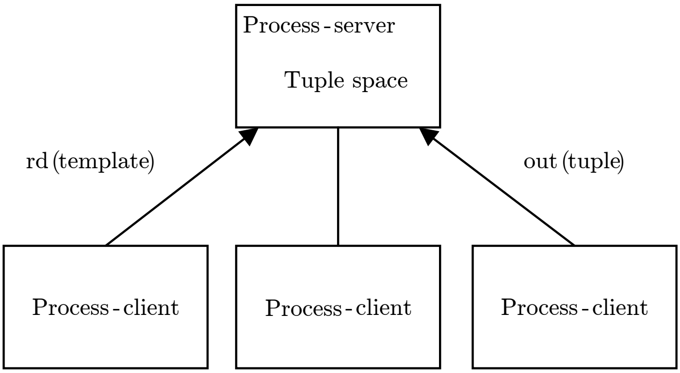

Figure 5: Centralized tuple space

A the tuple space *A* is local, whereas the tuple space *B* is remote.

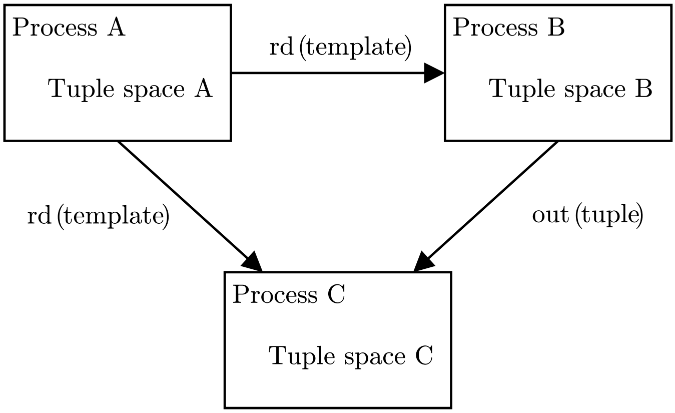

Figure 6: Distribution of tuple spaces

2.2 Data structures

The performance of tuple spaces depends on different aspects. One of them is the design of a concrete instance of the tuple space that resides in a process. In this section, we focus on data structures and techniques that are used while designing tuple spaces and consider the way pattern matching is implemented and the techniques used to improve the performance of the tuple space when there are many concurrent processes.

2.2.1 Data structures

In what follows, we discuss the data structures and techniques that have been used for implementing, manipulating and querying tuple spaces. We pay particular attention to the computational complexity of operations. We recall that $O(-)$ represents a complexity upper bound, $\Omega(-)$ a lower bound and $\Theta(-)$ a tight bound [50].

Vectors. A vector is a random access data structure similar to an *array* with the difference that a vector can dynamically change its size. The computational complexity for adding an element at the end of a vector and accessing any element via its index is $\Theta(1)$. If an insertion is performed at a specific position of the vector the time is $O(n)$ where n is the number of elements stored in the vector. The removal of any element of a vector costs $O(n)$. In order to find an element, it is necessary to check sequentially all elements before the required one, hence searching an element takes linear time $O(n)$. Vectors have been used for the implementation of a distributed variant of Linda, known as KLAIM [51].

A tuple space built on a vector has a very fast writing time $\Theta(1)$, and a slow search time being it proportional to the number of the tuples. If we suppose that all tuples in the tuple space have the same dimension m (number of fields) the search time is $O(nm)$.

Linked lists are sometimes used as variants of vectors. They are similar to vectors but offer a constant time $\Theta(1)$ to remove an element, when its position is known. This renders withdrawing slightly faster because we can first search for an element in the tuple space and then delete it. It is evident that using a vector-based tuple space in an application in which writing operations are much more than reading ones has a positive impact on performances.

Hash tables. A hash table is a data structure which implements an associative array mapping keys to values. It performs very well on reading operations when each element has a unique key; it requires $O(1)$ time for insertion and deletion and $\Theta(1)$ time for lookup in the best case, but because of collisions (when different values share the same key) the lookup time can grow to $O(n)$.

The general idea of implementing tuple spaces via hash tables is the following. For a write operation, we need to compute a hash value of a tuple and place it in the appropriate slot. The time required for this operation depends on the chosen hash function and on the data. The hash function should have a wide range and map inputs as evenly as possible (e.g., co-domain as big as the domain). To avoid collisions *separate chaining* or *open addressing* can be used. The first method, separate chaining, permits associating a collection of elements (e.g. vector, linked list) with a single slot corresponding to a given key. The second method states that whenever the slot of the hash table turns out to be used after probing it is necessary to look for an unused one where to place the data. The performance of two methods for collision resolution depends on the data and on the chosen hash function.

Performances of hash-based tuple spaces are heavily affected by the type of stored tuples. Hashing guarantees fast search time in most of the cases but, if compared with vector-based implementations, it requires more time for tuples insertion. Another drawback is that, to have the fastest search time, by relying on efficient, ad-hoc, hash functions, it is necessary to know a priori all the templates the application will use. In fact, different templates, with different wildcards and values, can be used to match the same tuple. This implies that when we add a tuple and compute its hash value, we need to take into account all kind of templates that could match that specific tuple.

Implementations such as LUATS [14] and TUPLEWARE [13] are based on hash tables but only certain fields are used to calculate the hash value. In LUATS all the tuples are augmented by a field, the first one, storing the key of the slot to which they belong. This solution is similar to the idea of *partitioning* of tuple space, that will be discussed later on. On the contrary, TUPLEWARE uses the first two or three fields of each tuple, depending on the number of its fields, to compute its hash value.

Trees. *Self-balancing binary search trees* are binary search trees that automatically adjust their height after insertion and deletion operations, to be as small as possible. The implementation can be based on such trees as *AVL trees* or *red black trees* [50]. Insertion, deletion and search operations take $O(log\ n)$ in both the best and the worst cases. Querying a tree-based implementation using different templates implies the same problem as in the hash-based case, that we have discussed previously.

2.2.2 Speed-up techniques

Metadata can be built on top of existing tuple spaces in order to improve their reading time. This is the case for *indexing*, a special data structure that collects and maintains metadata about tuples and *partitioning* where "similar" tuples (usually similarity is at the level of their type) are stored in the same data structure. Other speed-up techniques are concerned with multiple pending blocking operations and with concurrent accesses. Below, we discuss how these techniques can be implemented.

Indexing. In database theory *an index* is a data structure that improves the speed of data retrieval but at the same time requires additional storage space and additional computation and maintenance time, see, e.g., [52], Chapter 14. When indexing is used, each tuple has to have a unique identifier and the index is an associative array that associates to the fields of a general pattern the actual values paired with the set of identifiers of tuples containing that value in that field.

When modifying and querying indexing-based tuple spaces most of the time is spent on processing indexes. Writing and reading times depend on the number of indexes used: if k indexes are used, the writing time is $O(k)$ and the reading time varies from $\Omega(1)$ to $O(k)$.

Indexing-based tuple spaces are similar to those based on *hash tables* because the structure of each index looks like a hash table where data with the same value are grouped in one slot. However, the former offers a more general way of querying with the option of flexible searches that allows one to use different templates on the same dataset. Obviously, indexing could be implemented also with hash tables, but this would require computing a hash value for each template. Indexing guarantees a searching time that is as fast as the one offered by hash tables. In addition, since for each tuple's type, there is a certain set of indexes, it is possible to partially parallelize querying. However, maintaining indexing structures requires more time and space and, in utmost cases, the additional size can be comparable with that of the data; e.g., when the data in all tuple fields are different.

Indexing is exploited in GIGASPACES[16] and MOZARTSPACES[9]. Both of them use objects to represent tuples and programmers can select the fields of the tuple to be indexed by using annotations.

Partitioning. A technique used to reduce searching time in tuple spaces containing a high number of tuples is to group them into smaller spaces according to specific *heuristics*. A tuple space can be partitioned according to the nature of a tuple: number of fields, tuple structure, fields type and so on. An example of partitioning can be found in [17] where they introduce the notion of *homogenization* of a tuple space and divide it into several parts according to the type of tuple.

This technique has several advantages: a smaller number of tuples per container (more balanced), less time for type comparison, better granularity (e.g. at the level of the partitions) of lock/synchronization for parallel processing. The performance depends on data distribution and chosen criteria of partitioning. When a tuple space has to store tuples of many different types, partitioning decreases the search time that is especially important for the tuple spaces based on data structures, like vectors, which require non-constant search time. In this case, it is not necessary to spend time for type verification because the type is verified just once when the partition is chosen. Moreover, partitioning allows for parallel processing of different partitions.

Speeding up blocking operations. A naive approach to deal with blocking `rd` and `in` operations is to require the whole tuple space to be searched whenever a new tuple is inserted. However, such an approach may cause spending the time to check whether previously existing (and ruled out) tuples can be matched. A more efficient solution would require checking only newly added tuples. In some implementation, each process waiting for a matching tuple *subscribes* for updates and is notified whenever a new tuple is added to the tuple space. Obviously, this technique turns out to be very efficient only when tuples are not frequently added.

Parallel tuple space access. The use of concurrent data structures is another technique to make a tuple space more suitable to a high number of concurrent accesses. However, it is not immediate that allowing concurrent accesses to the data structure will boost the performance of the application. Indeed, one crucial part is to define the right *lock granularity* of the data structure in order to avoid a global lock leading to accesses sequentialisation.

Another way to parallelize operations on tuple space is to analyze the request to a tuple space and execute in parallel the operations that do not modify the tuple space (e.g., `rd` operations) and perform the other operations sequentially.

Considering the works on concurrent data structures and excluding vector-like ones, it is worth to note that concurrent hash tables are used more (see, e.g. [53], Chapter 47) than concurrent trees. Achieving a good level of parallelization with trees is much more problematic because it is difficult to avoid a global lock for operations that update the tree (insertion and removal of a node). For example, *relaxed red-black trees* [54] do not use tight coupling between updates and rebalancing in trees. Instead, they postpone the rebalancing and perform urgent updates first. In [55] it is reported that these trees guarantee a significant gain with respect to the strictly balanced red-black trees and allow to exploit the benefits of the trees even when they are not completely balanced.

DHT. DHT [56] is a decentralized distributed system for storing data in the format of pairs (key, value). Usually, the lookup complexity in DHTs is $O(log\ n)$ where n is the number of nodes. DHTs have inbuilt mechanisms of replication and guarantee properties such as autonomy, decentralization, fault tolerance, and scalability. However, in spite of these properties, there are limitations that prevent building tuple spaces on top of DHTs. Similarly to ordinary hash tables, one of the main drawbacks is that DHTs limit expressiveness of pattern matching described in the Linda model and prevent preserving the original structure of data. Some works, like, e.g., DTUPLES [15] and [57] adapt DHTs to the paradigm of tuple spaces. DTUPLES uses the first field of the tuple to determine the node of the network where data have to be placed. Thus, the pattern matching is limited by matching the first field. In [57], the stored value is not just a single data, but a container, a separate data space, while the name of the container is used as a key. In this case, the container is an ordinary tuple space that does not have any restriction on templates used for data retrieval.

2.2.3 Discussion

Programmability. One of the challenges in implementing tuple spaces with hash tables or trees is how to identify a tuple within the data structure. For binary trees, it is the designation of non-root nodes as left or right child. For hash tables, it is the computation of the key for each slot. This requires the designer of the tuple space to choose how to adapt these data structures in order to use them with the Linda model. Possible solutions are:

- To limit the number of fields to be considered. In TUPLEWARE, the initial *n* fields of a template never contain wildcards and provide information to find a required tuple. For TUPLEWARE, this is a design choice since the aim of this tuple space is to improve performance for applications that use array-based data.
- To specify the templates that will be used for querying. In the case of hash tables, for each type of templates (templates with different

positions of wildcards) values of different fields are used for computing the hashes. In the case of binary trees, for each predefined type of templates, a separate tree has to be built.

Considering the tuple space with indexing, the latter can be applied either to all fields or to some of them. In the first case, no additional work for programmers is needed, whereas in the second case, it is necessary to specify these fields (e.g., using field's annotation as in GIGASPACES).

Comparison. Trees, indexing, and hash tables can provide nonlinear (to the number of elements) search time. According to the computational complexity of operations, tuple spaces based on hash tables do outperform others. Comparing indexing and hash tables, indexing provides more flexible control, especially, if the tuple space is actively used as a database meaning that many different templates are used to query data.

The performance of blocking operations can be improved in several ways: e.g. performing a repeat search only on newly added data or using concurrent data structures. Both of them do not necessarily improve the performance. So, the first one is beneficial when the tuple space does not change much over time and is used mostly for data retrieval.

Partitioning and concurrent data structures can improve the performance of tuple spaces allowing processes to access a tuple space in parallel. Some concurrent data structures use partitioning in their internal implementation. For example, in Java JDK's *ConcurrentHashMap* the number of partitions that can be accessed concurrently is a parameter and can be set while instantiating. However, concurrent data structures can perform worse than the ordinary ones when they are misused and can slow down the tuple space. Hash tables are well-studied data structures [58] and there are many implementations that provide a good level of parallelization, see e.g., *Threading Building Blocks* [59] and *Folly*, the Facebook Open Source Library [60, 61].

2.3 Implementations

In this section, we review a number of implementations of tuple spaces highlighting their main features. Then, we compare the different implementations. It is worth noting, that we decide to not consider implementations of tuple spaces that provide replication to increase data availability. We postpone their description to Section 2.4, there we talk about replication.

While describing implementations of tuple spaces, we pay attention to the following criteria which can be divided into two groups. The first group contains criteria that we consider fundamental for any tuple space system:

`eval` **operation** This criterion denotes whether the tuple space system has implemented the `eval` operation and, therefore, offers the possibility of code mobility. It is worth mentioning that the original `eval` operation was about asynchronous evaluation and not code mobility, but in the scope of a distributed tuple space, it makes programming data manipulation more flexible.

Tuples clustering This criterion determines whether some tuples are grouped by particular parameters that can be used to determine where to store them in the network.

Absence of domain specificity Some implementations have been developed having a particular application domain in mind. On the one hand, this implies that domain-specific implementations outperform the general purpose one, but on the other hand, this can be considered as a limitation if one aims at generality.

Security This criterion specifies whether an implementation has features to control access to tuples. For instance, a tuple space can require authorizations and regulate the access to its tuples, limiting the specific operations (e.g. only writes or read).

The second group of criteria gathers features which are desirable for any fully distributed implementation that runs over a computer network, does not rely on a single node of control or management and is scalable. We did not consider fault tolerance as an evaluation criterion because we think this a somewhat orthogonal issue.

Distributed tuple space This criterion denotes whether tuple spaces are stored in one single node of the network or they are spread across the network.

Decentralized management Distributed systems rely on a node that controls the others or the control is shared among several nodes. Usually, systems with the centralized control have bottlenecks which limit their performance.

2.3.1 Tuple space systems

Since the first publication on Linda, there have been a plenty of implementations of its coordination model in different languages. Our purpose is to review the most significant and recent ones, that are possibly still maintained, avoiding toy implementations or the one short paper implementations. However, we do not consider tuple space based middleware specifically devised for ad-hoc wireless networks [62] or for agent-based programming [63]. Based on these considerations, we have chosen: JavaSpaces [64] and TSpaces [65] which are two industrial proposals of tuple spaces for Java; GigaSpaces [16] which is a commercial implementation of tuple spaces; Tupleware [13] featuring an adaptive search mechanism based on communication history; Grinda [66], Blossom [17], DTuples [15] featuring distributed tuple spaces; LuaTS [14] which mixes reactive models with tuple spaces; MozartSpaces [9
and Klaim [51] which are two academic implementations with a good record of research papers based on them.

Blossom. Blossom [17] is a C++ implementation of Linda which was developed to achieve high performance and correctness of program codes. In Blossom all tuple spaces are homogeneous with a predefined structure that demands less time for type comparison during the tuple lookup.

BLOSSOM was designed as a distributed tuple space and can be considered as a distributed hash table. To improve scalability each tuple can be assigned to a particular place (a machine or a processor) on the basis of its values. The choose the place where a certain tuple should be located the following mechanism is used: every tuple is associated with the access pattern that determines which fields have to contain values (also for templates); based on the values of these fields it is possible to compute an identifier of the tuple's location. Conversely, using the data from the template, the exact place where a required tuple is potentially stored can be found. Prefetching allows a process to send an asynchronous (i.e. non-blocking) request for a tuple and to continue its work while the search is performed. When the requested tuple is needed, if found, it is received without waiting.

TSPACES. TSPACES [65] is an implementation of the Linda model developed at the IBM Almaden Research Center. It combines asynchronous messaging with database features. TSPACES provides a transactional support and a mechanism of tuple aging. Moreover, the embedded mechanism for access control to tuple spaces is based on access permission. It checks whether a client is able to perform specific operations in the specific tuples space. Pattern matching is performed using either standard `equals` method or `compareTo` method.

KLAIM. KLAIM [51] (A Kernel Language for Agents Interaction and Mobility) is an extension of Linda supporting distribution and processes mobility. Processes, like any other data, can be moved from one locality to another and can be executed at any locality. Klava [67] is a Java implementation of KLAIM that supports multiple tuple spaces and permits operating with explicit localities where processes and tuples are allocated. In this way, several tuples can be grouped and stored in one locality. Moreover, all the operations on tuple spaces are parameterized with a locality. The emphasis is put also on access control which is important for mobile applications. For this reason, KLAIM introduces a type system which allows checking whether a process is allowed to perform specific operations at specific localities.

JAVASPACES. JAVASPACES [64] is one of the first implementations of tuple spaces developed by Sun Microsystems. It is based on a number of Java technologies (e.g., Jini and RMI) and has been recently integrated into the Apache River project. Like TSPACES, JAVASPACES supports transactions and a mechanism of tuple aging. A tuple, called entry in JAVASPACES, is an instance of a Java class and its fields are the public properties of the class. This means that tuples are restricted to contain only objects and not primitive values. The tuple space is implemented by using a simple Java collection. Pattern matching is performed on the byte-level, and the byte-level comparison of data supports object-oriented polymorphism.

GIGASPACES. GIGASPACES [16] is a contemporary commercial implementation of tuple spaces. Nowadays, the core of this system is GIGASPACES XAP, a scale-out application server; user applications should interact with the server to create and use their own tuple space. The main areas where GIGASPACES is applied are those concerned with big data analytics. Its main features are linear scalability, optimization of RAM usage, synchronization with databases and several database-like features such as complex queries, transactions, and replication.

LUATS. LUATS [14] is a reactive event-driven tuple space system written in Lua. Its main features are the associative mechanism of tuple retrieving, fully asynchronous operations and the support of code mobility. LUATS provides centralized management of the tuple space which can be logically partitioned into several parts using indexing. LUATS combines the Linda model with the event-driven programming paradigm. This paradigm was chosen to simplify program development since it allows avoiding the use of synchronization mechanisms for tuple retrieval and makes more transparent programming and debugging of multi-thread programs. Tuples can contain any data which can be serialized in Lua. To obtain a more flexible and intelligent search of tuples, processes can send to the server code that once executed returns the matched tuples. The reactive tuple space is implemented as a hash table, in which data are stored along with the information supporting the reactive nature of that tuple space (templates, client addresses, callbacks and so on).

MozartSpaces. MozartSpaces [9] is a Java implementation of the space-based approach [68]. The implementation was initially based on the eXtensible Virtual Shared Memory (XVSM) technology, developed at the Space Based Computing Group, Institute of Computer Languages, Vienna University of Technology. The basic idea of XVSM is related to the concept of *coordinator*: an object defining how tuples (called entries) are stored. For the retrieval, each coordinator is associated with a *selector*, an object that defines how entries can be fetched. There are several predefined coordinators such as FIFO, LIFO, Label (each tuple is identified by a label, which can be used to retrieve it), Linda (corresponding to the classic tuple matching mechanism), Query (search can be performed via a query-like language) and many others. Along with them, a programmer can define a new coordinator or use a combination of different coordinators (e.g. FIFO and Label). MozartSpaces provides also transactional support and a role-based access control model [69].

DTuples. DTuples [15] is designed for peer-to-peer networks and based on *distributed hash tables* (DHT), a scalable and efficient approach. Key features of DHT are autonomy and decentralization. There is no central server and each node of the DHT is in charge of storing a part of the hash table and of keeping routing information about other nodes. As the basis of the DTH's implementation DTuples uses FreePastry[1]. DTuples supports transactions and guarantees fault-tolerance via replication mechanisms. Moreover, it supports multi tuple spaces and allows for two kinds of tuple space: *public* and *subject*. A public tuple space is shared among all the processes and all of them can perform any operation on it. A subject tuple space is a private space accessible only by the processes that are

[1]FreePastry is an open-source implementation of Pastry, a substrate for peer-to-peer applications (http://www.freepastry.org/FreePastry/).

bound to it. Any subject space can be bound to several processes and can be removed if no process is bound to it. Due to the nature of DHTs, the expressiveness of the pattern matching is limited, since only the first field of the tuple participates in matching. Value of the first field is used to identify a node of the DHT where the tuple has to be stored.

GRINDA. GRINDA [66] is a distributed tuple space which was designed for large scale infrastructures. It combines the Linda coordination model with grid architectures aiming at improving the performance of distributed tuple spaces, especially with a lot of tuples. To boost the search of tuples, GRINDA utilizes spatial indexing schemes (X-Tree, Pyramid) which are usually used in spatial databases and Geographical Information Systems. Distribution of tuple spaces is based on the grid architecture and implemented using structured P2P networks (based on Content Addressable Network and tree-based).

