Tuple spaces implementations and their efficiency

Vitaly Buravlev, Rocco De Nicola, and Claudio Antares Mezzina

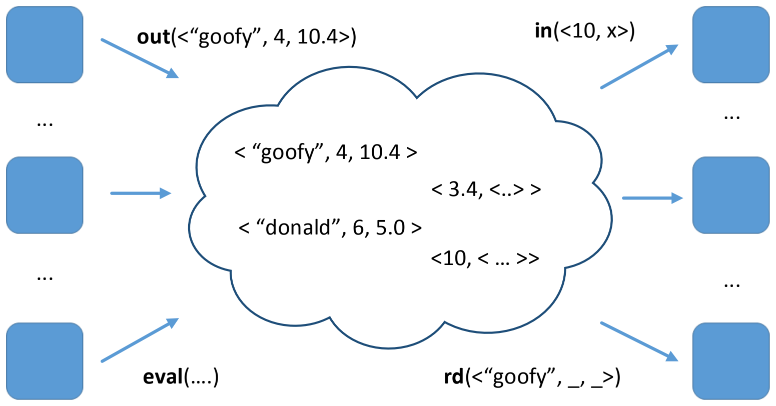
IMT School for Advanced Studies Lucca,
Piazza S. Francesco, 19, 55100 Lucca, Italy
{vitaly.buravlev, rocco.denicola, claudio.mezzina}@imtlucca.it

Abstract. Among the paradigms for parallel and distributed computing, the one popularized with Linda and based on tuple spaces is the least used one, despite the fact of being intuitive, easy to understand and to use. A tuple space is a repository of tuples, where process can add, withdraw or read tuples by means of atomic operations. Tuples may contain different values, and processes can inspect the content of a tuple via pattern matching. The lack of a reference implementations for this paradigm has prevented its widespread. In this paper, first we do an extensive analysis on what are the state of the art implementations and summarise their characteristics. Then we select three implementations of the tuple space paradigm and compare their performances on three different case studies that aim at stressing different aspects of computing such as communication, data manipulation, and cpu usage. After reasoning on strengths and weaknesses of the three implementations, we conclude with some recommendations for future work towards building an effective implementation of the tuple space paradigm.

1 Introduction

Distributed computing is getting increasingly pervasive, with demands from various applications domains and highly diverse underlying architectures from the multitude of tiny things to the very large cloud-based systems. Several paradigms for programming parallel and distributed computing have been proposed so far. Among them we can list: distributed shared memory, message passing, actors, distributed objects and tuple spaces. Nowadays, the most used paradigm seems to be message passing, with MPI [2] being its latest incarnation, while the least popular one seems to be the one based on tuple space that was proposed by David Gelernter for the Linda coordination model [8].

As the name suggests, message passing provides coordination abstractions based on the exchange of messages between distributed processes, where message delivery is often mediated via brokers and messages consist of a header and a body. In its simplest incarnation, message-passing provides a rather low-level programming abstraction for building distributed systems. Linda, instead provides a higher level of abstraction by defining operations for synchronization



and exchange of values between different programs that can share information by accessing common repositories named *tuple spaces*.

The key ingredients of Linda are few basic operations which can be embedded into different programming languages. These are atomic operations used for writing (`out`), withdrawing (`in`), reading (`rd`) tuples into/from a tuple space. The operations for reading and withdrawing select tuples via *pattern-matching*. Another operation `eval` is used to spawn new processes. The figure above illustrates an example of tuples space with different, structured, values. For example tuple $\langle$"goofy"$, 4, 10.4\rangle$ is produced by a process via the `out`$(\langle$"goofy"$, 4, 10.4\rangle)$ operation, and it is read by the operation `rd`("goofy"$, \_, \_)$ after pattern-matching: that is the process reads any tuple of three elements whose first one is exactly the string "goofy". Moreover, tuple $\langle 10, \langle \ldots \rangle\rangle$ is consumed (atomically retracted) by operation `in`$(10, x)$ which consumes a tuple whose first element is 10 and binds its second element (whatever it is) to the variable x. Patterns are sometimes referred as *templates*.

The simplicity of this coordination model makes it very intuitive and easy to use. Some synchronization primitives, e.g. semaphores, barrier synchronization, can be implemented easily in Linda (cf. [6], Chapter 3). Unfortunately Linda's implementations of tuple space have turned out to be quite inefficient, and this has led researchers to opt for different approaches such Open MP or MPI, which are nowadays offered, as libraries, for many programming languages. When considering distributed applications, the limited use of Linda coordination model is also due to the need of keeping tuple spaces consistent. In fact, in this case, control mechanisms that can affect scalability are needed [7].

In our view, tuple spaces can be effectively exploited as a basis for the broad range of the distributed applications with different domains (from lightweight applications to large cloud based systems). However, in order to be effective, we need to take into account that performances of a tuple space system may vary depending on the system architecture and the type of interaction between its components. The aim of this paper is to examine the state of the art implementations of tuple spaces, and to find out strengths and weaknesses.

We start by cataloguing the existing implementations according to their features, and then we focus on the most recent Linda based systems that are still maintained, while paying specific attention to those featuring decentralized tuples space. For the selected systems, we compare their performances on three different case studies that aim at stressing different aspects of computing such as communication, data manipulation, and cpu usage. After reasoning on strength and weakness of the three implementations, we conclude with some recommendation for future work towards building effective implementation of the tuple space paradigm.