TUPLEWARE. TUPLEWARE [13] is specially designed for array-based applications in which an array is decomposed into several parts each of which can be processed in parallel. It aims at developing a scalable distributed tuple space with good performances on a computing cluster and provides simple programming facilities to deal with both distributed and centralized tuple space. The tuple space is implemented as a hash table, containing pairs consisting of a key and a vector of tuples. Since synchronization lock on Java hash table is done at the level of the hash element, it is possible to access concurrently to several elements of the table. To speed up the search in the distributed tuple space, the system uses an algorithm based on the history of communication. Its main aim is to minimize the number of communications for tuples retrieval. The algorithm uses *success factor*, a real number between 0 and 1, expressing the likelihood of the fact that a node can find a tuple in the tuple space of other nodes. Each instance of TUPLEWARE calculates success factor on the basis of previous attempts and first searches tuples in nodes with greater success factor.

2.3.2 Comparison

	JSP	**TSP**	**GSP**	**TW**	**GR**	**BL**	**DTP**	**LTS**	**KL**	**MS**
`eval` operation								✓	✓	
Tuple clustering			?	✓				✓		
No domain specificity	✓	✓	✓		✓	✓	✓		✓	✓
Security		✓	✓						✓	
Distributed tuple space			?	✓	✓	✓	✓		✓	✓
Decentralized management				✓	✓		✓	✓	✓	✓
Scalability			✓	✓	✓		✓			

JAVASPACES (**JSP**), TSPACES (**TSP**), GIGASPACES (**GSP**), TUPLEWARE (**TW**), GRINDA (**GR**), BLOSSOM (**BL**), DTUPLES (**DTP**), LUATS (**LTS**), KLAIM (**KL**), MOZARTSPACES (**MS**)

Table 1: Results of the comparison

We evaluated described earlier tuple spaces using criteria listed at the beginning of the section. Table 1 summarizes the result of our comparison: ✓ means that the implementation enjoys the property and ? means that we were not able to provide an answer due to the lack of source code and/or documentation.

After considering the results in Table 1, we are going to perform detailed experiments that are presented in Chapter 4 with the following tuple spaces:

- TUPLEWARE because it enjoys most of the wished features;

- KLAIM because it offers distribution and code mobility;
- MOZARTSPACES because it satisfies two important criteria of the second group (fully distribution) and is one of the most recent implementations.
- GIGASPACES because it is the most modern among the commercial systems; we will use it as a yardstick to compare the performance of the others.

We would like to add that DTUPLES has not been considered for the more detailed comparison because we have not been able to obtain its libraries or source code and that GRINDA has been dropped because it seems to be the less maintained one. It is worth noticing that while GIGASPACES and MOZARTSPACES are still maintained, the other implementations are not, and their code is based in the state-of-the-art technology of their last release, which nowadays could result *deprecated*.

2.4 Replication

In this section, we discuss how replication is exploited to improve data availability. In particular, we focus on applications of replication to tuple spaces. First, we describe and discuss tuple spaces where replication is applied. Then, we talk about approaches that are used to optimize replica placement.

2.4.1 Tuple space's implementations with replication

DEPSPACE. DEPSPACE [19] is a Byzantine fault tolerant coordination service[2] that provides a tuple space abstraction whose main emphasis is fault tolerance and security. DEPSPACE uses different replication approaches such as Byzantine fault tolerant state machine replication [19] and Byzantine quorum systems [70]. Security is achieved by access control mechanism and cryptography. Servers create a dependable tuple space

[2]Byzantine fault tolerance implies that the architecture with at least 3n+1 servers can tolerate failures of n servers

enforcing reliability, availability, integrity and confidentiality of data, and tolerating Byzantine faults [19]. The DEPSPACE project is still ongoing and in a recent work [71] it has been enhanced with extensible coordination services, e.g., the possibility of adding custom operations to be executed on the server side. These services are somehow similar to the original *eval* primitive and aim at providing mechanisms to have tuples close to data processing units to guarantee fast data access. However, because of security and fault tolerance requirements, an extension has to satisfy a number of constraints in order to be executed, whereas *eval* operation has no specific limitations.

GSPACE. GSPACE [72] is an implementation of the distributed data space that uses self-adaptation to increase data availability.

In [72] the self-adaptation is based on the analysis of the numbers of reads and writes performed by a certain node and deals with potential failures of nodes where application components reside. According to the aforementioned metrics, a certain replication policy among the following ones has to be automatically chosen: *Full Replication*, *n-Fixed Replication* (tuples are replicated to a certain number of nodes), *Dynamic Consumer Replication* (tuples are replicated to nodes that consume tuples of the same type), *Dynamic Producer Replication* (tuples are replicated to nodes which produce tuples of the same type).

Another version of GSPACE [73] focuses on replication and self-adaptation that optimizes replica placement according to such parameters of the nodes as their memory usage and bandwidth.

LIME. LIME [74] is a distributed tuple space for mobile ad hoc networks. A version of LIME [75] with replication was designed to guarantee data availability by relying on replication profiles, data that are stored in each node and that specify what and how tuples should be replicated.

Applications use different *replication profile* to separate data of each application from each other and decrease the size of data to be replicated. Replication profiles tell applications what tuples to replicate: it contains templates specifying tuples to be replicated and names of hosts from which tuples should be replicated. The application replicates a tuple when it receives a tuple with the same profile.

Another feature of LIME is how it deals with consistency. LIME distinguishes two types of tuples (the original copy and a replica) and allows to specify *replication mode* and *consistency mode*. Replication mode specifies how tuples should be replicated: only from a master tuple or also from its copies. Consistency mode guides how to update replicas (never, only from the master, from any newer version). LIME delegates control over replication to programmers letting them choose aforementioned parameters of tuples and apply them in applications.

REPLIKLAIM. REPLIKLAIM [76] is a language based on KLAIM that enriches it with primitives for replica-aware coordination. Two types of consistency are considered in REPLIKLAIM: weak and strong. Replicating tuples, programmers have to decide by themselves where to put data and REPLIKLAIM is responsible for keeping a tuple space in a consistent state.

In REPLIKLAIM the typical tuple space operations have a different meaning:

- operations $\mathtt{out}_\mathtt{w}(\mathtt{t})$ $\mathtt{out}_\mathtt{s}(\mathtt{t})$ are used to replicate a tuple `t` on the localities specified as a further argument using weak and strong consistencies correspondingly;
- operation $\mathtt{in}_\mathtt{w}(\mathtt{t})$ and $\mathtt{in}_\mathtt{s}(\mathtt{t})$ withdraw a tuple and removes all its copies from the localities where it was replicated;
- the reading operation `read(t)` remains the same and is not indexed by anyconsistency level;
- the mobility operation `eval(P)` is not considered in REPLIKLAIM.

Short discussion. Apart from the tuple spaces listed above, replication is exploited also in GIGASPACES, PEERSPACES [7], PLINDA [22], FT-LINDA [21]. It is used for different purposes; some works (DEPSPACE, PLINDA, FT-LINDA) use replication to guarantee a specific level of fault tolerance, others works exploit it to increase data availability.

The use of operations with strong consistency prevents tuple spaces to scale when the number of nodes grows. To improve scalability two approaches are exploited: to relax consistency requirements and to limit the number of nodes where data has to be stored.

The first one uses different consistency models to relax consistency requirements: REPLIKLAIM offers the possibility of using weak (eventual) consistency for withdrawing operations; LIME exploits predefined policies to determine how replicas should be created and updated. Another option is to use replication but avoid operations of data removal that require updating replicas. For instance, in PEERSPACES [7], to guarantee availability, replication is allowed only for read-only data on which withdrawing operations are not possible.

The second approach is to share certain information not with the whole network but with only nodes that need it decreasing the overall amount of information circulating in the network. So, LIME and GSPACE uses techniques that tell nodes how and where replicate tuples. Similarly, in TOTA [8], a tuple contains information that is enough to determine how tuples should be cloned to nearby nodes and maintained.

At the same time, most of the tuple space systems delegate to the programmer the management of replica creation and data placement. In this way, a fine-grained control is possible by carefully choosing nodes where to put data.

Strategies for replica placement Where to replicate data is another important problem because of its impacts on performance. There are two issues that replica placement mechanisms have to consider: what information to use for determining where to place replicas and how to do it efficiently. The following data are frequently considered for replicas placement:

- data popularity: applications collect information about how many times a specific resource or file is requested. Then, resources with higher popularity are replicated in more/better nodes of the network to reduce the average access time [28].

- network information: different informations such as cluster-aware placement, degree-aware placement, distance-aware placement [20, 24, 25], betweenness-centrality [77] placement, closeness-centrality [78] placement, greedy algorithms [79] are exploited. This information is useful to reduce the average data access time in the network.

- characteristics of nodes: their bandwidth, the available physical memory, their computational power [26]. Nodes of the network that have better characteristics are usually considered as better ones to place replicas.

- the activity of network nodes: Data should be placed closer to the nodes that need these data, see e.g., *Hot-Spot* [27]. .

The second issue is how to efficiently determine the locations where to put data. This is the main requirement for large networks. Usually, fast placement computing and high precision of placement cannot be achieved simultaneously and it is required to choose which characteristic is preferable for the system. Replication is a standard technique [29] in CDN, however, for CDN the speed of the algorithm for replica placement is not very important. For real-time systems, it is acceptable to sacrifice high precision to be able to work over larger networks. Among techniques to decrease the complexity of placementâĂŹs computing, it is worth mentioning the following two works. The authors of [30] propose to use a decentralized summary of recent accesses that decreases the amount of information to be processed and guarantees near-optimal performance. In [25] it is instead proposed a two-step approach that, using partial information about latencies in the network, selects a network region where replica should be placed and, then, determines the specific node.

Chapter 3

A new implementation of KLAIM

In this chapter, we focus on the programming language KLAIM, which has been already briefly described in Chapter 2. KLAIM is a coordination language and a process calculus that extends the original Linda primitives with explicit information about the location of the nodes where processes and tuples are located.

The organization of this chapter is as follows. In Section 3.1, we present KLAIM by describing its syntax and semantics. Section 3.2 describes how KLAVA, a Java implementation of KLAIM, can be used to build distributed applications exploiting the tuple space paradigm to connect and coordinate their components. The description focuses mostly on the main classes of the API aiming at showing how to use it by means of code snippets in Java. In the last section (Sect. 3.3), we discuss the main changes that we have introduced to make KLAVA more efficient.

3.1 KLAIM language

In this section, we talk about the syntax and the semantics of KLAIM.

3.1.1 Process syntax

The process syntax of KLAIM presented in Table 2 includes the description of processes, actions, and tuples (and templates). The following syntactic categories are exploited:

- Loc (l) is a set of localities.
- $VLoc$ (u) is a set of locality variables.
- Val (v) is a set of basic values.
- Var (x) is a set of value variables.
- Exp (e) is the category of value expressions.
- Ψ (A) is a set of parameterized process identifiers.
- χ (X) is a set of process variables.

In KLAIM, processes are built of a set of operators that similar to the ones in CCS [80]. The operators for building processes are the following: `nil` stands for the inactive process (e.g., a process that cannot perform any action), $a.P$ is a process that first executes the action a and then proceeds like P, $P \mid P$ stands for the parallel composition of two processes, $P + P$ stands for the nondeterministic composition, X is a process variable and $A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$ is a process invocation. Actions are ordinary operations on tuple spaces: `out`, `read`, `in`, and `eval`. Tuples are the sequence of actual and formal fields (with the prefix !).

The syntax of KLAIM specifies also a network of sites. This specification differs from one described in [51] since our variant does not uses the allocation environment that in the original work was used to define a mapping from logical to physical localities. In our case, each node has a single and unique address that can be used to identify it in the network using its address. We shall use the names logical localities, physical localities, and localities interchangeably.

Definition of tuples and templates in KLAIM is similar to the one of original Linda model. Tuples are sequences of (only) actual fields that can

P	$::=$	`nil`	(null process)
		$\mid a.P$	(action prefixing)
		$\mid P \mid P$	(parallel composition)
		$\mid P + P$	(choice)
		$\mid X$	(process variable)
		$\mid A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$	(process invocation)
a	$::=$	$\mathtt{out}(t)@\ell \mid \mathtt{in}(T)@\ell$	
		$\mid \mathtt{read}(T)@\ell \mid \mathtt{eval}(P)@\ell$	(actions)
t	$::=$	$e \mid P \mid \ell \mid !x \mid !X \mid !u \mid t_1, t_2$	(tuples and templates)
N	$::=$	$l :: P \mid N_1 \mid\mid N_2$	(network)

Table 2: Process syntax

be represented as expressions, processes and localities, whereas templates can contain both actual and formal fields that are denoted by $'!v'$ (where v is a generic variable). The evaluation function for tuples $\mathcal{T}[\![t]\!]$ is defined in Table 3. This function is used to evaluate processes, location, tuples and expressions (using an evaluation mechanism $\mathcal{E}[\![t]\!]$).

Pattern-matching is defined by the rules in Table 4.

3.1.2 Informal semantics

In what follows, we just give an informal semantics of KLAIM. The interested reader is referred to [51] for the structural operational semantics of all operators.

The network can be formed by composing localities or networks of smaller size using the composition operator $||$. This allows processes to be

$\mathcal{T}[\![t]\!] = \mathcal{E}[\![e]\!]$	$\mathcal{T}[\![t, t']\!] = \mathcal{T}[\![t]\!], \mathcal{T}[\![t']\!]$
$\mathcal{T}[\![P]\!] = P$	$\mathcal{T}[\![!x]\!] = !x$
$\mathcal{T}[\![l]\!] = l$	

Table 3: Tuple evaluation function

$match(v, v)$	$match(P, P)$
$match(l, l)$	$match(!x, v)$
$\dfrac{match(et_1, et_2)}{match(et_2, et_1)}$	$\dfrac{match(et_1, et_2) match(et_3, et_4)}{match((et_1, et_3), (et_2, et_4))}$

Table 4: Pattern-matching predicates

executed in an interleaving fashion. Each locality provides the ordinary set of operations on the tuple space located there: adding a tuple (`out`), reading (`read`) and withdrawing (`in`) a tuple using the template, and the operation that allows the migration of KLAIM program from one locality to another (`eval`).

Tuple and templates have to be evaluated before they are used to alter or query the tuple space. While evaluating tuples and templates, all expressions that are contained in tuple fields will be computed, except the ones that contain localities and formal fields. Then, the evaluated tuples can be added to the tuple space or used for pattern matching (according to the matching rules). The rules for matching are similar to those of the original Linda model (see Section 2.1).

Process variables supports *higher order* communication. This means that a process can be moved and executed in other locality and process variables can be bound to processes dynamically. For instance, a process can be added to a tuple space and, when it is retrieved, process variables will be bound to this process for later execution.

3.2 Programming with KLAIM

KLAVA is a Java implementation of KLAIM. The framework was originally developed by Lorenzo Bettini and presented in details in [81]. In what follows, we will describe and show by means of examples how the main ingredients of KLAIM (tuples, localities, code mobility and tuple spaces) can be programmed in Java and used to build distributed systems.

It is worth noting that most of the KLAVA API was designed by Lorenzo Bettini and here we show how to use it. Our main modification of the API offers the possibility of choosing the class of the tuple space to use. Other changes that we have done are related to the internal implementation of KLAVA and described separately in Section 3.3.

Defining tuples and templates. A tuple is a finite ordered list of elements. To define a tuple it is required to pass an array of objects to the constructor of the `Tuple` class. As shown in Listing 3.1, first, we create an array that may contain values of different types and then pass use it to initialize a tuple.

```
Object[] array = new Object[]{"array", 0, 10};
Tuple tuple = new Tuple(array);
```

Listing 3.1: Initialization of a tuple

To access a certain field of the tuple it is necessary to call the class method `getItem` with the index of the field (indexes start from 0). This method returns the current value of the field as an object of the `Object` class. To change the value of the field it is necessary to call the class method `setItem` passing the index of the field and a new value. An example of how to use these methods is presented below, where, first, we obtain the value of the counter and, then, increment it and set its update value (see Listing 3.2).

```
Tuple tuple = new Tuple(new Object[]{"counter", 0});
Integer counter = (Integer) tuple.getItem(1);
tuple.setItem(1, ++counter);
```

Listing 3.2: An example with methods `getItem` and `setItem`

Templates are similar to tuples but their fields can be not only actual but also formal. Because values of actual fields in tuples can have different types we need to take into account this information. To indicate that a certain field of the template is formal we assign a name of Java class to this field. The class should correspond to the type of values we want this field to be matched with. It means that KLAVA does not accept *null* values as values of fields. As shown in Listing 3.3, a template is used where the first field is actual and the second field is formal and contains only information about its type.

```
Object[] array = new Object[]{"counter", int.class };
Tuple template = new Tuple(array);
```

Listing 3.3: Initialization of the template

It is also assumed that values of primitive types *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* and *char* are interchangeable with object of classes *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Boolean* correspondingly. It means that for integer value it is possible to assign to a formal field either *int.class* or *Integer.class*.

Tuples matching is implemented as follows. For primitive data types, the equality operator "==" is used to check whether two objects are equal. For reference types, we rely on method `equals`, the standard method for the comparison of two objects. The method should be properly overridden since by default it checks an equality of object references.

Localities and nodes. KLAIM operates with explicit localities. Localities are the tools that processes can use for referring to nodes of the network. As we described in Section 3.1, our version of KLAIM distinguishes only *physical* localities. Physical localities are identifiers through which nodes can be uniquely identified within a net. The distinct locality *self* is used by processes to refer to local localities of their execution nodes.

A physical locality is defined by relying on the `PhysicalLocality` class and its object is initialized with the explicit address of the node. The communication part is based on message passing and implemented using Java NIO [82]. KLAVA uses *TCP/IP* protocol and accepts IP addresses with the format $< IP >:< port >$. TCP/IP protocol provides reliable, ordered, and error-detected delivery of data over unreliable networks from the sender to the receiver (cf. [83], Chapter 6).

The network of localities can be defined in two ways. In the first way, we can use classes `Net` and `ClientNode` and, first, initialize a server node and, then, using its locality's address instantiate client nodes (see Listing 3.4). When clients nodes are instantiated, addresses of client's physical localities will be assigned automatically.

```
PhysicalLocality serverLoc = new PhysicalLocality(ipAddress);
KlavaNode serverNode = new Net(serverLoc);
// define another node
KlavaNode clientNode = new ClientNode(serverLoc);
```

Listing 3.4: Difinition of the network using server and client localities

In the second way, we define all localities separately and, then, pass them to the constructor of the `Net` class as shown in Listing 3.5.

```
Vector<PhysicalLocality> localities = new Vector<>();
 localities .add(new PhysicalLocality(address1));
 localities .add(new PhysicalLocality(address2));
 ...
 localities .add(new PhysicalLocality(addressN));
new Net(localities);
```

Listing 3.5: Difinition of the network using a collection of localities

Operations of data manipulation. The `KlavaNode` class wraps a tuple space and allows to exploit it in networks. It provides all operation of tuple spaces. The operations on tuple spaces follow the semantics of the language:

- `out(t)@l`: adds the tuple `t` to the tuple space located at the location `l`.

- `in(tp)@l`: tries to withdraw a tuple using the template `tp` from the location `l`. If nothing is matched, the operation blocks the process of execution and waits until a tuple that satisfies the template `tp` is matched.

- `read(tp)@l`: the operation is similar to `in(tp)@l` with the difference that it does not remove the matched tuple from the tuple space.

- `in_nb(tp)@l`: non-blocking variant of `in(tp)@l`: tries to withdraw a tuple and if nothing is matched returns `false`. Otherwise, the operation returns `true` and assigns values to variables in the template `tp`.

- `read_nb(tp)@l`: the operation is similar to `in_nb(tp)@l` with the difference that it does not remove the matched tuple from the tuple space.

- `eval(P)@l`: spawns process `P` for execution at `l`.

To perform an operation on a tuple space, it is necessary to call a correspondent method of the node object with necessary parameters. For instance, the writing operation `out` has the method signature presented in Listing 3.6.

```
void out(Tuple t, Locality l)
```

Listing 3.6: Signature of the method `out`

To access a tuple space located in a remote locality it is necessary to know the address of this locality. As shown in Listing 3.7, first, we instantiate an object related to the remote locality to be accessed (line 1). Then, we send a tuple to this locality (line 3).

```
1  PhysicalLocality remoteLocality = new PhysicalLocality(address);
2  Tuple tuple = new Tuple(new Object[]{"counter", 1});
3  node.out(tuple, remoteLocality);
```

Listing 3.7: Performing a remote operation

If we do not specify the location, it is assumed that we perform an operation on a local tuple space, a tuple space that resides in the node. For instance, as shown in Listing 3.8, we insert a tuple ⟨"counter", 1⟩ in the tuple space of the node, i.e. locally.

```
Tuple tuple = new Tuple(new Object[]{"counter", 1});
node.out(tuple)
```

Listing 3.8: Inserting a tuple into the tuple space

Blocking querying operations do not return anything because they either succeed or throw an exception if an error occurs (e.g. exceptions specific to the framework, such as `KlavaConnectException`, or general Java exceptions, such as `RuntimeException`). To read or withdraw a tuple it is necessary to define a template and call a correspondent method of the node. These operations can also be performed either locally or remotely (see Listing 3.9).

```
  // withdraw a tuple locally
Tuple template = new Tuple(new Object[]{"counter", Integer.class});
node.in(template);

// withdraw a tuple from a remote locality
PhysicalLocality remoteLocality = new PhysicalLocality(address);
template = new Tuple(new Object[]{"counter", Integer.class});
node.in(template, remoteLocality);
```

Listing 3.9: An example with the blocking operation `in`

Non-blocking operations of data retrieval return boolean value: `true` if a tuple is found (with the assignment of values to the fields of the template) and `false` otherwise. An example is shown in Listing 3.10 where we read a current value of the counter. If the tuple space contains a tuple matched with the template, its data can be used in a further computation.

```
Tuple template = new Tuple(new Object[]{"counter", Integer.class});
boolean res = node.rd_nb(template);
if (res)
  compute(template.getItem(1));
```

Listing 3.10: An example with the non-blocking operation `rd_nb`

Code mobility. KLAVA supports code mobility meaning that a piece of code can be sent to a certain node and be executed there. In the destination locality, the node creates a new process for executing the received code. The executing process can interact with the tuple space of the destination locality. KLAVA supports *weak mobility* [84] that means only a code with some parameters can be transferred but not the execution state. In contrast, the *full mobility* allows moving a code along with the state of the executing process. Also, KLAVA follows the approach when the code and all necessary classes are delivered together [85].

To implement code mobility it is necessary to use the method `eval` of the node (see Listing 3.11).

```java
void eval(KlavaProcess p, Locality l)
```

Listing 3.11: Signature of the method `eval`

The parameters of this method are the process code `p` to be executed and the locality `l` of the execution. The process code is an object of the class that is inherited from the `KlavaProcess` class. In the inherited class it is necessary to implement method `executeProcess`. This method will be executed on a remote node when the object with the code is delivered to it.

```java
PhysicalLocality serverLoc = new PhysicalLocality("192.168.1.1:6001");
KlavaNode serverNode = new Net(serverLoc);
// define of a client
KlavaNode clientNode = new ClientNode(serverLoc);
KlavaProcess processCode = new KlavaProcess() {
  // implement method executeProcess
  @Override
  public void executeProcess() throws KlavaException {
    // do necessary work
    Object compResult = doComputation();
    // write results
    Tuple tuple = new Tuple(new Object[]{"result", compResult});
    this.out(tuple, this.self);
  }
};
clientNode.eval(processCode, serverLoc);
// wait for results
Tuple template = new Tuple(new Object[]{"result", Object.class});
clientNode.read(template, serverLoc);
```

Listing 3.12: An example of code mobility in KLAVA

To show how to use code mobility lets consider the following scenario. We have two nodes: one is a server node and another is a client one. The client node sends a code to be executed to the server node and waits for the results. The server node executes the code that produces a tuple with the result while its execution. The code of this scenario is presented in Listing 3.12. First, we instantiate the server node and the client node (lines 1-4). Then, we define a process to be executed (lines 5-15) by means of the variable `processCode` that instantiates an anonymous class expression where we extend the abstract `KlavaProcess` class. The client node sends the code of the process to the server node using operation `eval` (line 16) and immediately after begins to wait for the result reading it from the tuple space of the server node (lines 18-19). The defined process does some computation (line 10) and then puts in the local tuple space of the node where it is executed results of the computation (lines 12-13).

Initializing tuple space. The new KLAIM implementation (see more details in Section 3.3) allows configuring tuple spaces based on specified data structures. The implemented one are:

- `TupleSpaceList`;
- `TupleSpaceHashtable`;
- `TupleSpaceTree`;
- `TupleSpaceConcurrentHashtable`;
- `IndexedTupleSpace`.

Each of these classes represents a tuple space with different performance characteristics. By default, the implementation based on linked lists is used and to choose a certain implementation it is necessary to specify a class of the implementation while initializing a node object. For instance, as shown in Listing 3.13, we initialize a node with the tuple space based on trees.

```
KlavaNode clientNode = new ClientNode(serverLoc, TupleSpaceTree.class);
```

Listing 3.13: Initialization of the tuple space

In some cases, a certain implementation of tuple space may require some configuration. That is why the method `addSettings` is provided that allows specifying configuration information for tuple spaces in the form of a pair (key, value) where the key is an identifier of the parameter and the value is a correspondent value. For instance, in Listing 3.14 we report an example how to set parameters for tuple spaces based on hash tables and trees. We assume that an application uses one type of tuples consisting of three `String` fields (lines 1-2), that can be matched by two different templates (lines 4-5): one has a formal field in the second element of the tuple, the other in the third one. Then, we add these data as a setting with the name "$template\_rules$" (lines 7-8). Considering a tuple space based on hash tables, while the execution of this application for each template, two hash values have to be computed and added as keys to the hash table.

```
1  Object[] tupleType = new Object[]{String.class,
2        String.class, String.class};
3  List<Boolean[]> templatePattern = new List<Boolean[]>();
4  templates.add(new Boolean[]{true, false, true});
5  templates.add(new Boolean[]{true, true, false});
6  // add settings to the tuple space
7  Pair templateData = new Pair(tupleType, templatePattern);
8  node.addSettings("template_rules", templateData);
```

Listing 3.14: An example with the method `addSettings`

Defining a custom tuple space. Programmers can define their own implementation of tuple space. For instance, one can code such data structures as a *queue* or a *stack* and use them by means of ordinary operations on the tuple space. To do so, it is required to implement methods of the `ITupleSpace` interface. The main methods of this interface (see Listing 3.15) are operations on tuple spaces and several auxiliary methods.

```
public inteface ITupleSpace {
    abstract void out(Tuple t);
    abstract boolean read(Tuple t);
    abstract boolean in(Tuple t);
    abstract boolean read_nb(Tuple t);
    abstract boolean in_nb(Tuple t);
    abstract void addSettings(String key, Object settings);
    abstract void removeAll();
    abstract void stop();
}
```

Listing 3.15: `ITupleSpace` interface

Methods `out`, `read`, `in`, `read_nb`, `in_nb` stand for common operations on the tuple space. Method `removeAll` is used to delete all tuples from the tuple space and method `stop` is used to stop all processes that might be used while the tuple space works. Method `addSettings` allows passing to the tuple space additional information to configure it (see Listing 3.14).

3.3 A new KLAIM's implementation

We improved the implementation of KLAIM by enhancing the communication part and its pattern matching. Regarding the communication part, we changed the module of KLAIM that is responsible for sending and receiving tuples. Previously, it was based on *Java IO*, the package containing classes for the data transmission over the network. For the renewed part, we have opted for *Java NIO* [82], non-blocking input/output (IO), which is a modern version of IO and in some cases allows using resources more efficiently. Java NIO is beneficial when used to program applications dealing with many incoming connections. Moreover, for synchronization purposes, we used a more recent package (java.util.concurrent) instead of synchronization methods of the previous generation. Tuning parameters of sockets makes possible to achieve faster transmission of large chunks of data.

Additional work was done to improve pattern matching. KLAIM was based on Java data structure `Vector` that provides a very fast insertion with the complexity $O(1)$ when it is performed at the end of the vector and a slow lookup with the complexity $O(n)$. This performance is in contrast with the requirements of modern applications. As we mentioned before, we implemented a number of tuple spaces based on other data structures and techniques (hash tables, indexing, etc.) and provided the possibility to choose one of them. The following Java classes represent implemented tuple spaces:

- `TupleSpaceList` based on linked lists. We did not apply any techniques to boost the throughput of the tuple space since it would not improve substantially, given that linear search time will still apply.

- `TupleSpaceHashtable` based on hash tables. Since the reference to a specific tuple can be stored in several hash tables, the implementation of the withdrawing operation `in` needs to remove all its occurrences. To guarantee this, we add the attribute `isRemoved` to each tuple. This attribute is an *atomic variable* (all operations on it are atomic) of the Java class `AtomicBoolean`. When a process checks whether a tuple is marked as *removed* it uses the method `CompareAndSet` to check its value. If the value is `false`, it is set to `true` and `true` is returned. Otherwise, `false` is returned and its reference is removed from the searched hash table. When withdrawing a tuple, we remove the reference to it only from the part of the tuple space where we performed the search and mark this tuple `removed`. Other references will be removed when other querying operations will be performed: if a process finds a tuple that is marked `removed` it removes its reference and continues the search. For the tuple space based on hash tables, we implemented *separate chaining* with linked lists to store tuples that have the same hash value.

- `TupleSpaceTree` based on red-black trees. The implementation is similar to the one based on ordinary hash tables. For each type of used templates, a separate tree has to be added. For each tree, we use a global lock meaning that only one operation on the tuple space can be performed at the same time.

- `IndexedTupleSpace` based on indexing. The tuple space based on indexing uses hash tables to store tuples and their unique identifiers.

Tuples with the same structure are collected in a separate hash table and indexes are stored separately from tuples.

- `TupleSpaceConcurrentHashtable` based on a concurrent version of hash tables `ConcurrentHashMap` from Java JDK. This collection supports full concurrency of retrievals and adjustable expected concurrency for updates [86]. The actual concurrent implementation is similar to one based on ordinary hash tables but operations for querying and modifying can be performed in parallel and, thus, it is necessary to check also whether the tuple space has been previously modified. For this check, we use the techniques for handling subscriptions of processes for updates that were previously described (see Section 2.2).

In our experiments, we have used KLAIM instantiated with tuple spaces based on indexing since this option is the most versatile one and does not require extra tuning. Tuple spaces based on hash tables and trees could offer better performance but require a careful tuning that unequalizes the comparison and disallows the creation of automatic generic tests.

Additional operations. Our implementation improves the previous one by also adding *group operations* and *an atomic operation*. Group operations are presented in many works, see, e.g., *Grinda* [66], *EgoSpace* [87], *TuCSoN* [88], and are offered to write clearer code reducing the number of its lines; to reduce the time of data transmission and communication; to make operations on tuple spaces faster. We used the following group operations:

- `outMany(tuples)` to write a number o tuples;
- `rdMany(template, maxNumber)` to read several tuples at once;
- `inMany(template, maxNumber)` to retrieve several tuples at once.

For synchronization purpose, we implemented the atomic operation `outIfAbsent` that can be seen as a combination of the non-blocking `rd` operation followed by the `out` operation in case there is no tuple matching the template of `rd`. The operation does not return any value but can guarantee that a tuple will be not doubled. This operation is useful when it is required to prevent the presence of multiple copies of a certain tuple. For instance, when several identical processes have to set an initial data. An example of how this operation can be used is presented in Listing 3.16 where a tuple that initializes a counter $\langle \text{“counter”}, 0 \rangle$ will be inserted into a tuple space only if this counter does not exist before.

```
Tuple template = new Tuple(Object[]{"counter", Integer.class });
Tuple tuple = new Tuple(Object[]{"counter", 0});
node.outIfAbsent(template, tuple);
```

Listing 3.16: An example with the operation `outIfAbsent`

Chapter 4

Evaluating implementations

This chapter is devoted to the experimental evaluation of the implementations that we have selected in Section2.3: GIGASPACES, KLAIM, MOZARTSPACES, and TUPLEWARE. First, we present the testing methodology that we use to evaluate tuple spaces, i.e., we describe the considered case studies and how we conducted the experiments. Then, we consider two sets of experiments. The first one aims at comparing the implementation of the new version of KLAIM described in Chapter 3 with the old implementation. The second one compares the new KLAIM with the existing implementations of tuple spaces that we introduced and selected in Section 2.3.

4.1 Methodology

In this section, we present the case studies that we will use to evaluate the performance of different implementations of tuple spaces and describe how we perform the experiments.

4.1.1 Case study

We consider four case studies: *Password search*, *Sorting*, *Ocean model* and *Matrix multiplication*. We describe them below.

The first case study is of interest since it deals with a large number of tuples and requires performing a huge number of write and read operations. This helps us understand how efficiently an implementation performs operations on local tuple spaces with a large number of tuples.

The second case study is computation intensive since each node spends more time for sorting elements than on communicating with the others. This case study has been considered because it needs structured tuples that contain both basic values (with primitive type) and complex data structures, and these impact on the speed of the inter-process communication.

The third case has been chosen because it introduces particular dependencies among nodes, which if exploited can improve the application performances. Our aim was to check whether adapting a tuple space system to the specific inter-process interaction pattern of a specific class of applications could lead to significant performance improvements.

The last case study is a communication-intensive task and it requires much reading on local and remote tuple spaces. It was chosen to assess how the communication part of each implementation affects performances of operation on remote tuple spaces.

All case studies are implemented using the master-worker paradigm [89] because among other design patterns (e.g., Pipeline, SPMD, Fork-join) [90] it fits well with all our case studies and allows us to implement them in a uniform way. In the rest of this subsection, we briefly describe all the case studies.

We would like to stress that, with our case studies, we aim at evaluating advantages and disadvantages of the chosen tuple space implementations. We do not aim at providing efficient solutions for them. Actually, some of our solutions have intentional redundancy that slows down performance but allows us to highlight the difference between the different implementations. For instance, the code for the *Ocean model* and the one for the *Matrix multiplication problem* can be easily improved by assigning data or jobs to specific workers thus making data search unnecessary.

To highlight the key steps of the proposed algorithms, we present the basic Linda code of the master and of the workers for each case study. The full Java code used in the actual experiments has many additional lines that render it less intuitive but allow us to run it on the different tuple space implementations. It can be accessed at Github: `www.github.com/IMTAltiStudiLucca/Klava2`.

Password search. The main aim of this application is to find a password using its hash value in a predefined "database" distributed among processes. Such a database is a set of files containing pairs (password, hash value). The application creates a master process and several worker processes (Figure 7): the master keeps asking the workers for passwords corresponding to specific hash values, by issuing tuples of the form:

$$\langle \text{``search\_task"}, dd157c03313e452ae4a7a5b72407b3a9, \text{``not\_processed"} \rangle$$

Each worker first loads its portion of the distributed database and then obtains from the master a task to look for the password corresponding to a hash value. Once it has found the password, it sends the result back to the master, with a tuple of the form:

$$\langle \text{``found\_password"}, dd157c03313e452ae4a7a5b72407b3a9, 7723567 \rangle$$

For multiple tuple space implementations, it is necessary to start searching in one local tuple space and then to check the tuple spaces of other workers. The application terminates its execution when all the tasks have been processed and the master has received all required results.

Listing 4.1 presents the code for *Password Search*. The master writes, in its tuple space, n tasks with known hash values of passwords to be found (lines 2-3), waits for the found passwords (lines 5-6) and informs workers to terminate their work because all tasks have been accomplished (lines 9-10). A worker loads predefined data (lines 15-16), takes a task to perform (lines 18, 26), looks for a password locally, if needed looks for

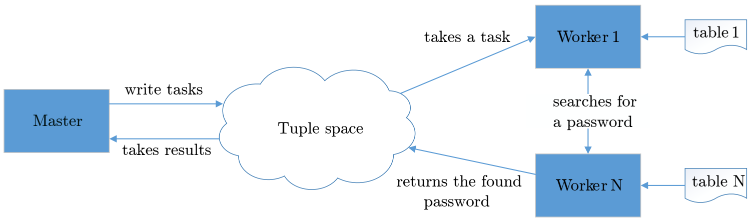

Figure 7: Schema of the case study *Password search*

a password in tuple spaces of other workers (line 22-24) and returns the found password to the master (line 25).

```
1  Master():
2    for i in range(n):
3      masterTS.out("search_task", hash[i], "not_processed")
4
5    for i in range(n):
6      masterTS.in("found_value", ?hashValue, ?password)
7
8    // poison tuples for all workers
9    for i in range(workerNumber):
10     masterTS.out("search_task", null, "finished")
11
12
13 Worker():
14   // loads passwords
15   for i in range(m):
16     localTS.out("hash_set", db[i].hashValue, db[i].password)
17
18   masterTS.in("search_task", ?hashValue, ?status)
19   while status != "finished":
20     if localTS.rdp("hash_set", hashValue, ?password) == false:
21       // search in other tuple spaces
22       for i in range(workerNumber):
23         if workerTS[i].rdp("hash_set", hashValue, ?password) == true:
24           break
25     masterTS.out("found_value", hashValue, password)
26     masterTS.in("search_task", ?hashValue, ?status)
```

Listing 4.1: Password search. Listing of master and worker processes

Sorting. This program sorts an array of integers. The master performs the initial data loading and collects the sorted data; workers perform the actual sorting. The master, after loading the data, splits them into many parts, stores them in its own tuple space and then waits for the sorted arrays from the workers. Once the master has collected all sorted sub-arrays, it builds the whole sorted sequence. A worker looks for unsorted data in the tuple space of the master and when it finds a tuple with unsorted data, it sorts it, sends the result to the master and continues looking for data to sort. An example of sorting is shown in Figure 8.

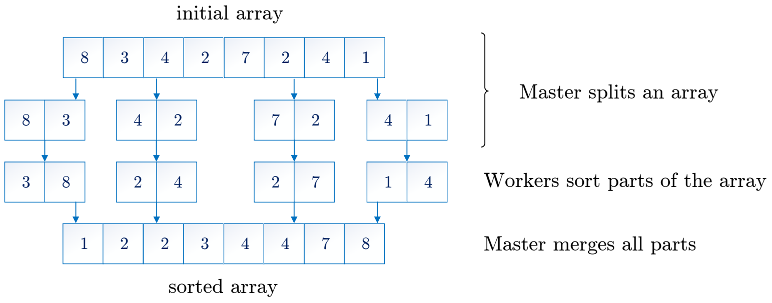

Figure 8: Schema of the case study *Sorting*

Listing 4.2 presents the code for *Sorting*. The master splits an initial array into *nParts* parts (in our tests *nParts* = 200), adds them to its tuple space (lines 3-4) and begins to collect sorted parts (lines 6-9). When all parts are collected (line 6), the master reconstructs the full sorted array (line 11) and notifies all workers to terminate their work (lines 14-15). A worker takes an unsorted array from the master (line 19), sorts it and sends the sorted array to the master (lines 21-22).

```
1  Master():
2    unsortedParts = splitInitialArray(initialArray, nParts)
3    for i in range(unsortedParts.size):
4      masterTS.out("sort_array", unsortedParts[i], "unsorted")
5
6    while n != initialArray.size:
7      masterTS.in("sorted_part", sortedPart)
8      sortedParts.add(sortedPart)
```

```
    n += sortedPart.size
  // reconstruct a sorted array
  sortedArray = reconstructArray(sortedParts)

  // poison tuples for all workers
  for i in range(workerNumber):
    masterTS.out("sort_array", null, "finished")


Worker():
  masterTS.in("sort_array", ?arrayPart, ?status)
  while status != "finished":
    sortedArrayPart = sort(arrayPart)
    masterTS.out("sorted_parts", sortedArrayPart)
    masterTS.in("sort_array", ?arrayPart, ?status)
```

Listing 4.2: Sorting. Listing of master and worker processes

Ocean model. The ocean model is a simulation of the enclosed body of water that was considered in [13]. The two-dimensional (2-D) surface of the water in the model is represented as a 2-D grid and each cell of the grid represents one point of the water. The parameters of the model are current velocity and surface elevation which are based on a given wind velocity and bathymetry. In order to parallelize the computation, the whole grid is divided into vertical panels (Figure 9), and each worker owns one panel and computes its parameters. The parts of the panels, which are located on the border between them are colored. Since the surface of the water is continuous, the state of each point depends on the states of the points close to it. Thus, the information about bordering parts of panels should be taken into account. The aim of the case study is to simulate the body of water during several time-steps. At each time-step, a worker recomputes the state (parameters) of its panel by exploiting parameters of the adjacent panels.

The tasks of the master and workers are similar to the previous case studies. In the application the master instantiates the whole grid, divides it into parts and sends them to the workers. When all the iterations are completed, it collects all parts of the grid. Each worker receives its share

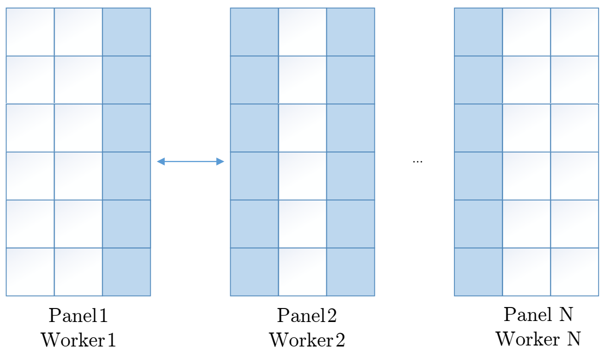

Figure 9: Schema of the case study *Ocean model*

of the grid and at each iteration it communicates with workers which have adjacent grid parts in order to update and recompute the parameters of its model. When all the iterations are completed, each worker sends its data to the master.

Listing 4.3 presents the code for *Ocean model*. The master generates a model, splits it into a number of panels and distributes them among workers (lines 2-4). Then it collects the processed panels (lines 6-8) and reconstructs the model (line 9). A worker gets a panel from the master (line 13) and passes it through a number of iterations (lines 16-26); at each iteration, it shares a part of its data with other processes (line 18-19) and gets data of adjacent panels (lines 21-24). In the end, the worker returns a panel to the master (line 28).

```
1   Master():
2     panels = splitModel(oceanModel)
3     for  i  in  range(workerNumber):
4       masterTS.out("oceanModel", "panel", panels[i])
5     // take back
6     for  i  in  range(workerNumber):
7       masterTS.in("oceanModel", "ready_panel", ?panel)
8       processedPanels.add(panel)
9     oceanModel = reconstructModel(processedPanels)
10
11
12  Worker():
```

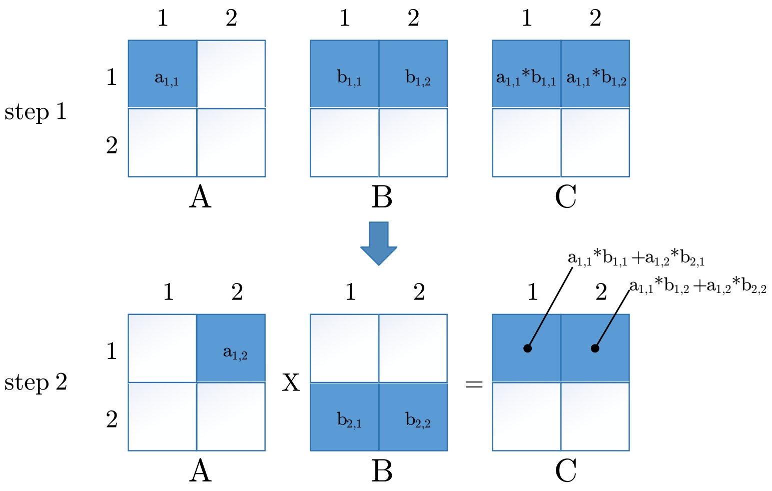

Figure 10: Schema of the case study *Matrix multiplication*

```
masterTS.in("oceanModel", "panel", ?panel)

// do several iterations
while panel.iterationNumber < iterationMax:
  // for each adjacent panel write required boundary data
  for borderInfo in panel.borders:
    localTS.out("oceanModel", "border", borderInfo.name,
         borderInfo.data)

  for borderInfo in panel.borders:
    for i in range(workerNumber):
      workerTS[i].inp("oceanModel", "border", borderInfo.name,
           ?borderData)
      panel.borderData.add(borderData)
  // update its state
  panel.process()

masterTS.out("oceanModel", "ready_panel", panel)
```

Listing 4.3: Ocean model. Listing of master and worker processes

Matrix multiplication. The case study is designed to multiply two square matrices of the same order. The algorithm of multiplication [91] operates with rows of two matrices A and B and puts the result in matrix C. The latter is obtained via subtasks where each row is computed in parallel. At the j-th step of a task the i-th task, the element, a_{ij}, of A is multiplied by all the elements of the j-th row of B; the obtained vector is added to the current i-th row of C. The computation stops when all subtasks terminate. Figure 10 shows how the first row of C is computed if A and B are 2×2 matrices. In the first step, the element $a_{1,1}$ is multiplied first by $b_{1,1}$ then by $b_{1,2}$, to obtain the first partial value of the first row. In the second step, the same operation is performed with $a_{1,2}$, $b_{2,1}$ and $b_{2,2}$ and the obtained vector is added to the first row of C thus obtaining its final value.

Initially, the master distributes the matrices A and B among the workers. In our case study, we have considered two alternatives: (i) the rows of both A and B are spread uniformly, (ii) the rows of A are spread uniformly while B is entirely assigned to a single worker. This helped us in understanding how the behavior of the tuple space and its performances change when only the location of some tuples changes.

Listing 4.4 presents the code for *Matrix multiplication*. The master, first, makes the matrices (A and B) to be multiplied available to workers, by putting them into its tuple space (lines 2-4), then it collects the results sent back by the workers (lines 6-8). Each worker, first, loads parts of A and B (lines 13-23), then it computes rows of the matrix products that correspond to the rows of A (lines 26-48). Using the received rows of A, a worker looks for the necessary rows of B (lines 33-35) and stores intermediate results locally (lines 39-48), finally it sends to the master the computed rows of matrix product (line 46).

```
1  Master():
2    for i in range(matrixSize):
3      masterTS.out("matrixA", i, matrixA[i], matrixSize)
4      masterTS.out("matrixB", i, matrixB[i], matrixSize)
5
6    for i in range(matrixSize):
7      masterTS.take("matrixC", i, ?matrixCRow, matrixSize)
```

```
    matrixC.add(i, matrixCRow)


Worker():
  // rows per each worker
  rowsPerWorker = matrixSize/workerNumber
  if workerID < matrixSize % workerNumber:
    rowsPerWorker++;

  for i in range(rowsPerWorker):
    masterTS.in("matrixB", ?rowID, ?rowData, ?counter)
    localTS.out("matrixB", rowID, rowData, counter)

  for i in range(rowsPerWorker):
    masterTS.in("matrixA", ?rowID, ?rowData, ?counter)
    tasks.add(rowID, rowData)

  // look at each row lk tasks
  for i in range(tasks.size()):
    rowID = tasks[i].id
    rowFirstOperand = tasks[i].rowData

    // check all rows of matrix B
    for k in range(matrixSize):
      // take k-row from matrix B
      for w in range(workerNumber):
        if workers[w].rdp("matrixB", i, ?rowB, ?counterB) == true:
          break
      rowMatrixC = rowB * rowFirstOperand[k]

      // check matrix C (existence of partial result)
      if localTS.inp("matrixC", rowID, ?valC, ?counter) == false:
        localTS.out("matrixC", rowID, rowMatrixC, 1)
      else
        rowMatrixC = rowMatrixC + valC
        counter++

        if counter == matrixSize:
          masterTS.out("matrixC", rowID, rowMatrixC, counter)
        else
          localTS.out("matrixC", rowID, rowMatrixC, counter)
```

Listing 4.4: Matrix multiplication. Listing of master and worker processes

4.1.2 Experiment setup

Parameters of case studies. All the conducted experiments are parametric with respect to two values. The first one is the number of workers $w \in \{1, 5, 10, 15\}$. This parameter is used to test the scalability of the different implementations. The second parameter is application specific, but it aims at testing the implementations when the workload increases.

- *Password search*: We vary the number of the entries in the database (1×10^4, 1×10^5, 1×10^6 passwords) where it is necessary to search a password. This parameter directly affects the number of local entries each worker has. Moreover, for this case study, the number of passwords to search was fixed to 100.
- *Sorting*: We vary the size of the array to be sorted (1×10^5, 1×10^6, 1×10^7 elements). In this case, the number of elements does not correspond to the number of tuples because parts of the array are transferred also as arrays of smaller size.
- *Ocean model*: We vary the grid size (300, 600 and 1200) which is related to the computational size of the initial task.
- *Matrix multiplication*: We vary the order of a square matrix (50, 100).

Remark 1 (*Execution environment).* Our tests were conducted on an Ubuntu server (version 14.04.5 LTS) with 4 processors Intel Xeon E5-4607, each with 6 cores, 12 M Cache, 2.20 GHz, and with hyper-threading, offering in total 48 threads and 256 GB RAM. All case studies are implemented in Java 8.

Measured metrics. For the measurement of metrics, we have created a profiler which is similar to Clarkware Profiler[1]. However, Clarkware Profiler calculates just the average time for the time series, while ours also calculates other statistics (e.g., standard deviation). Moreover, our profiler was designed also for analyzing tests carried out on more than

[1]The profiler was written by Mike Clark; the source code is available on GitHub: `https://github.com/akatkinson/Tupleware/tree/master/src/com/clarkware/profiler`.

one machine. For that reason, each process writes raw profiling data on a specific file; all files are then collected and used by specific software to calculate required metrics.

We use the manual method of profiling and insert methods `begin(label)` and `end(label)` into the program code around the part of the code we are interested in to start and stop counting time respectively. For each metrics, a different label is used. An example of profiling is shown in Listing 4.5 where the writing time is measured: the operation of the remote writing is surrounded with profiling methods that are called with the label "`writing_time`". At the end of the execution of each application, all the data are stored on disk for subsequent analysis.

```
TupleLogger.begin("writing_time");
node.out(tuple, remoteLocality);
TupleLogger.end("writing_time");
```

Listing 4.5: Manual profiling

Each set of experiments has been conducted 10 times with a randomly generated input and we have computed an average value and a standard deviation for each metrics. To extensively compare the different implementations, we have collected the following measures:

- *Local writing time:* time required to write one tuple into a local tuple space.
- *Local reading time:* time required to read one tuple from a local tuple space using a template. This metrics checks also how fast pattern matching works.
- *Remote writing time:* time required to communicate with a remote tuple space and to perform one write operation on it.
- *Remote reading time:* time required to communicate with a remote tuple space and to perform one read operation on it.
- *Search time:* time required to search a tuple in a set of remote tuple spaces.

- *Total time:* total execution time. This time does not include an initialization of tuple spaces.
- *Number of visited nodes:* number of visited tuple spaces before a searched tuple was found.

Notice that all plots used in the paper report results of our experiments on a logarithmic scale. When describing the outcome, we have only used the plots which are more relevant to highlight the difference between the performances of the different tuple space systems.

4.2 Old KLAIM vs new KLAIM

In this section, we compare performances of the improved version of KLAIM with its older version. To do it, we use the *Password search*, *Sorting* and *Matrix multiplication* case studies. The outcome of the *Ocean model* case study is similar to the of *Matrix multiplication* and, thus, we do not report them. In the rest of the section, by writing KLAIM we will indicate the new version of the framework and will refer to its previous version by calling it *old* KLAIM.

Password search. Adding an element at the end of an array takes less time than adding an element in other data structures. This is the reason the writing time of KLAIM based on indexing is lower than that of old KLAIM (the tuple space of old KLAIM is vector-based). This fact is evidenced in Figure 11.

Differently from the outcomes of the measures of writing time, the tuple space based on indexing outperforms old KLAIM. As shown in Figure 12, the local reading and withdrawing times are much lower for KLAIM. In Figure 12, we also consider the execution times for the implementation based on hash tables (KLAIM(H)), it shows that the latter provides better results than the others.

Sorting. While improving the implementation, we noticed a problem with inefficient data transmission in old KLAIM that slows down writing

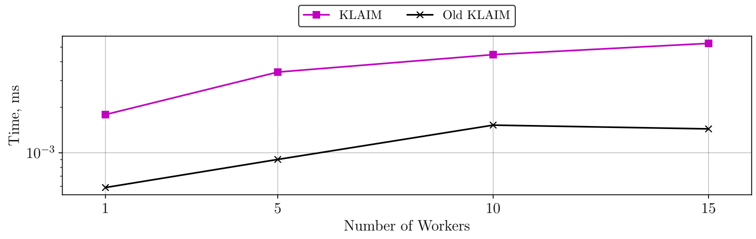

Figure 11: Password search. Local writing time (1×10^6 passwords)

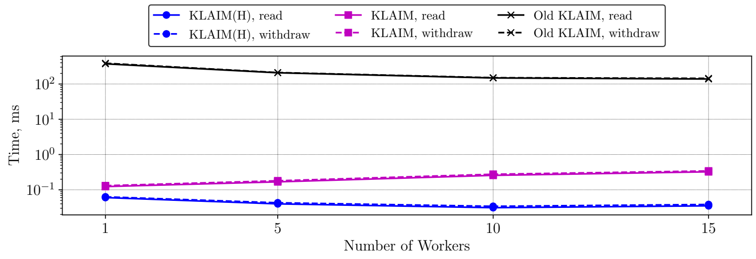

Figure 12: Password search. Local reading and withdrawing times (1×10^6 passwords)

and reading times especially when bigger portions of data are transmitted. As shown in Figure 13, the remote reading time is much smaller for KLAIM than for old KLAIM.

Matrix multiplication. Performance in the *Matrix multiplication* case study depends on remote operations more than for other case studies. Here, we report only the results for the case in which matrix B is uniformly distributed among the workers; the other case leads to similar results. As shown in Figure 14-15, the remote writing and reading times decrease significantly when we compare KLAIM with old KLAIM. In Figure 15, the remote reading time for the runs with one worker is not shown since, in this case, only the local tuple space of the worker is used.

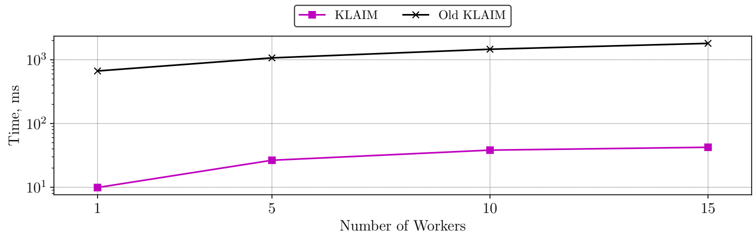

Figure 13: Sorting. Remote reading time (10×10^6 elements)

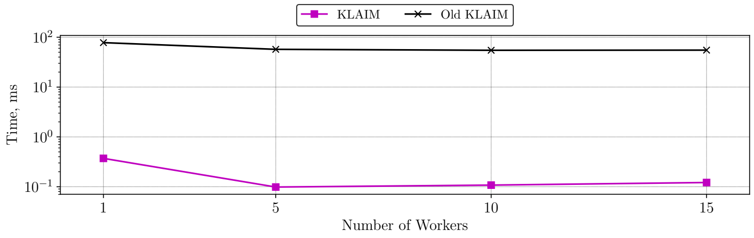

Figure 14: Comparisons of two KLAIMs. Matrix multiplication. Remote writing time (the matrix order is 100)

4.3 Assessing different implementations

In this section, we evaluate four implementations of the tuple space that have been selected in Section2.3 (GIGASPACES, KLAIM, MOZARTSPACES, and TUPLEWARE) using four case studies (Password search, Sorting, Ocean model, and Matrix multiplication). In our experiments, we use only the new version of KLAIM (that we call KLAIM). In addition to the results of experiments carried out on a single host machine, we also present results of experiments that were conducted on several host machines.

Password search. In Figures 16-17, the trend of the total execution time is reported as the number of workers and size of considered database increase. In Figure 16 the size of the database is 1×10^5 entries, while Figure 17 reports the case in which the database contains 1×10^6 elements. From the plot, it is evident that GIGASPACES and KLAIM exhibit better

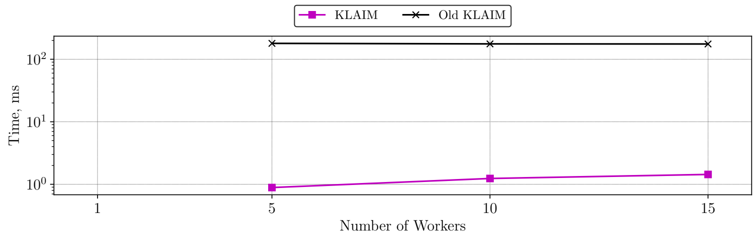

Figure 15: Comparisons of two KLAIMs. Matrix multiplication. Remote reading time (the matrix order is 100)

performances than the other systems. In the diagrams below, in addition to the results for four implementations, we will provide measures for a different implementation of the case study with TUPLEWARE. We will refer to the additional experiment by TUPLEWARE(H) and will describe it in some details at the end of this paragraph.

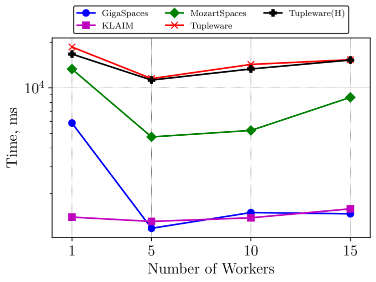

Figure 16: Password search. Total time (1×10^5 passwords)

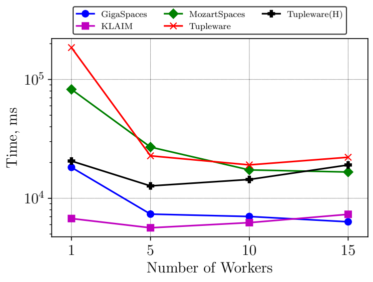

Figure 17: Password search. Total time (1×10^6 passwords)

Figure 18 depicts the local writing time for each implementation with different numbers of workers. As we can see, by increasing the number of workers (that implies reducing the amount of local data to consider), the local writing time decreases. This is more evident for TUPLEWARE, that really suffers when a big number of tuples (e.g. 1×10^6) is stored in a single local tuple space. The writing times of KLAIM and GIGASPACES are the

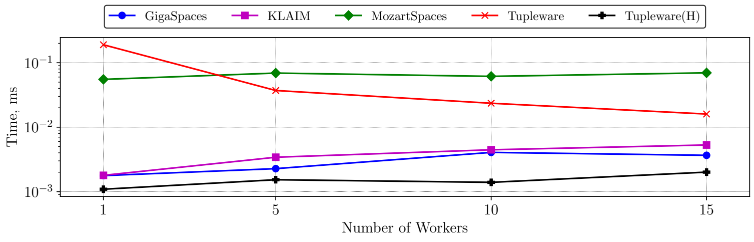

Figure 18: Password search. Local writing time (1×10^6 passwords)

lowest among other systems and do not change significantly during any variation in the experiments. The local writing time of MozartSpaces remains almost the same when the number of workers increases. Nonetheless, its local time is bigger with respect to the other systems, especially when the number of workers is equal or greater than 10.

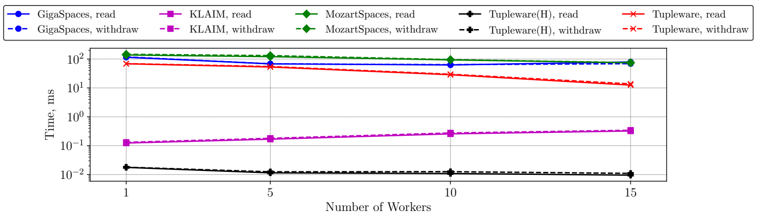

Figure 19: Password search. Local reading and withdrawing times (1×10^6 passwords)

In our experiments, we did measure only the writing and reading time which are the most frequently used in the considered case studies. We ignored the withdraw operation; that can be seen as an atomic step consisting of a read followed by a removal action. The latter requires synchronization on (part of) the tuple space, and the way synchronization is guaranteed highly impacts on performances. Thus, we only evaluated the cost of withdrawing for the Passwords search case study by considering local searches and allowing withdrawing, and not just reading, passwords. The local reading and withdrawing times for the new program are shown in Figure 19. As expected, withdrawing time is slightly greater than reading time.

The reading times of GIGASPACES and MOZARTSPACES are always similar, whereas the reading time of TUPLEWARE decreases when the number of workers grows. KLAIM performs much better than others. Since this case study requires little synchronization among workers, generally, performance improves when the level of parallelism (the number of workers) increases.

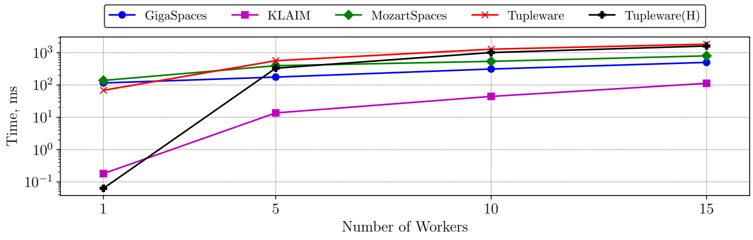

Figure 20: Password search. Search time (1×10^6 passwords)

We would like now to comment on the behavior of TUPLEWARE whose source code is publicly available[2]. TUPLEWARE exploits `Hashtables` as a container for tuples but their efficient use relies on using specific kinds of templates (the first fields should contain values, not variables). This is not the case for our skeleton implementation and thus TUPLEWARE's potentials are not fully exploited. Thus, we considered an alternative skeleton with tuples that meet TUPLEWARE requirements and noticed that its performances greatly improved while those of the others remained unchanged. In Figures 16 - 20 TUPLEWARE(H) refers to tests with this adapted code. TUPLEWARE(H) exhibits a writing time close to the best one of KLAIM (Figure 18) and the best reading time (Figure 19). TUPLEWARE(H) is particularly efficient when databases of bigger size are used.

In general, searching time is similar to local reading time but it needs

[2]Source code of TUPLEWARE is available on GitHub`https://github.com/akatkinson/Tupleware`.

to take into account searching in remote tuple spaces. When considering just one worker, the searching time is the same as the reading time in a local tuple space, however, when the number of workers increases the searching time of TUPLEWARE and KLAIM grows faster than the time of GIGASPACES. Figure 20 shows that GIGASPACES and MOZARTSPACES are more sensitive to the number of tuples than to the number of accesses to the tuple space.

Summing up, we can remark that the local tuple spaces of the four systems exhibit different performances depending on the operation on them: KLAIM and GIGASPACES exhibit best writing time, while KLAIM demonstrates also fast querying operations. However, some of the implementations do not exhibit their best performance without specific tuning. For instance, without any change in the code of case study's skeleton, TUPLEWARE demonstrates mean performance, whereas, with the adapted code that satisfies some requirements of TUPLEWARE, its performance increases significantly.

Sorting. Figure 21 shows that GIGASPACES exhibits significantly better execution time when the number of elements to sort is 1 million. As shown in Figure 22 when 10 million elements are considered and several workers are involved, TUPLEWARE, KLAIM and MOZARTSPACES exhibit a more efficient parallelization and, thus, require less time.

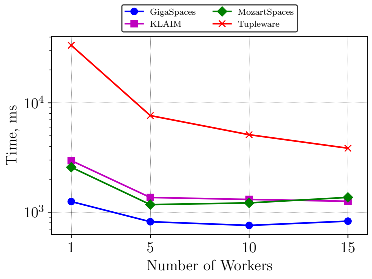

Figure 21: Sorting. Total time (1×10^6 elements)

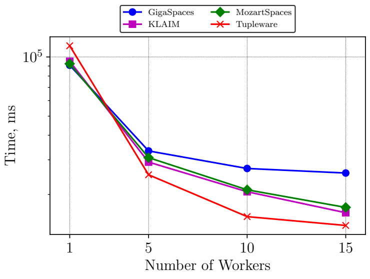

Figure 22: Sorting. Total time (10×10^6 elements)

This case study is a computation intensive one but requires also an exchange of structured data. The benefits of the parallelization are more evident for the tests with an array of 10 million elements where the percentage of the time spent for data communication and transmission is less than the time spent for sorting.

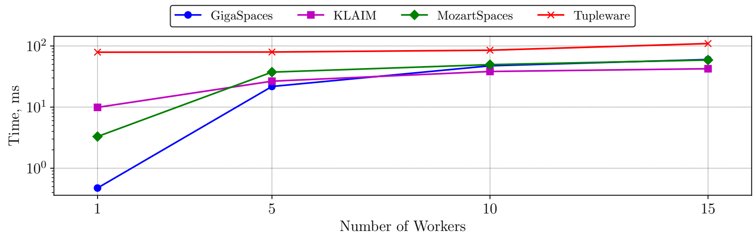

Figure 23: Sorting. Remote reading time (10×10^6 elements)

As shown in Figure 23, the remote reading time of TUPLEWARE is always higher than that of other tuple spaces, and all those demonstrate similar behavior. Figure 24 shows that, in all cases, increasing the number of workers does not affect the local writing time. At the same time, by comparing Figures 18 and 24, we see that all implementations of tuple spaces except TUPLEWARE and KLAIM spend more time for writing. This is especially evident for GIGASPACES.

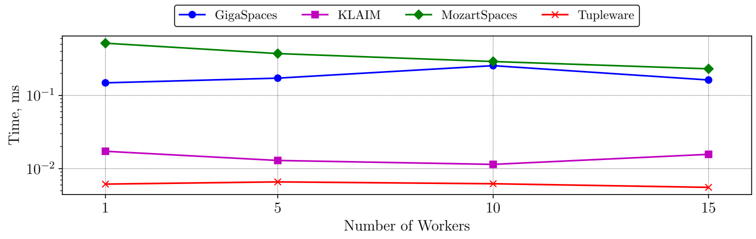

Figure 24: Sorting. Local writing time (10×10^6 elements)

It is worth noting that, during our tests, we experienced some problems with MOZARTSPACES (the version of the library is *mozartspaces-dist-*

2.3-SNAPSHOT-r15239). The problems, related to data loss, occurred when more than 10 workers were present. After analyzing our code, the Vienna group pinpointed the bug and fixed it (the version of the library is *mozartspaces-dist-2.3-SNAPSHOT-r21c82ce88e5456*). The experiments presented here are based on the revised implementation.

Ocean model. This case study was chosen to examine the behavior of tuple space systems when specific patterns of interactions come into play. Out of the four considered systems, only TUPLEWARE has a method for reducing the number of visited nodes during search operation which helps in lowering search time. Figure 25 depicts the number of visited nodes for different grid sizes and a different number of workers (for this case study in all figures we consider only 5, 10, 15 workers because for one worker generally tuple space is not used). The curve depends weakly on the size of the grid for all systems and much more on the number of workers. Indeed, from Figure 25 we can appreciate that TUPLEWARE performs a smaller number of nodes visits and that when the number of workers increases the difference is even more evident[3].

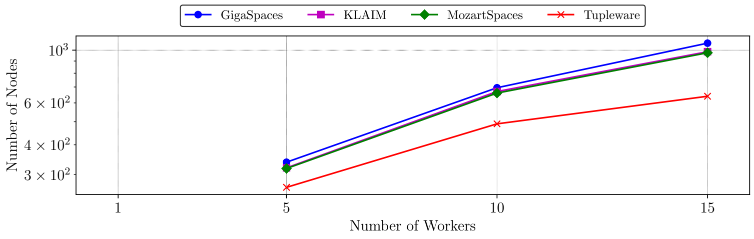

Figure 25: Ocean model. Number of visited nodes (the grid size is 1200)

The difference in the number of visited nodes does not affect significantly the total time of execution for different values of the grid size (Figure 27-28) mostly because the case study requires many read operations from remote tuple spaces (Figure 26).

[3]In Figures 25, the curves for KLAIM and MOZARTSPACES are overlapping and green wins over purple.

As shown in Figure 26 the time of remote operation varies for different tuple space systems. For this case study, we can neglect the time of the pattern matching and consider that this time is equal to the time of communication. For TUPLEWARE, this time is significantly greater than that of the others. GIGASPACES, that has a centralized implementation, most likely does not use TCP for data exchange but relies on a more efficient memory-based approach. The communication time of KLAIM and MOZARTSPACES is in the middle (in the plot with logarithmic scale) but close to GIGASPACES by its value: for GIGASPACES this time varies in the range of 0.0188 to 0.0597 ms, for KLAIM and MOZARTSPACES in the range of 2.0341 to 3.0108 ms, and for TUPLEWARE it exceeds 190 ms[4].

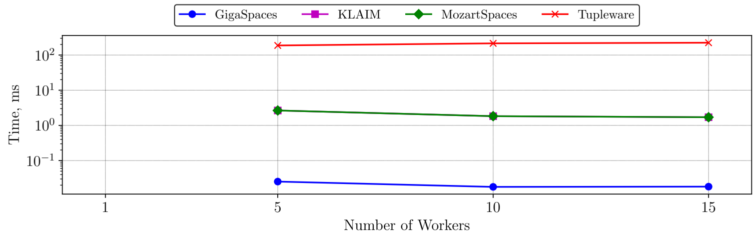

Figure 26: Ocean model. Remote reading time (the grid size is 1200)

The total execution time generally follows the remote reading time as shown in Figures 27 and 28. Even with the reduced number of visited nodes, because of the high remote reading time, TUPLEWARE falls behind other tuple spaces.

Matrix multiplication. This case study consists mostly of searching tuples in remote tuple spaces, and this implies that the number of remote read operations is by far bigger than the other operations. Therefore, GIGASPACES outperforms other tuple space systems total execution time (Figure 29).

As discussed above, we consider two variants of this case study: one in which matrix B is uniformly distributed among the workers (as the

[4]In Figures 26, the curves for KLAIM and MOZARTSPACES are overlapping.

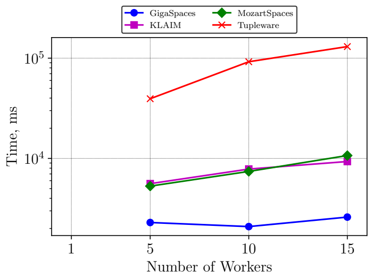

Figure 27: Ocean model. Total time (the grid size is 600)

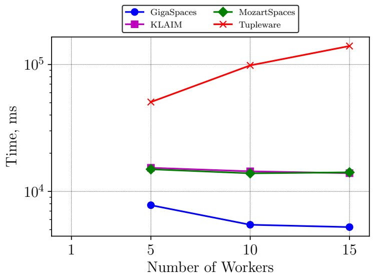

Figure 28: Ocean model. Total time (the grid size is 1200)

matrix A), and one in which the whole matrix is assigned to one worker. In the following plots, solid lines correspond to the experiments with uniform distribution and dashed lines correspond to ones with the second type of distribution (names ending with B-1 are used to refer this kind of distribution).

Figure 30 depicts the average number of the nodes that it is necessary to visit in order to find a tuple for each worker. When considering experiments with more than one worker all tuple space systems except TUPLEWARE demonstrate similar behavior: the total time almost coincides for both types of the distribution. However, for the uniform distribution TUPLEWARE exhibits always greater values and for the second type of distribution, the values are significantly lower. The second case reaffirms the results of the previous case study because in this case all workers know where to search the rows of the matrix B almost from the very beginning that leads to the reduction of the amount of communication, affects directly the search time (Figure 31) and, in addition, implicitly leads to the lower remote reading time (Figure 32, the remote reading time is not displayed for one worker because only the local tuple space of the worker is used). In contrast, for the uniform distribution TUPLEWARE performs worse because of the same mechanism which helps it in the previous case: when it needs to iterate over all the rows one by one it always starts the checking from the tuple spaces which were already checked at the previous time and which do not store required rows. Therefore, every time it checks roughly all tuple spaces.

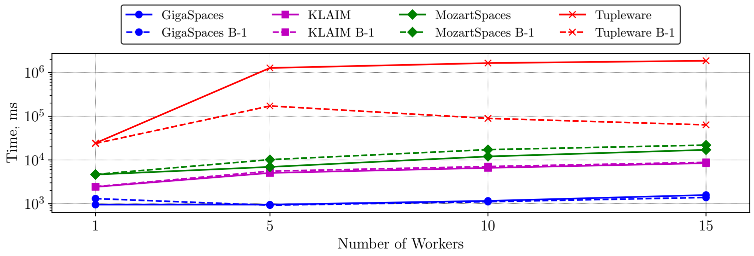

Figure 29: Matrix multiplication. Total time (the matrix order is 100)

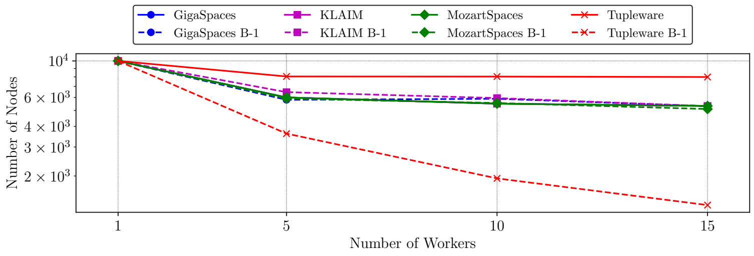

Figure 30: Matrix multiplication. Number of visited nodes (the matrix order is 100)

The results of this case study are generally consistent with the previous ones: remote operations of GIGASPACES, KLAIM and MOZARTSPACES are much faster and better fits to the application with frequent inter-process communication; TUPLEWARE continues to have an advantage in the application with a specific pattern of communication. At the same time, we noticed that in some cases this feature of TUPLEWARE had some side-effects that negatively impacted on its performance.

Block-wise matrix multiplication. We now introduce a variant of the matrix multiplication algorithm supporting a higher degree of parallelization and relying on a different communication strategy. In the "row-wise"

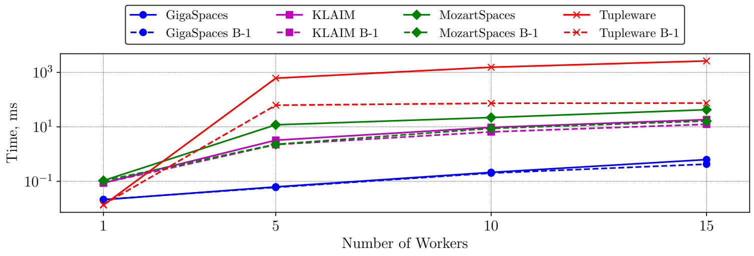

Figure 31: Matrix multiplication. Search time (the matrix order is 100)

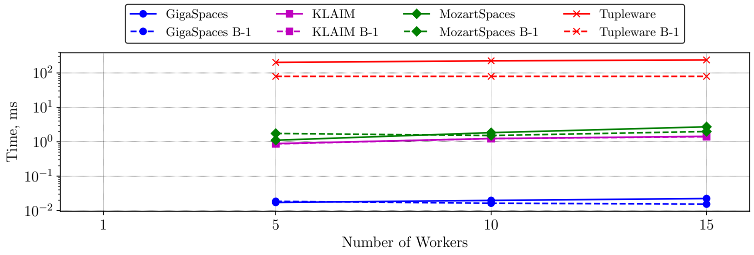

Figure 32: Matrix multiplication. Remote reading time (the matrix order is 100)

multiplication, each worker evenly gets parts of the initial matrices and searches for required rows in tuple spaces of the other workers. In the block-wise approach, instead, the master keeps all data and workers access its space to get them. The block-wise algorithm of multiplication operates with matrices of smaller size (called blocks) obtained by evenly splitting the initial matrices.

Suppose that matrix A is split into blocks of q rows and n columns and matrix B is split into blocks of n rows and r columns. Then each cell $c_{i,j}$ of the resulting matrix C can be iteratively obtained using formula $c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j}$ where $i \in \{1, .., q\}$, $j \in \{1, .., r\}$ and $k \in \{1, .., n\}$.

In Listing 4.6 we report the actual code for *Block-wise matrix multiplication*. The master first initializes matrix C (line 2) and adds to its local tuple space the blocks for matrices A, B, C (lines 3-7), then it assigns tasks (lines 9-10). Finally, it collects the results sent by the workers (lines

12-14) and notifies termination to them (lines 17-18). Each worker performs the assigned tasks (lines 26-30) and terminates its work (line 24-25). After getting a task (line 23) a worker gets the two assigned matrix blocks (lines 26-27) and adds their product to the intermediate result (lines 28-30). Finally, it puts the result into the space of the master (line 31) and continues its work.

```
1  Master():
2    initMatrix(matrixC)
3    for i in range(blockNumber):
4      for j in range(blockNumber):
5        masterTS.out("matrixA", i, j, matrixA.block(i, j))
6        masterTS.out("matrixB", i, j, matrixB.block(i, j))
7        masterTS.out("matrixC", i, j, 0, matrixC.block(i, j))
8
9        for k in range(blockNumber):
10         masterTS.out("task", i, j, k)
11
12   for k in range(blockNumber∗blockNumber):
13     masterTS.in("matrixC", ?i, ?j, blockNumber, ?blockC)
14     C.block(i, j) = blockC
15
16   // poison tuples for all workers
17   for i in range(workerNumber):
18     masterTS.out("task", −1, −1, −1)
19
20
21 Worker():
22   while true:
23     masterTS.in("task", ?i, ?j, ?k)
24     if i == −1:
25       break
26     masterTS.rd("matrixA", i, k, ?blockA)
27     masterTS.rd("matrixB", k, j, ?blockB)
28     tempBlock = blockA∗blockB
29     masterTS.in("matrixC", i, j, ?count, ?blockC)
30     blockC = blockC + tempBlock
31     masterTS.out("matrixC", i, j, count+1, blockC)
```

Listing 4.6: Block-wise matrix multiplication. Listing of master and worker processes

We compared the performances of the two matrix multiplication algorithms using GIGASPACES since this implementation exhibited the best performances in the tests for the row-wise version. In our experiments, we divided the matrices into square blocks (e.g., $q = r = n$) to consider just one parameter that we call *block size* (we used *block size* $\in \{5, 10\}$). As shown in Figure 33, depending on the size of blocks and the number of workers the block-wise multiplication exhibits different times. When blocks of bigger size are used, we have that row-wise multiplication is constantly outperformed.

As shown in Figure 34, the remote reading time is much higher for the block-wise implementation; in this case reading requests are directed to master, whereas in the row-wise case they are directed to workers.

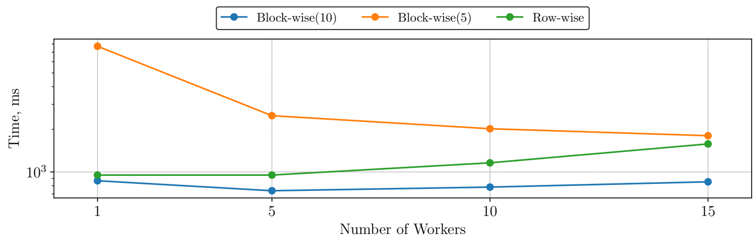

Figure 33: Block-wise and row-wise matrix multiplications. Total time (the matrix order is 100)

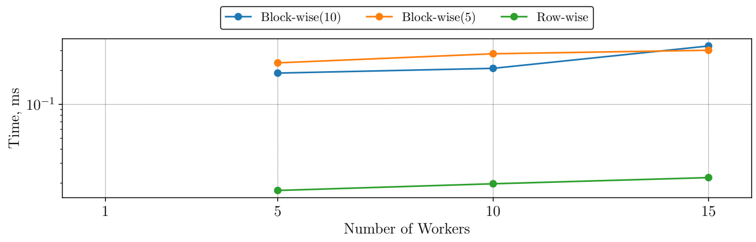

Figure 34: Block-wise and row-wise multiplications. Remote reading time (the matrix order is 100)

Experiments with several host machines. The results of the previous experiments which were conducted using only one host machine provide us evidence that GIGASPACES has a more efficient implementation of communication and that is very beneficial when many operations on remote tuple spaces are used. Since we do not have access to GIGASPACES's source code we conjecture that it uses an efficient inter-process communicating mechanism and do not resort to socket communications as done by the other implementations.

To check whether GIGASPACES remains efficient when running over a network, we have used Docker[5]. This technology allowed us to create a number of lightweight containers, similar to virtual machines, that are isolated environments equipped with an operating system where a minimal set of software components is installed that allows using fewer resources. Each container has a network interface to communicate with others. Masters and workers were launched in their own containers and each set of experiments was conducted 10 times. We conducted experiments over two case studies: *Sorting* and *Matrix multiplication* with uniform distribution and we focused on remote reading time, the most frequently used operation in all case studies.

To further explore the impact of GIGASPACES's inter-machine communication, we have also conducted experiments on a real network: 16 identical virtual machines (Oracle VM VirtualBox[6]) evenly placed in 3 real servers (see in Remark 2). Each virtual machine is equipped with 3 GB RAM and Linux Lubuntu 16.04.

When evaluating performances of *Sorting* we have that the remote reading time of the networked version is smaller than the one of the single host version (Table 5). This might be related to the amount of processed data (much larger than those of other case studies) and to the way Java Virtual Machine (JVM) works. In the networked version, each tuple space is managed by an independent JVM, whereas in a single host version only one JVM is used.

[5]Docker is a software technology providing containers, promoted by the company Docker - `www.docker.com`

[6]Oracle VM VirtualBox is a free and open-source hypervisor for x86 computers from Oracle Corporation (`www.virtualbox.org`).

When evaluating *Matrix multiplication*, we have that the remote reading time exceeds significantly the one for the single host version (Table 6). In our view, this is an evidence of the fact that GIGASPACES does not use network protocols when all processes run on a single machine.

	Remote reading time	**Total time**
Single host version	60.0817	25701
Docker version	12.4407	28773
Networked version	12.2382	11112

Table 5: Sorting (host machines, 10×10^6 elements, 15 workers)

	Remote reading time	**Total time**
Single host version	0.0223	1577
Docker version	2.867	19641
Networked version	1.3340	13918

Table 6: Matrix multiplication (host machines, the matrix order is 100, 15 workers)

As shown in Tables 5 and 6, the Docker and the networked versions lead to the similar results. Indeed, their remote reading times differ very little for both case studies. The total execution time is lower for the network-based tests only because more powerful machines were used.

Discussion. The presented results provide us information on what are the critical aspects of a tuple space system that deserve specific attention to obtain efficient implementations. The critical choices are concerned with inter-process and inter-machine communication and local tuple space management.

In our experiments, we have varied the workload by changing the size of the input data; in this way, we have been able to track the different performances of the different tuple space systems. In our opinion, the results of our experiments are sufficient to provide a full account of all tuple space systems and increasing the workload would not introduce any significant difference.

The first aspect, concerned with communication, is related to data exchange and is influenced by the choice of algorithms that reduce communication. For instance, the commercial system GIGASPACES differs from the other systems that we considered for the technique used for data exchange, exploiting memory based inter-process communication, that guarantees a considerably smaller access time to data. Therefore, the use of this mechanism on a single machine does increase efficiency. However, when working with networked machines, it is not possible to use the same mechanism and we need to resort to other approaches (e.g. the TUPLEWARE one) to reduce inter-machine communication and to have more effective communications. To compare GIGASPACES with the other tuple space systems under similar conditions and thus to check whether it remains efficient also in the case of distributed computing, we have carried out experiments using a network where workers and masters processes are executed in separate environments (Docker containers, virtual machines). The results of these experiments show that, although the remote operations are much slower, in this case, performances of inter-process communication of GIGASPACES are similar those of KLAIM and MOZARTSPACES.

The second aspect, concerned with the implementation of local tuple spaces, is heavily influenced by the data structure chosen to represent tuples and by the corresponding data matching algorithms and also by the lock mechanisms used to prevent conflicts when accessing the tuple space. In our experiments, the performance of different operations on tuple spaces varies considerably. The performances of a tuple space system would depend also on the chosen system architectures which determine the kind of interaction between their components. Indeed, it is evident that all the issues should be tackled together since they are closely interdependent.

Chapter 5

Replicating for efficiency

In this chapter, we extend the tuple space paradigm with a sharing abstraction to improve data availability. In Section 5.1, we describe the sharing abstraction and how it is applied to tuple spaces. Section 5.2 sheds some lights on the impact of improving data availability on a tuple space implementation. Finally, Section 5.3 is devoted to the evaluation of our new implementation.

5.1 Sharing abstractions

In this section, we discuss how sharing abstractions can be added to tuple spaces and, by means of an example, we provide some intuition on how such abstraction can be beneficial when programming applications for the Internet of Things (IoT). One of the main problems with data availability is data consistency; we will also discuss how data consistency in presence of replication affects efficiency and correctness of the operations on tuple spaces.

5.1.1 Motivating example

To show the benefit of sharing abstractions for tuple spaces we consider a scenario borrowed from the IoT realm [92]. IoT can be seen as the extension of the Internet to the world of physical devices. Usually, an IoT system consists of a number of communicating devices that collect and exchange data, and coordinate their work over a distributed environment.

Let us consider a system that controls the average temperature in a house consisting of several rooms, each of them equipped with different devices: temperature sensors, alerts, and controllers. When the average temperature exceeds a certain threshold the system has to activate alerts in the whole house.

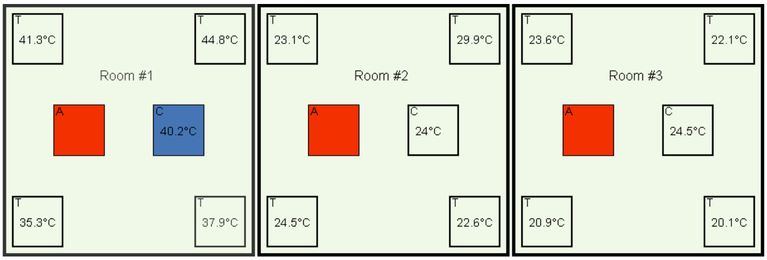

Figure 35: Smart home

Our scenario is presented in Figure 35. Temperature sensors of each room are situated at the four corners of the rooms and share information about the measured temperature with the other devices inside the room. A controller reads the information about temperature, computes the average value and, if the average is greater than a threshold, notifies the dangerous situation to the group of alert devices. We can distinguish two functional groups: the group of devices located in the same room and the group of alert devices and controllers. Only devices within the same group share information. For instance, sensors of the first room do not share data with the controller of the second room.

5.1.2 Description of the sharing abstraction

Appropriate abstractions are needed for programming distributed systems when data are shared not among all nodes of the network but only among specific groups. In our approach, when data have to be shared among a group of nodes, we select a subset of them on which the data is locally copied.

When implementing data sharing we distinguish three different levels: a *data-level* where data consistency is defined; an *operations-level* where operations on data are defined; and a *location-level* where data replicas are determined. This separation of concerns makes it easier to modify and tune an application by selecting the appropriate replication and consistency strategies.

The nodes of a network can be divided into several groups according to their communication pattern or functional meaning. It is assumed that groups should be determined by programmers. For instance, in our motivating example for the devices we have three "ROOM" groups, one for each room ("ROOM1", "ROOM2", "ROOM3") and an"ALERT_GROUP". An example of how several devices of the first room are initialized is shown in Listing 5.1: the controller belongs to both of them (line 2), whereas alerts and temperature sensors to just one group (lines 4, 6).

```
1   // for a controller
2   controllerNode.init(refAddress, {"ROOM1", "ALERT_GROUP"});
3   // for an alert
4   alertNode.init(refAddress, {"ALERT_GROUP"});
5   // for a temperature sensor
6   temperatureSensorNode.init(refAddress, {"ROOM1"});
```

Listing 5.1: Initialization of tuple spaces

From the programming point of view, using a single operation to access and modify data of several nodes makes programming clear and simple. Moreover, since our sharing mechanism relies on the use of replication, we may improve performance while hiding the complexity of its implementation.

When we consider data placement, our tuple space uses implicitly replication to make data more available. For each group, a node can be either a *replica node* which stores a tuple space for the group; or an *ordinary node* which does not store data but can perform operations which will be directed to a replica node. In Figure 36, an example of the network is presented where replica nodes are marked in green and ordinary nodes are marked in blue. The network consists of 6 nodes and 2 sharing groups *G1* and *G2*. Tuple spaces of each group are separated. However, every node can be a part of several sharing groups and, depending on how data are replicated, a node can be a replica node for one sharing group and an ordinary node for another one. For instance, in our example, the node *N3* is a replica node for the group *G1* and an ordinary node for the group *G2*.

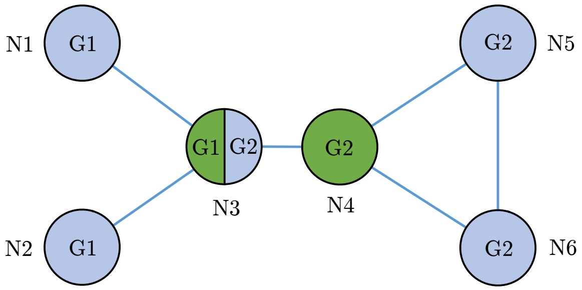

Figure 36: An example network

5.1.3 Operations on data and their consistency

Data consistency. We consider consistency model and operations on tuple spaces similar to the ones of `RepliKlaim` [76]. However, in our approach, the consistency is defined not on the level of operations but on the level of data which are used in applications. Thus, data consistency is defined separately from operations on these data and the standard Linda operations `out`, `rd` and `in` are not distinguished according to the data consistency type (strong, weak, . . .) like in `RepliKlaim`. In our approach, the consistency type is a property of tuples, and it determines how the tuple space performs operations. We consider two types of consistency: *strong* and *eventual* (weak) consistency.

Operations on data. Depending on the required consistency type, operations on a tuple space act differently. Operations with strong consistency are atomic and guarantee that the states of all the replicas are consistent immediately after the operation is performed. Operations with weak consistency are less strict as a replica can reach a consistent state after a certain operation is performed. Due to this difference, operations with weak consistency perform usually faster.

To explain how operations on data can be used, we consider the case of alert notification taken from the motivating example. Here, we provide code snippets to better describe the sharing and give a more structured description of how to use framework later. When the average temperature exceeds a certain threshold, one of the controllers sends an alert by sharing a tuple (Listing 5.2) with the group "ALERT_GROUP", strong consistency is required (line 1). Alerts are waiting for notification (Listing 5.3), with the blocking read operation (line 2) and proceed when they see the tuple.

```
1   Tuple alertTuple = new Tuple("alert", true);
2   alertTuple.setConsType(eConsistencyType.STRONG);
3   controllerLoc.out(alertTuple, "ALERT_GROUP");
```

Listing 5.2: Controller notifies alerts about an alert situation

```
1   Tuple template = new Tuple("alert", true);
2   boolean result = alertLoc.read(template, "ALERT_GROUP");
3   notifyAboutDangerousSituation();
```

Listing 5.3: Alert waits for an alert notification

Listing 5.4 shows a snippet where a temperature sensor of the first room updates its data about the current temperature. First, it tries to remove the previous data if they were shared using the non-blocking withdrawing operation *in_nb* (lines 2-3). Operation *in_nb* is a variant of *in* and it either finds a matching tuple right away and returns *true*, assigning values to variables in the template, or does not find it and returns *false* without doing any assignments. Then, it shares a new value of temperature using weak consistency (lines 5-7). In both cases, the shared group is "ROOM1".

```
1   // remove the previous data of the sensor if they exist
2   Tuple template = new Tuple("temperature", sensorID, Double.class);
3   boolean result = tempSensorLoc.in_nb(template, "ROOM1");
4   // write a current data of the sensor
```

```
    Tuple tuple = new Tuple("temperature", sensorID, currentTemp);
    tuple.setConsType(eConsistencyType.WEAK);
    tempSensorLoc.out(tuple, "ROOM1");
```

Listing 5.4: A temperature sensor updates its data

The writing operation `out(t)@G` (where `G` is a group of locations sharing data) with strong consistency terminates only when the tuple t is added to all replica nodes. It is implemented as follows: the process performing the writing operation sends a tuple t to the replica node it is connected to, then, this replica node disseminates t to other replica nodes and sends the notification to the process when the dissemination is finished. In case of weak consistency, the replica node sends a notification to the process immediately after putting t into its local tuple space and only after it sends a copy of t to other replica nodes. In both cases, the dissemination of the tuple and the waiting for the confirmation run in parallel. An example of how the writing operation is defined and operates is shown in Figure 37. Three levels are indicated: a data-level where a tuple with weak consistency is defined, an operation-level where *out* operation is called and a location-level where the network overlay is defined. On the location-level, the node writes to the replica *R1* and immediately receives a confirmation "conf" while the replica *R1* distributes a copy of the tuple to other replicas *R2* and *R3*. Using dashed lines we demonstrate the difference between tuples with weak and strong consistencies: writing a tuple with weak consistency we do not need to wait until all actions (dashed lines) are finished.

Withdrawing a tuple requires a mechanism that guarantees that only one copy can be taken when several concurrent processes ask for it at different replica nodes. For each sharing group, there is a *primary replica node*. This node acts as a judge when two or more concurrent processes try to withdraw the same tuple. The primary replica node is similar to *the owner* of the tuple used in `RepliKlaim`. However, since we aim to work with dynamic networks where nodes can enter and leave the network, we cannot rely on a fixed node, since it might disappear. To this end, the primary replica node may change over time: the only requirement is that for each sharing group there should be only one primary replica

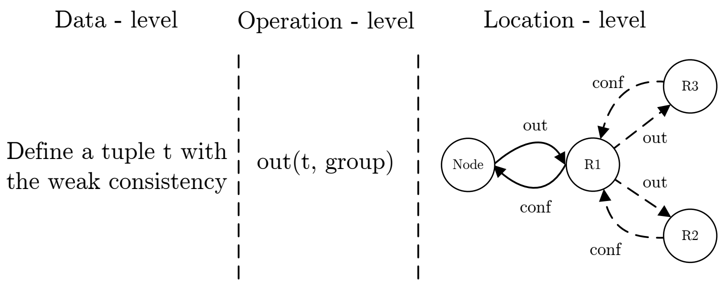

Figure 37: The writing operation

node. When one of the replicas wants to withdraw a tuple it asks the primary replica node whether it is possible: one of the requests will be accepted, whereas the others will be discarded and the issuing processes have to search for another matching tuple. If the withdrawn tuple has strong consistency, it is returned to the process only after all its copies have been removed from the replica nodes they are stored in. In case of weak consistency, the tuple is returned immediately and its copies are removed asynchronously. An example of how withdrawing operations do work is shown in Figure 38. First, a node asks the closest replica *R1* to withdraw a certain tuple. When this node *R1* finds a matching tuple, it asks a primary replica node *PR* whether this tuple exists. Then, node *PR* removes a copy of the tuple and sends requests to remove this tuple from other replicas. In case of a tuple with strong consistency actions denoted with dashed lines should be finished before the node *PR* sends a confirmation to the node *R1*.

Reading operations do not change tuple spaces, so they behave the same regardless of the consistency type: a process asks a replica and this replica searches only in its own tuple space without involving the primary replica. Hence, the reading time for a node depends on the location of the closest replica.

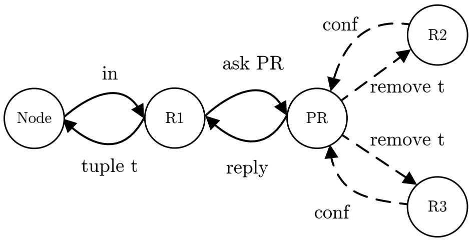

Figure 38: The withdrawing operation

5.2 Implementing replication

In this section, we provide additional details about the actual implementation of the tuple space. In particular, we talk about a network overlay, a routing table, and replication strategies. At the end, we provide a short guide of how to use our new framework of tuple spaces with sharing.

Figure 39 shows the main components of our framework and the way they interact. The block of tuple spaces and their operation is responsible for data storing and data manipulation. Replica placement strategies specify rules for computing the set of locations where it is more beneficial to store data. Network overlay is responsible for maintaining information about replicas (routing table) and gathering data about the network.

Network overlay. The replication mechanism depends on the overlay network designed to coordinate nodes that share data. The possible operations are concerned with main operations that can be done on this network and are summarized below:

- connecting to the network;
- leaving the network;
- determining the replica placement;
- adding or removing a replica.

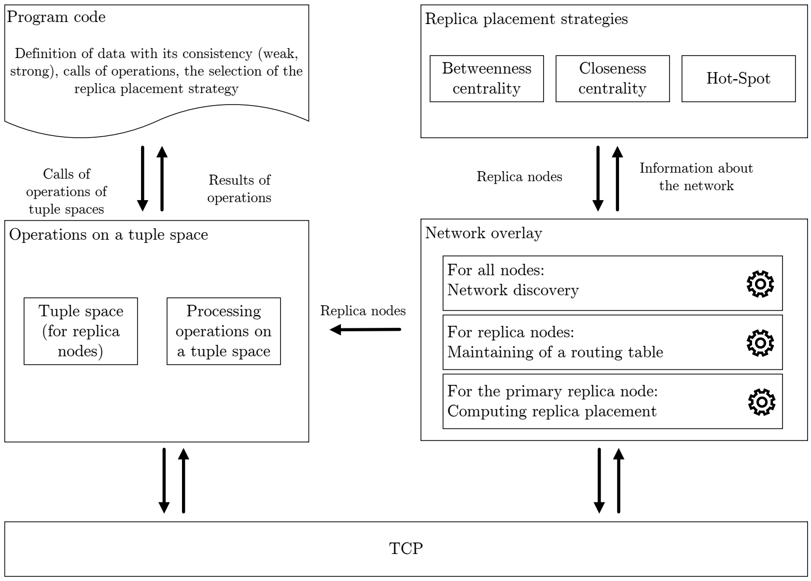
Program code
Definition of data with its consistency (weak, strong), calls of operations, the selection of the replica placement strategy
Replica placement strategies
Betweenness centrality
Closeness centrality
Hot-Spot
Calls of operations of tuple spaces
Results of operations
Replica nodes
Information about the network
Operations on a tuple space
Tuple space (for replica nodes)
Processing operations on a tuple space
Replica nodes
Network overlay
For all nodes: Network discovery
For replica nodes: Maintaining of a routing table
For the primary replica node: Computing replica placement
TCP

Figure 39: The diagram of components

A node joining the network has to declare to which sharing groups it wants to belong. The procedure of finding replica nodes consists of several steps. The pseudocode of this procedure is presented in Figure 40. If a node uses several groups, it has to follow this procedure for each of them. When a node connects to the existing network, first, it has to obtain information about existing replica nodes (line 2) by getting this information from any known node of the network. Then, for each group, the new node chooses one replica node and notifies its choice by connecting to it (lines 6-7). In our implementation, the node chooses the closest replica. Since the chosen replica node is aware of a new node of the network, the information about the node can be taken into account while computing the optimal replica placement for the sharing group. For each sharing group, every replica node knows all connected nodes and according to the replication strategy can decide whether the replica should be moved to another node or it is, instead, necessary to add or remove a replica. If there are no replica nodes or it is not possible to connect to any of them, the node becomes a replica node (line 4).

```
1: function FINDREPLICANODES($referenceAddress$, $group$)
2:     $replicaList$ = GETREPLICANODES($referenceAddress$, $group$)
3:     if $replicaList$.empty() then
4:         ESTABLISHREPLICANODE()
5:     else
6:         $replicaAddress$ = CHOOSEREPLICA($replicaList$)
7:         CONNECTTOREPLICANODE($replicaAddress$)
8:     end if
9: end function
```

Figure 40: Algorithm. Connect to the network

Routing table. Each replica node maintains a routing table that contains the list of records about each group with all necessary information. As shown in Figure 41, the record of a group contains the following information: the addresses of replica nodes for the group, the address of the primary replica, the time stamp indicating when it was last modified, a hash value of the group information that is used to check if this information is changed.

group name	- replica addresses
	- primary replica
	- time stamp
	- hash value of group's content

Figure 41: The record of the routing table

Periodically replica nodes ask other nodes for an update of the routing table. Each group has only one *primary replica node* that is assigned two additional tasks. First, it is responsible for finding an optimal replica placement for its group and has to notify the other nodes when the current configuration of replica nodes changes. Second, it has to play the *arbiter* role for withdrawing operations when several processes look for the same tuple. Replica nodes and the primary replica node are selected using the information about all nodes of the group and according to the chosen replication strategy.

Replica placement and replica migration. Over time, network topology and nodes features might change, hence, performances, such as the network throughput, might worsen due to the fact that replica placement may not be optimal anymore. Thus, it is necessary to periodically check the configuration of replica nodes. This requires two independent actions, namely **network discovery** and **replica placement computation**. For network discovery, every replica node collects information about the connected nodes and shares it with the primary replica node. For instance, to collect information about latencies in the network, a node can ask all replicas to provide a set of nodes to examine, then the nodes measure latencies of connections with them and return this information to the replica. In large networks, it is difficult to examine latencies between all pairs of nodes, so, often only a subset of them is considered.

To avoid conflicts between different versions of the routing table, only the primary replica node is responsible for computing replica placements. It asks replica nodes to provide available network information, computes the replica nodes on the basis of available information of sharing nodes and changes it if necessary. If a new replica placement is different to the current one, it is necessary to introduce new or remove old replicas. When a node becomes a replica node, it does receive a copy of the tuple space and the updated routing table in order to be enabled to receive and process operations on the tuple space. When a node stops being a replica node, the coordination information stored on it are removed and it has to choose a replica to connect. All ongoing requests which were sent to it have to be redirected to other replica nodes. All nodes previously connected to the "was" replica node have to choose a new closest replica.

Replication strategies. To determine the set of the nodes where replicas should be placed, we introduce a replication strategy that is a function $S : N \times Inf \to N$ that receives as input a set of nodes and information about them and returns a set of nodes where data should be placed.

Exact locations of the replicas determined by the replication strategy are hidden to programmers. All nodes sharing the same data should use the same strategy. In our implementation, the node (represented by the `KlavaNode` class) has a parameter 'replication strategy' that has to be set during the initialization phase. A strategy can be any class which implements a specific interface, whose main method returns a set of replica nodes that is ordered by scores; the node with the highest score becomes a primary replica node.

Replication strategies can exploit different information such as node availability, computational performance, and so on. In this work, we consider strategies for decreasing the average data access time by considering information on latencies between nodes in the network. During their work, all nodes collect the data about the latencies between nodes and send them to the replica nodes. Replica nodes share this data with the primary replica that calculates the optimal replica placement. We rely on the following metrics and techniques:

- *Betweenness centrality* [77] that is based on the number of shortest paths that pass through each vertex.
- *Closeness centrality* [78] that is based on the sum of the shortest paths between the vertex and all other vertices in the graph.
- *Hot-Spot* [27] that aims at placing replicas close to the nodes that generate the greatest load. In our case, the parameter of the load is the total number of read/write operations of the node.

The default number of replicas equals $\lceil \lg n \rceil$ where n is the number of nodes in the group. For betweenness and closeness centralities, the node with the highest centrality value becomes the primary replica node. For Hot-Spot, the primary replica node is the one that performs the highest number of operations.

A short guide to using tuple spaces with data sharing. Although the underneath implementation is quite different, the application interface of our tuple space with sharing is pretty similar to one described in

Chapter 3. In what follows, we describe how the tuple space with sharing can be used while highlighting changes in the application interface of the framework.

To connect to the network, a new node has to specify (i) the sharing groups, (ii) at least one address to connect to the sharing network, (iii) the replication strategy for replica placement. Profiles of replication aggregate this information and define how data are stored and distributed. Each profile corresponds to a certain replication group and is identified by its group name. Each node can have several profiles that have to be specified while initializing the node.

The object of the `ReplicationProfile` class represents a replication profile; it contains:

- a group name that is an identifier of the profile.
- a replication strategy that defines how data are distributed. An object of any class that extends IReplicationStrategy abstract class can be used. The strategy allows setting ‘the number of replicas.
- an implementation of tuple spaces that can be any class that implements `ITupleSpace` interface.

```
// define a  locality  and a node
Physical  locality  = new PhysicalLocality(address);
node = new KlavaNode(locality);
// defining replication  strategies
BetweennessStrategy syncStrategy = new new BetweennessStrategy();
syncStrategy.setReplicaNumber(2);
BetweennessStrategy dataStrategy = new new BetweennessStrategy();
dataStrategy.setReplicaNumber(10);
// defining replication  profiles
ReplicationProfile  syncProfile  = new ReplicationProfile("syncGroup",
syncStrategy, TupleSpaceHashtable.class);
ReplicationProfile  dataProfile  = new ReplicationProfile("dataGroup",
dataStrategy, IndexedTupleSpace.class);
node.init(refAddresses, Arrays.asList({syncProfile, dataProfile}));
```

Listing 5.5: An example with two replication profiles

An example with two replication profiles is presented in Listing 5.5. In the beginning, we define a physical address of the node (line 2), a node itself (line 3), and two replication strategies (lines 5-8). The first profile (lines 10-11) is associated with the group “syncGroup”, the sharing group managing data for synchronization primitives, such as counters and semaphores. The data used by these groups should have strong consistency and possibly a small number of replicas to keep the data access time low. The second profile (lines 12-13) is associated with the group “dataGroup”, the group of nodes that shares some data that can have weak consistency. In the end, profiles are used for the initialization of the node (lines 14).

Operations on tuple spaces with sharing are the same as for ordinary tuple spaces. The difference lies in how we define localities in programming codes. Instead of localities we pass a string name of a group (see Listing 5.6) that refers to a dynamically changeable collection of localities.

```
void out(Tuple tuple, String groupName);
void rd(Tuple template, String groupName);
void in(Tuple template, String groupName);
void rd_nb(Tuple template, String groupName);
void in_nb(Tuple template, String groupName);
```

Listing 5.6: Signatures of operations for the tuple space with sharing

Tuples are characterized by consistency level that has to be set when tuples are emitted via `out` operation (see Listing5.7). Programmers can choose to set either weak consistency (this is set by default) or strong consistency. Consistency level is not specified for templates but it is taken into account when matching tuples and managing copies of tuples.

```
Tuple tuple = new Tuple("counter", 10);
tuple.setConsType(eConsistencyType.STRONG);
node.out(tuple, "SYNC");
```

Listing 5.7: An example of the writing operation `out`

While initializing a node, it is possible to choose a replication strategy among the following classes: `BetweennessStrategy`, `ClosenessStrategy`, `HotSpotStrategy`. In Listing 5.8, an example of how to define a replication strategy is shown. Additionally, it is possible to set a maximum number of the replicas for each group using method `setReplicaNumber` (a default value is $\lceil \lg n \rceil$ where n is a number of nodes in a group). It is worth noting, that the number of replicas should be set once and has to be the same for all nodes of a group.

```java
IReplicationStrategy strategy = new BetweennessStrategy();
strategy.setReplicaNumber(1);
```

Listing 5.8: Initialization of a replication strategy

To introduce a new replication strategy it is necessary to implement methods of the `IReplicationStrategy` abstract class. As shown in Listing 5.9, the main methods are `setData`, `getReplicaNodes`, and `getPrimaryNode`. Method `setData` loads data to be processed. Up to now, we have only considered nodes that collect information about latencies between nodes and, thus, our replication strategies are based only on this kind of information. Method `getReplicaNodes` computes a number of replica nodes and receives as parameter a subset of the nodes of the network. Method `getPrimaryNode` determines a primary replica among replica nodes.

```java
// set data to be processed
public abstract void setData(NetworkDiscoveryInfo data);
// get replica nodes
public abstract ArrayList<String> getReplicaNodes(Set<String> nodes);
// get primary replica
public abstract String getPrimaryNode(ArrayList<String> replicaNodes);
// set a custom number of replicas
public abstract void setReplicaNumber(int numberOfReplicas);
```

Listing 5.9: Methods of the `IReplicationStrategy` abstract class

5.3 Evaluation of the implementation

In this section, we assess the implementation of the tuple space with sharing. We start by describing the IoT case study used to evaluate the proposed tuple space. Then, we describe the experiments we conducted listing the parameters we varied and the measured metrics. In the end, we present the results of our experiments.

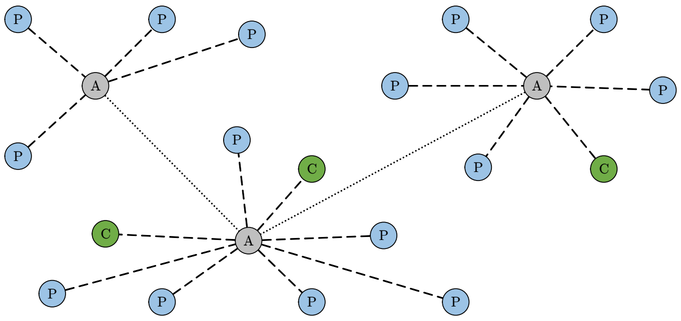

Figure 42: Sensor network

5.3.1 Case study

To evaluate the performance of the proposed tuple space and compare different replica placement strategies we use a generalization of the case study presented in Section 5.1. This generalization allows us to be parametric with respect to the network topology. We consider the network of mobile homogeneous sensors with two types of devices: producers (P) that generate information in the network, controllers (C) that collect and process the information generated by producers (Figure 42). The physical network relies on a number of antennas A, spread across an area, that transmit signals.

In the experiments, for each group, we consider one controller and many producers. Periodically, producers generate information and the controller performs most of the data retrieval operations. Therefore, the number of querying operations that the controller performs becomes important. To evaluate its performance in our experiments we consider two cases: (i) the controller just reads the information provided by producers and (ii) it withdraws this information. For each run, the controller is placed in a random node of the network.

5.3.2 Network topologies

To simulate networks we have chosen 2 representative classes of graphs: *scale-free* [93] and *random* graphs, based on the Erdös-Rényi model [94]:

- Undirected scale-free graph, a graph with a power law degree distribution of nodes (parameters[1] α = 0.49, β = 0.02, γ = 0.49). For our experiments, it is important that scale-free graphs create *hubs*, i.e., nodes that have much more connections than others.

- Random graph with the probability of an edge to be present that equals $2 * \ln n/n$.

Scale-free graphs are well-suited for representing Internet topologies, while the second ones are usually used to model complex networks. Each graph has 50 vertices. We have chosen this number since it is close to the maximum number of physical threads that the machine we used for experiments can have. Experiments with bigger graphs lead to results with a very high standard deviation, making their interpretations difficult. The weight of each edge is randomly chosen in the range $[15, 45]$. The weight represents a network latency between nodes in milliseconds. In this way, each tuple space performs requests and replies with a delay corresponding to the latency between nodes.

5.3.3 Assessment methodology

Execution environment. We conducted our test on an Ubuntu server (version 14.04.5 LTS) with 4 processors Intel Xeon E5-4607, each with 6 cores, 12 M Cache, 2.20 GHz, and with hyper-threading, offering in total 48 threads and 256 GB RAM.

Measured metrics. For our evaluation, we collect the following measures:

- Writing time: the time required to write a tuple into a shared tuple space.

[1]α and γ are probabilities for adding a new node connected to an existing node chosen randomly according to the in-degree and out-degree distributions correspondingly; β is the probability of adding an edge between two nodes.

- Reading time: the time required to read a tuple from a shared tuple space using a template.
- Withdrawing time: the time required to withdraw a tuple from a shared tuple space using a template.

Since the numbers of operations performed by a producer and a controller differ significantly, we measure separately the reading and withdrawing times for the producers and the controller. We conducted each set of experiments 20 times and computed the average value and the standard deviation for each metrics.

For the case study, we considered the following parameters:

- Number of replicas: $r \in \{1, 5, 10, 25, 50\}$. When we talk about the number of replicas, we mean the number of data instances.
- Graph of the network: scale-free graph, random graph.
- Replication strategy: betweenness centrality, closeness centrality and Hot-Spot algorithm, but also the random strategy that selects replica nodes randomly for each run[2].

The computation of the betweenness and the closeness centralities is done by means of the JUNG library[3].

5.3.4 Results

In all plots we use the following abbreviations for the replica placement strategies: **BC** stands for betweenness centrality, **CC** stands for closeness centrality, **HS** stands for the Hot-Stop strategy and **RS** stands for the random strategy. The postfix **-w** stands for operations with weak consistency and the postfix **-s** stands for operations with strong consistency. The abbreviation **LB** stands for an ordinary location-based tuple

[2]Since we have only one node which performs most of the operations, we combined Hot-Spot approach with the strategy based on the betweenness centrality and, thus, the primary replica is always the node where the controller resides.

[3]JUNG - Java Universal Network/Graph Framework. The library is available from `www.jung.sourceforge.net`.

space (without sharing and based on explicit locations). Using this tuple space, all processes write their data locally and look for other data by visiting all other location one by one.

First, we consider the performance of operations performed by producers and, then, we evaluate the performance of operations performed by the controller.

Figures 43-44 show that with strong consistency and with strategies based on betweenness and closeness centralities the writing time grows monotonously with the number of replicas. With weak consistency, the writing time is generally lower than with strong consistency for all strategies and decreases as the number of replicas increases.

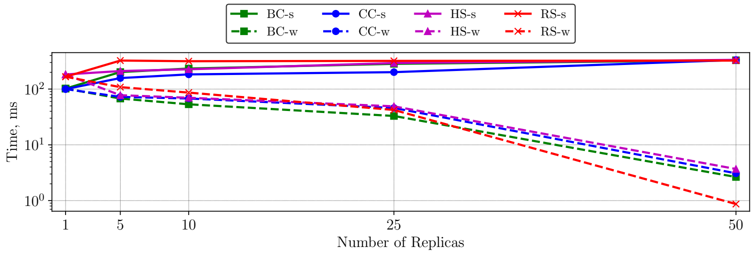

Figure 43: Writing time (scale-free graph)

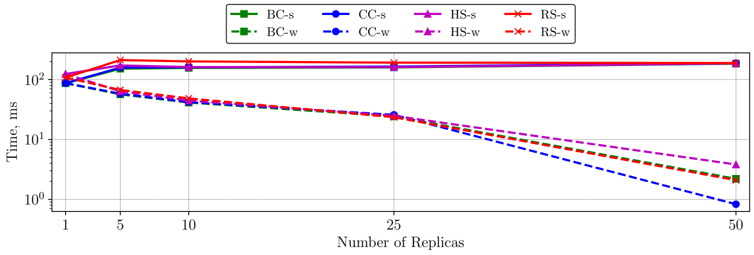

Figure 44: Writing time (random graph)

It is worth noting that, when computing betweenness and closeness centralities on our graphs, the sets of the nodes with the highest score turns out to be similar. In all experiments, the node with the highest score is the same one and this explains why these two strategies behave identically when only one replica is considered.

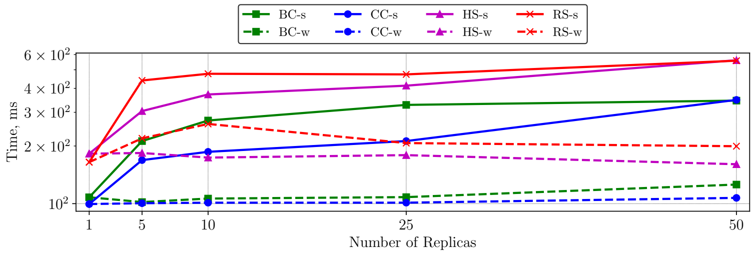

Figure 45: Withdrawing time (scale-free graph)

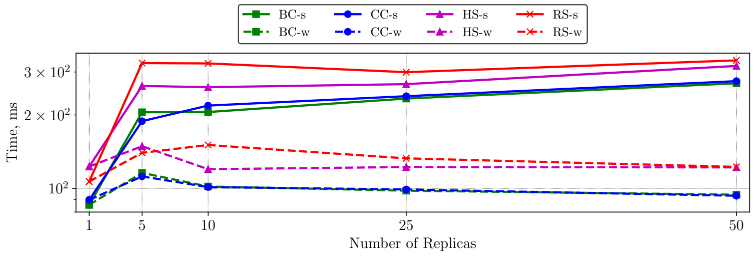

Figure 46: Withdrawing time (random graph)

Figures 45-46 show that the reading time does not change much when the number of replicas increases with any of the strategies. For the Hot-Spot one, the reading time is similar to that of the random strategy because a primary replica is placed where the controller process resides, and it is chosen randomly. For the withdrawing operation, the location of the primary replica is the one that determines mostly the time required to perform it. Considering strong consistency, the withdrawing time mostly grows for all the strategies except for the random one.

The reading operations for weak and strong consistencies are the same and their duration depends only on where the closest replica is placed.

Figures 47-48 show that the biggest difference between the strategies is when 1 and 5 replicas are considered. Indeed, the fewer replicas are used, the more actual data locations affect the average access time. This is the reason why strategies based on betweenness and closeness centralities perform better. When the number of replicas is higher than 5, the difference between the strategies is less evident. When the number of replicas increases we have that the probability to have a closer replica is higher and thus the reading time decreases.

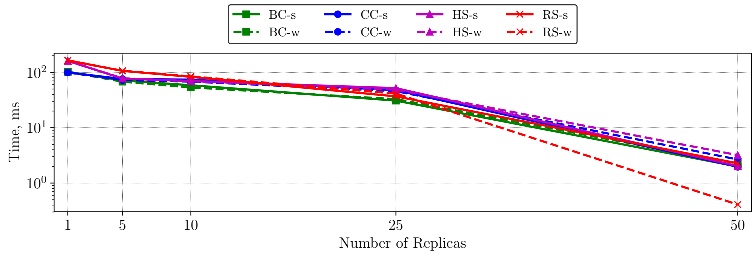

Figure 47: Reading time (scale-free graph)

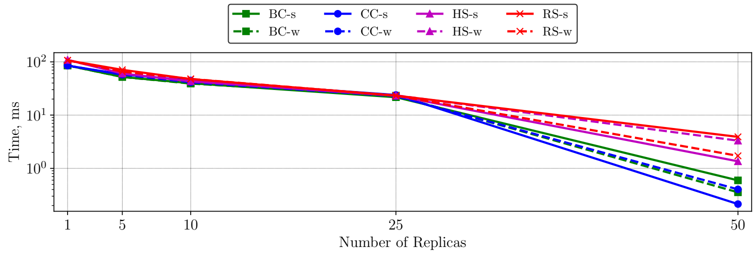

Figure 48: Reading time (random graph)

Performances of the querying operations of the controller were measured separately. The results for the strategies based on betweenness and closeness centralities and the random strategy are similar to those exhibited by producers.

With the Hot-Spot strategy, the reading operation is local for the controller because a replica is placed in its node; thus, the reading time is very low. The withdrawing operation with weak consistency is very fast because the replica in the node of the controller is a primary replica (Figures 49-50). With strong consistency, the withdrawing time of the Hot-Spot strategy is similar (Figure 49) or better (Figure 50) than the one exhibited with the strategies based on betweenness and closeness centralities.

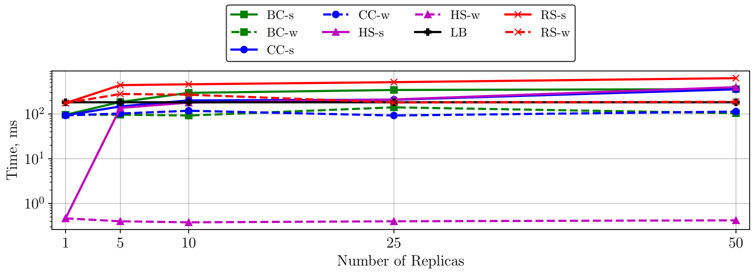

Figure 49: Withdrawing time (controller, scale-free graph)

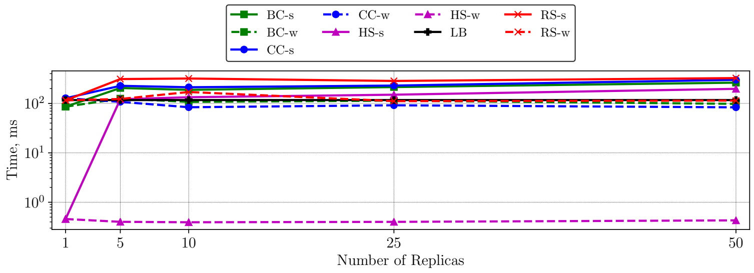

Figure 50: Withdrawing time (controller, random graph)

To evaluate the impact of the data sharing, we considered also the implementation of the same case study by relying on a tuple space using explicit locations but without data sharing. Each producer and controller has its own tuple space. All writing operations are local and to search for data it is necessary to use a specific address of the tuple space. Therefore,

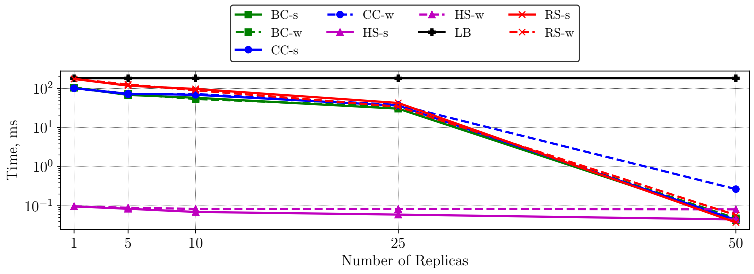

Figure 51: Reading time (controller, scale-free graph)

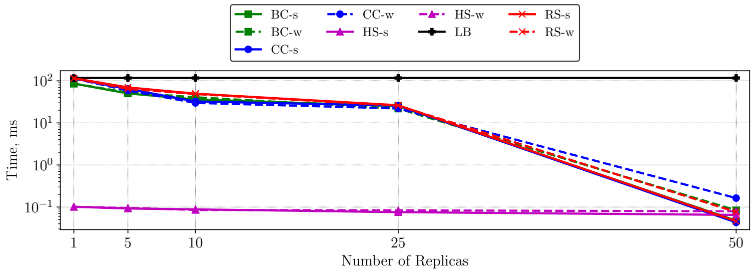

Figure 52: Reading time (controller, random graph)

to collect the information provided by producers, the controller has to visit all their tuple spaces. Since data are not replicated, there is no distinction between strong and weak consistencies and the number of replicas is also not considered. From Figures 49-50 it is evident that it takes more time to perform withdrawing operations with strong consistency for tuple spaces with sharing than for the locations-based version when the number of replicas exceeds a given threshold (5 for the scale-free graph and 1 for the random graph). However, with weak consistency, withdrawing operations take less time for all the strategies except the random one. This is more evident when the scale-free graph is used. In our experiments, the reading time is very similar to the withdrawing time. And, hence, taking into account Figures 47-48, the reading time of the locations-based version is higher than the reading time of tuple spaces with data sharing (see Figures 51-52).

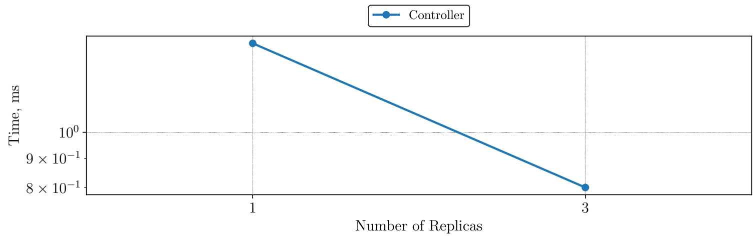

Figure 53: Reading time of controller (smart home case study)

To validate the results of our experiments, we have also implemented the smart home scenario described in Section 5.1 using several machines. In the case study, we do not distinguish groups of rooms but have a single group for all rooms. We deployed the application in this way: the code of each room (i.e., the code of the devices) has been placed on a different server, and the scenario consists of a cluster of three servers/rooms. For the test, we considered 1 and 3 replicas (that corresponds to the number of rooms) and the strategy based on the betweenness centrality. We measured the reading time for the controller (see Figure 53) and the results are in line with the ones of our simulations.

Chapter 6

Conclusion

Communication among the components is a crucial point when designing a distributed system. The choice of tools and their implementations can seriously impact the overall performance of the system. This work has been devoted to the evaluation of tuple spaces, that can be seen as an associative shared memory, as a suitable tool for building distributed systems. As a representative of the indirect communication, the tuple space model differs significantly from approaches of other communication paradigms, such as interprocess communication. Programming simplicity, time- and space- decoupling are the main features of tuple spaces that greatly facilitate programming of interactions between components of a distributed system. The price to pay for these features is in terms of performances.

We assessed a number of tuple space's implementations. First, we preselected some of them on the base of their description and, then, experimentally evaluated their performances and discussed how certain techniques and approaches affect performances of tuple spaces and which of them are more important than others. In particular, we focused on the cost of pattern matching and communication.

We presented an improved implementation of tuple spaces as a middleware. The implementation of KLAIM, KLAVA, was taken as the starting point. The main aim was to improve its pattern matching algorithm. The previous version was obsolete and not suitable to be used for building modern applications. The main concerns were related to the performances of communication and pattern matching. Thus, we implemented several tuple spaces based on different data structures and data querying techniques that considerably improved the performances of data retrieving operations on tuple spaces and their throughput. We added the possibility for programmers to choose which implementation of tuple spaces to use. Such choice would depend on the data and the patterns used in the specific applications. We improved also the parts related to interprocess communication and added a number of new operations that aim at reducing the number of tuple space's accesses. The new implementation was shown to outperform the old one on several aspects; in particular, it exhibited faster pattern matching and faster communication. We also compared the performance of our new implementation with several others, namely GIGASPACES, MOZARTSPACES, and TUPLEWARE.

Then, we have extended our implementation with data sharing abstraction. Sharing simplifies programming communication among several processes: the programmer specifies only a group of nodes that share data and do not care where data are actually stored. At the same time, the middleware guaranteeing the sharing makes it possible to optimize the actual placement of data to improve average access time. The new implementation allows specifying how to distribute (via replication strategies) data, how data have to be stored (by selecting the appropriate data structure) and what consistency model to use for them. Each interaction among nodes can be described in terms of sharing groups. Information about each group is aggregated in replication profiles that programmers have to define in the code. Moreover, programmers specify sharing groups and at run-time the middleware optimizes the data placement adjusting it to certain network characteristics. The description of the abstraction and the implementation of the tuple space with sharing were accompanied by experiments used to validate proposed techniques. The results of the experiments, where we varied different replication strategies, consistency models and the number of replicas, confirmed our expectations. By choosing each of these parameters, it is possible to improve performances by resorting to programming with sharing rather than ordinary locality-based programming.

Directions for future works. We are considering several directions for future work. They are concerned with the extension of the current implementation, its improvement to meet the requirements of modern distributed systems, and with the development of methodologies that make it easier for programmers to tune applications using our framework.

First, programming facilities make it easy to extend our framework with new replication strategies, implementations of tuple spaces, and different kinds of consistency. Providing richer tools to programmers allows tuning applications more precisely. Talking about replication strategies, one possibility we are considering is to add a fault tolerance strategy that can cope with a predetermined (parametric) number of failures. Similarly, new implementations of tuple spaces can be added, e.g. the one that represents a queue or a stack, that can be naturally used in applications that tend to emulate these data structures on top of canonical tuple spaces.

Another direction is to implement additional features and requirements that modern distributed systems demand. The first one is data security. In fact, access control and secure communication channels are standard requirements of many distributed systems that work over the Internet. The second one is scalability: the algorithm for choosing replicas placement can become a bottleneck that prevents applications to be scalable. In the context of our work, it is necessary to find a balance between the amount of network information to be analyzed and the quality of replica placement.

The last (but not least) direction is to help programmers in their decision on how to use the available set of techniques provided by the framework. We believe that this process can be automated, according to the specific application; the developer could be provided with the appropriate combination of tools which will optimize the overall performance of his application.

References

[1] Vitaly Buravlev, Rocco De Nicola, and Claudio Antares Mezzina. Tuple spaces implementations and their efficiency. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, pages 51–66, 2016. doi: 10.1007/978-3-319-39519-7_4. URL `https://doi.org/10.1007/978-3-319-39519-7_4`. xvi

[2] Vitaly Buravlev, Rocco De Nicola, and Claudio Antares Mezzina. Evaluating the efficiency of linda implementations. In *Concurrency and Computation: Practice and Experience 2017*. John Wiley and Sons, 2017. doi: 10.1002/cpe.4381. URL `https://doi.org/10.1002/cpe.4381`. xvi

[3] Vitaly Buravlev, Rocco De Nicola, Alberto Lluch Lafuente, and Claudio Antares Mezzina. Improving availability in distributed tuple spaces via sharing abstractions and replication strategies. In *PDP 2018*, 2018. xvi

[4] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and designs (5. ed.)*. International computer science series. Addison-Wesley-Longman, 2012. ISBN 978-0-13-214301-1. 1, 11

[5] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN 978-0-13-239227-3. 2

[6] Davide Balzarotti, Paolo Costa, and Gian Pietro Picco. The lights tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems*, 5(2):215–231, 2007. URL `http://content.iospress.com/articles/web-intelligence-and-agent-systems-an-international-journal/wia00114`. 4

[7] Nadia Busi, Cristian Manfredini, Alberto Montresor, and Gianluigi Zavattaro. Peerspaces: Data-driven coordination in peer-to-peer networks. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA*, pages 380–386, 2003. doi: 10.1145/952532.952608. URL `http://doi.acm.org/10.1145/952532.952608`. 4, 33, 34

[8] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *23rd ICDCS Workshops, USA*, pages 342–347, 2003. 4, 34

[9] Stefan Craß, eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *International Database Engineering and Applications Symposium (IDEAS 2009)*, pages 301–306, 2009. 4, 5, 20, 25, 28

[10] N. Carriero and D. Gerlernter. Tuple analysis and partial evaluation strategies in the linda precompiler. *Languages and Compilers for Parallel Computing*, pages 114–125, 1990. 4

[11] Antony I. T. Rowstron and Alan Wood. Bonita: A set of tuple space primitives for distributed coordination. In *30th Annual Hawaii International Conference on System Sciences (HICSS-30), 7-10 January 1997, Maui, Hawaii, USA*, pages 379–388, 1997. doi: 10.1109/HICSS.1997.667285. URL `https://doi.org/10.1109/HICSS.1997.667285`. 4

[12] Charles J. Fleckenstein, Helen Gill, David Hemmendinger, Carolyn McCreary, John D. McGregor, Roy P. Pargas, Arthur M. Riehl, and Virgil Wallentine. Multiprocessing. *Advances in Computers*, 35:255–324, 1992. doi: 10.1016/S0065-2458(08)60597-5. URL `https://doi.org/10.1016/S0065-2458(08)60597-5`. 4

[13] A.K. Atkinson. *Tupleware: A Distributed Tuple Space for the Development and Execution of Array-based Applications in a Cluster Computing Environment*. University of Tasmania School of Computing and Information Systems thesis. University of Tasmania, 2010. 5, 18, 25, 29, 57

[14] Marcus Amorim Leal, Noemi Rodriguez, and Roberto Ierusalimschy. Luats âĂŤ a reactive event-driven tuple space. *Journal of Universal Computer Science*, 9(8):730–744, 2003. 5, 18, 25, 27

[15] Yi Jiang, Zhaoqing Jia, Guangtao Xue, and Jinyuan You. Dtuples: A distributed hash table based tuple space service for distributed coordination. *Grid and Cooperative Computing, 2006. GCC 2006. Fifth International Conference*, pages 101–106, 2006. 5, 22, 25, 28

[16] GigaSpaces. Concepts - XAP 9.0 Documentation - GigaSpaces Documentation Wiki. `http://wiki.gigaspaces.com/wiki/display/XAP9/Concepts`. [Online; accessed 15-September-2016]. 5, 20, 25, 27

[17] R.A. Van Der Goot. *High Performance Linda Using a Class Library*. PhD thesis. Erasmus Universiteit Rotterdam, 2001. 5, 20, 25

[18] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2): 51–59, 2002. doi: 10.1145/564585.564601. URL `http://doi.acm.org/10.1145/564585.564601`. 6

[19] Alysson Neves Bessani, Eduardo Adílio Pelinson Alchieri, Miguel Correia, and Joni da Silva Fraga. Depspace: a byzantine fault-tolerant coordination service. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 163–176, 2008. doi: 10.1145/1352592.1352610. URL `http://doi.acm.org/10.1145/1352592.1352610`. 6, 31, 32

[20] Roberta Barbi, Vitaly Buravlev, Claudio Antares Mezzina, and Valerio Schiavoni. Block placement strategies for fault-resilient distributed tuple spaces: An experimental study - (practical experience report). In *DAIS 2017*, volume 10320 of *LNCS*, pages 67–82. Springer, 2017. ISBN 978-3-319-59664-8. doi: 10.1007/978-3-319-59665-5_5. 6, 35

[21] David E. Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Trans. Parallel Distrib. Syst.*, 6(3):287–302, 1995. doi: 10.1109/71.372777. URL `https://doi.org/10.1109/71.372777`. 6, 33

[22] Karpjoo Jeong and Dennis E. Shasha. Plinda 2.0: A transactional/checkpointing approach to fault tolerant linda. In *13th Symposium on Reliable Distributed Systems, SRDS 1994, Dana Point, California, USA, October 25-27, 1994, Proceedings*, pages 96–105, 1994. doi: 10.1109/RELDIS.1994.336905. URL `https://doi.org/10.1109/RELDIS.1994.336905`. 6, 33

[23] David Mosberger. Memory consistency models. *Operating Systems Review*, 27 (1):18–26, 1993. doi: 10.1145/160551.160553. URL `http://doi.acm.org/10.1145/160551.160553`. 6

[24] Pavlin Radoslavov, Ramesh Govindan, and Deborah Estrin. Topology-informed internet replica placement. *Comp. Commun.*, 25(4):384–392, 2002. 6, 35

[25] Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. Latency-driven replica placement. In *2005 IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2005), 31 January - 4 February 2005, Trento, Italy*, pages

399–405, 2005. doi: 10.1109/SAINT.2005.37. URL https://doi.org/10.1109/SAINT.2005.37. 6, 35

[26] Pierre Matri, Alexandru Costan, Gabriel Antoniu, Jesús Montes, and María S. Pérez. Towards efficient location and placement of dynamic replicas for geo-distributed data stores. In *Proceedings of the ACM 7th Workshop on Scientific Cloud Computing, ScienceCloud@HPDC 2016, Kyoto, Japan, June 1, 2016*, pages 3–9, 2016. doi: 10.1145/2913712.2913715. URL http://doi.acm.org/10.1145/2913712.2913715. 6, 35

[27] Lili Qiu, Venkata N. Padmanabhan, and Geoffrey M. Voelker. On the placement of web server replicas. In *IEEE INFOCOM 2001*, pages 1587–1596, 2001. doi: 10.1109/INFCOM.2001.916655. 6, 35, 94

[28] Jussi Kangasharju, James W. Roberts, and Keith W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4): 376–383, 2002. doi: 10.1016/S0140-3664(01)00409-1. URL https://doi.org/10.1016/S0140-3664(01)00409-1. 6, 34

[29] Jagruti Sahoo, Mohammad A. Salahuddin, Roch H. Glitho, Halima Elbiaze, and Wessam Ajib. A survey on replica server placement algorithms for content delivery networks. *IEEE Communications Surveys and Tutorials*, 19(2): 1002–1026, 2017. doi: 10.1109/COMST.2016.2626384. URL https://doi.org/10.1109/COMST.2016.2626384. 6, 35

[30] Fan Ping, Xiaohu Li, Christopher McConnell, Rohini Vabbalareddy, and Jeong-Hyon Hwang. Towards optimal data replication across data centers. In *31st IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2011 Workshops), 20-24 June 2011, Minneapolis, Minnesota, USA*, pages 66–71, 2011. doi: 10.1109/ICDCSW.2011.49. URL https://doi.org/10.1109/ICDCSW.2011.49. 6, 35

[31] IBM. Unix domain sockets. www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.14/gtpc1/unixsock.html. [Online; accessed 01-February-2018]. 11

[32] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, 1978. doi: 10.1145/359576.359585. URL http://doi.acm.org/10.1145/359576.359585. 11

[33] Russell R. Atkinson and Carl Hewitt. Parallelism and synchronization in actor systems. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, USA*, pages 267–280. ACM, 1977. 11

[34] Brandon Barker. Message Passing Interface (MPI). In *Workshop: High Performance Computing on Stampede*, 2015. 11

[35] Google LLC. The go programming language specification, 2018. URL `https://golang.org/ref/spec`. [Online; accessed 01-February-2018]. 11

[36] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984. doi: 10.1145/2080.357392. URL `http://doi.acm.org/10.1145/2080.357392`. 11

[37] Esmond Pitt and Kathy McNiff. *java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., USA, 2001. ISBN 0201700433. 11

[38] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: a language for distributed programming. *SIGPLAN Notices*, 25(5):17–24, 1990. doi: 10.1145/382080.382082. URL `http://doi.acm.org/10.1145/382080.382082`. 12

[39] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996.*, pages 174–185, 1996. doi: 10.1145/237090.237179. 12

[40] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. 12

[41] Jarek Nieplocha and Robert J. Harrison. Shared memory programming in metacomputing environments: The global array approach. *The Journal of Supercomputing*, 11(2):119–136, 1997. doi: 10.1023/A:1007955822788. URL `https://doi.org/10.1023/A:1007955822788`. 12

[42] Kei Simon Pedersen and Brian Vinter. Java pastset: a structured distributed shared memory system. *IEE Proceedings - Software*, 150(2):147–153, 2003. doi: 10.1049/ip-sen:20030135. URL `https://doi.org/10.1049/ip-sen:20030135`. 12

[43] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992. doi: 10.1145/129630.376083. URL `http://doi.acm.org/10.1145/129630.376083`. 12

[44] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2): 114–131, 2003. doi: 10.1145/857076.857078. URL `http://doi.acm.org/10.1145/857076.857078`. 12

[45] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Comput. Surv.*, 21(3):323–357, 1989. doi: 10.1145/72551.72553. URL http://doi.acm.org/10.1145/72551.72553. 13, 15

[46] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999. doi: 10.1023/A:1010060322135. URL https://doi.org/10.1023/A:1010060322135. 13

[47] Matteo Ceriotti, Amy L. Murphy, and Gian Pietro Picco. Data sharing vs. message passing: synergy or incompatibility?: an implementation-driven case study. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 100–107, 2008. doi: 10.1145/1363686.1363714. URL http://doi.acm.org/10.1145/1363686.1363714. 13

[48] Ronaldo Menezes and Robert Tolksdorf. A new approach to scalable linda-systems based on swarms. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA*, pages 375–379, 2003. doi: 10.1145/952532.952607. URL http://doi.acm.org/10.1145/952532.952607. 13

[49] Robert Tolksdorf and Ronaldo Menezes. Using swarm intelligence in linda systems. In *Engineering Societies in the Agents World IV, 4th International Workshop, ESAW 2003, London, UK, October 29-31, 2003, Revised Selected and Invited Papers*, pages 49–65, 2003. doi: 10.1007/978-3-540-25946-6_3. URL https://doi.org/10.1007/978-3-540-25946-6_3. 15

[50] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms. 17, 19

[51] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998. doi: 10.1109/32.685256. URL http://doi.ieeecomputersociety.org/10.1109/32.685256. 17, 25, 26, 37, 38

[52] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009. ISBN 978-0-13-187325-4. 19

[53] Dinesh P. Mehta and Sartaj Sahni, editors. *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. ISBN 978-1-58488-435-4. doi: 10.1201/9781420035179. URL https://doi.org/10.1201/9781420035179. 21

[54] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21, 1978. doi: 10.1109/SFCS.1978.3. URL `https://doi.org/10.1109/SFCS.1978.3`. 21

[55] Sabine Hanke. The performance of concurrent red-black tree algorithms. In *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19-21, 1999, Proceedings*, pages 287–301, 1999. doi: 10.1007/3-540-48318-7_23. URL `https://doi.org/10.1007/3-540-48318-7_23`. 21

[56] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. Lh* - A scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4):480–525, 1996. doi: 10.1145/236711.236713. URL `http://doi.acm.org/10.1145/236711.236713`. 22

[57] Sandford Bessler, Alexander Fischer, eva Kühn, Richard Mordinyi, and Slobodanka Tomic. Using tuple-spaces to manage the storage and dissemination of spatial-temporal content. *J. Comput. Syst. Sci.*, 77(2):322–331, 2011. doi: 10.1016/j.jcss.2010.01.010. URL `https://doi.org/10.1016/j.jcss.2010.01.010`. 22

[58] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: fast *and* general?(!). In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 34:1–34:2, 2016. doi: 10.1145/2851141.2851188. URL `http://doi.acm.org/10.1145/2851141.2851188`. 23

[59] Chuck Pheatt. Intel threading building blocks. *J. Comput. Small Coll.*, 2008. 23

[60] FOLLY. Folly: Facebook open-source library. `www.github.com/facebook/folly`. [Online; accessed 01-February-2018]. 23

[61] C Click. A lock-free wait-free hash table, 2007. URL `http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`. [Online; accessed 01-February-2018]. 23

[62] Christine Julien and Gruia-Catalin Roman. Active coordination in ad hoc networks. In Rocco De Nicola, Gian Luigi Ferrari, and Greg Meredith, editors, *Coordination Models and Languages, COORDINATION 2004*, volume 2949 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2004. ISBN 3-540-21044-X. 25

[63] Andrea Omicini and Franco Zambonelli. TuCSoN: a coordination model for mobile information agents. In David G. Schwartz, Monica Divitini, and Terje Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9 June 1998. IDI – NTNU, Trondheim (Norway). 25

[64] Sun Microsystems. JS - JavaSpaces Service Specification. https://river.apache.org/doc/specs/html/js-spec.html. [Online; accessed 15-September-2016]. 25, 27

[65] T.J. Lehman, S.W. McLaughry, and P. Wyckoff. Tspaces: The next wave. In *HICSS 1999: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences*, volume 8. Springer Berlin Heidelberg, 1999. 25, 26

[66] S. Capizzi. *A Tuple Space Implementation for Large-Scale Infrastructures*. Department of Computer Science Univ. Bologna thesis. University of Bologna, 2008. 25, 29, 50

[67] Lorenzo Bettini, Rocco De Nicola, and Michele Loreti. Implementing mobile and distributed applications in X-Klaim. *Scalable Computing: Practice and Experience*, 7(4), 2006. 26

[68] Richard Mordinyi, eva Kühn, and Alexander Schatten. Space-based architectures as abstraction layer for distributed business applications. In *CISIS 2010, The Fourth International Conference on Complex, Intelligent and Software Intensive Systems*, pages 47–53. IEEE Computer Society, 2010. 28

[69] Stefan Craß, Tobias Dönz, Gerson Joskowicz, eva Kühn, and Alexander Marek. Securing a space-based service architecture with coordination-driven access control. *JoWUA*, 4(1):76–97, 2013. URL http://isyou.info/jowua/papers/jowua-v4n1-4.pdf. 28

[70] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. An efficient byzantine-resilient tuple space. *IEEE Trans. Computers*, 58 (8):1080–1094, 2009. doi: 10.1109/TC.2009.71. URL http://dx.doi.org/10.1109/TC.2009.71. 31

[71] Tobias Distler, Christopher Bahn, Alysson Neves Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 10:1–10:16, 2015. doi: 10.1145/2741948.2741954. URL http://doi.acm.org/10.1145/2741948.2741954. 32

[72] Giovanni Russello, Michel R. V. Chaudron, Maarten van Steen, and Ibrahim Bokharouss. An experimental evaluation of self-managing availability in shared data spaces. *Sci. Comput. Program.*, 64(2):246–262, 2007. 32

[73] Giovanni Russello, Michel R. V. Chaudron, and Maarten van Steen. Exploiting differentiated tuple distribution in shared data spaces. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*, pages 579–586, 2004.

doi: 10.1007/978-3-540-27866-5_76. URL https://doi.org/10.1007/978-3-540-27866-5_76. 32

[74] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001*, pages 524–533, 2001. doi: 10.1109/ICDSC.2001.918983. URL https://doi.org/10.1109/ICDSC.2001.918983. 32

[75] Amy L. Murphy and Gian Pietro Picco. Using lime to support replication for availability in mobile ad hoc networks. In *Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, pages 194–211, 2006. doi: 10.1007/11767954_13. URL https://doi.org/10.1007/11767954_13. 32

[76] Marina Andric, Rocco De Nicola, and Alberto Lluch-Lafuente. Replica-based high-performance tuple space computing. In Tom Holvoet and Mirko Viroli, editors, *COORDINATION*, volume 9037 of *LNCS*, pages 3–18. Springer, 2015. 33, 86

[77] L.C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40:35âĂŞ41, 1977. 35, 94

[78] M.A. Beauchamp. An improved index of centrality. *Behavioral Science*, 10: 161âĂŞ163, 1965. 35, 94

[79] Samee Ullah Khan, Anthony A. Maciejewski, and Howard Jay Siegel. Robust CDN replica placement techniques. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–8, 2009. doi: 10.1109/IPDPS.2009.5160908. URL https://doi.org/10.1109/IPDPS.2009.5160908. 35

[80] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN 978-0-13-115007-2. 37

[81] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a java package for distributed and mobile applications. *Softw., Pract. Exper.*, 32 (14):1365–1394, 2002. doi: 10.1002/spe.486. URL https://doi.org/10.1002/spe.486. 40

[82] Oracle. Package java.nio. https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html, . [Online; accessed 1-December-2017]. 41, 48

[83] Andrew S. Tanenbaum. *Computer networks, 4th Edition*. Prentice Hall, 2002. ISBN 978-0-13-038488-1. 42

[84] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Trans. Software Eng.*, 24(5):342–361, 1998. doi: 10.1109/32.685258. URL `https://doi.org/10.1109/32.685258`. 45

[85] Lorenzo Bettini, Rocco De Nicola, Daniele Falassi, Marc Lacoste, Luís M. B. Lopes, Licínio Oliveira, Hervé Paulino, and Vasco Thudichum Vasconcelos. A software framework for rapid prototyping of run-time systems for mobile calculi. In *Global Computing, IST/FET International Workshop, GC 2004, Rovereto, Italy, March 9-12, 2004, Revised Selected Papers*, pages 179–207, 2004. doi: 10.1007/978-3-540-31794-4_10. URL `https://doi.org/10.1007/978-3-540-31794-4_10`. 45

[86] Oracle. ConcurrentHashMap. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html`, . [Online; accessed 1-December-2017]. 50

[87] Christine Julien and Gruia-Catalin Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Trans. Software Eng.*, 32(5):281–298, 2006. doi: 10.1109/TSE.2006.47. URL `https://doi.org/10.1109/TSE.2006.47`. 50

[88] Elena Nardini, Mirko Viroli, and Emanuele Panzavolta. Coordination in open and dynamic environments with tucson semantic tuple centres. In *SAC 2010*, volume III, pages 2037–2044. ACM, 2010. ISBN 978-1-60558-638-0. doi: 10.1145/1774088.1774515. URL `http://portal.acm.org/citation.cfm?id=1774515`. 50

[89] Nicholas Carriero and David Gelernter. *How to write parallel programs - a first course*. MIT Press, 1990. 53

[90] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Programming*. The software patterns series. Addison-Wesley, 2005. ISBN 9780321228116. 53

[91] Michael Jay Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, 2004. 60

[92] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. In *From Active Data Management to Event-Based Systems and More - Papers in Honor of Alejandro Buchmann on the Occasion of His 60th Birthday*, pages 242–259, 2010. doi: 10.1007/978-3-642-17226-7_15. URL `https://doi.org/10.1007/978-3-642-17226-7_15`. 83

[93] Béla Bollobás, Christian Borgs, Jennifer T. Chayes, and Oliver Riordan. Directed scale-free graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, USA*, pages 132–139, 2003. 98

[94] P. Erdös and A. Rényi. On random graphs. In *Publicationes Mathematicae Debrecen*, volume 6, pages 290–297, 1959. 99