2 Tuple space systems

In this Section, first we review several existing tuple space systems by briefly describing each of them, and single out the main features of their implementations, then we summarise these features in Table 1. Later, we focus on the implementations that enjoy the characteristics we consider important for a tuple space implementation: code mobility, distribution of tuples and flexible tuples manipulation.

JavaSpaces. JavaSpaces [13] is one of the first implementations of the tuple space developed by Sun Microsystems. It is based on a number of Java technologies (Jini, RMI). As a commercial system, JavaSpaces supports transactions and mechanism of tuple leases. A tuple, called entry in JavaSpaces, is an instance of a Java class and its fields are the public fields of the class. This means that tuples are restricted to contain only objects but 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.

TSpaces. TSpaces [12] is an implementation of the Linda model at the IBM Almaden Research Center. It combines asynchronous messaging with database features. Like JavaSpaces, TSpaces provides transactional support and mechanism of tuple leases. 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. Patter matching uses SQL-like queries, allowing to match tuples regardless of their structure (e.g. the order in which fields are stored).

GigaSpaces. GigaSpaces [9] is a contemporary commercial implementation of tuple space. Nowadays, the core of that system is GigaSpaces XAP, a scale-out application server and any user application should interact with it for creating and manipulating its own tuple space. The main areas where GigaSpaces can be applied are concerned with big data analythics. GigaSpaces main features are: linear scalability, optimization of RAM usage, synchronization with databases and several database-like operations such as complex queries, transactions and replication.

Tupleware. Tupleware [1] 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 performance on a computing cluster and provides clear and simple programming facilities for dealing with distributed tuple space as well as with centralized one. The tuple space is implemented as a hashtable, containing pairs consisting of a key and a vector of tuples. Due to the nature of Jave hashtable, it is possible to access concurrently several elements of the hashtable, since synchonisation is at the level of hashtable element. To speed up the search in the distributed tuple space, an algorithm based on the history of communication is used. Its main aim is to minimize the number of communications between nodes 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 past attempts to get information from other nodes and tuples are first searched in nodes with greater success factor.

Grinda. Grinda [5] is a distributed tuple space which was designed for large scale infrastructures. It combines Linda coordination model with grid architecture aiming at improving performance of distributed tuple space, especially with a large amount 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 network (based on Content Addressable Network and tree based).

Blossom. Blossom [15] is a C++ implementation of Linda which was developed to achieve high performance and correctness of the programs using Linda model. In Blossom all tuple spaces are homogeneous with predefined structure, and this allows spending less time for type comparison during the search. To improve scalability, Blossom uses distributed tuple spaces and each processor is assigned a particular tuple space by considering tuple values. The technique of prefetching allows a process to send a request for some tuples to the tuple space and to continue its work while the search continues. When the process needs the requested tuples, it receives them without waiting and spending time for their search which have been already done.

DTuples. DTuples [10] is designed for peer-to-peer networks and based on distributed hash table (DHT), a scalable and efficient approach. Key points of DHT are autonomy and decentralization. There is no central server and each node of DHT is in charge of storing a part of hash table and of keeping routing information about other nodes. As the basis of the DTH's implementation DTuples uses FreePastry[1]. DTuples also supports transactions and guarantees fault-tolerance via replication mechanisms. DTuples supports multi tuple spaces and distinguishes *public* and *subject* tuple spaces. Public tuple space is a space shared among all the processes and all of them can perform any operation on it. Subject tuple space is a private space accessible only by the processes that are 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 two types of tuple spaces, pattern matching is specific for each of them. Templates in the subject tuple space can match tuples in the same subject tuple space and in the common tuple space. However, the templates in the common tuple space cannot match the tuple in the subject tuple spaces.

LuaTS. LuaTS [11] is a reactive event-driven tuple space system written in Lua. Its main features are associative mechanism of tuple retrieving, fully asynchronous operations and support of code mobility. LuaTS provides centralized management of the tuple space which can be logically partitioned into several parts using indexing. LuaTS combines Linda model with event-driven programming paradigm. This paradigm was chosen to simplify program development which allows avoiding the use of synchronization mechanisms for tuple retrieval and makes more transparent programming and debugging of multi-thread program. Tuples can contain any data which can be serialized in Lua, including strings with function code. In order to obtain a more flexible and intelligent search, function code can be sent to the server and once executed it can returns the matched tuples. Reactive tuple space is implemented as a hashtable, in which along with data also information supporting the reactive nature of that tuple space (templates, client addresses, ids of callback and so on) is stored.

Klaim. Klaim [3] (the Kernel Language for Agents Interaction and Mobility) is an extension of Linda supporting processes migration. The emphasis of Klaim is on process mobility, which means that processes as any data can be moved from one locality to another and they can be executed in any localities. Klava is a Java implementation of Klaim [4]. Klaim supports multiple tuple spaces and operates 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 parametric to localities. Emphasis is put also on access control which is important for mobile applications. For this reason Klaim introduces type system which allows checking whether a process can perform an operation at specific localities.

In order to compare the implementations we have discussed so far, we have singled out the following criteria:

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

	JSP	**TSP**	**GSP**	**TW**	**GR**	**BL**	**DTP**	**LTS**	**KL**
Distributed tuple space			?	✓	✓	✓	✓		✓
Decentralized management			?	✓	✓		✓		
Tuple clustering			?	✓				✓	✓
Domain specificity				✓				✓	
Scalability			✓	✓	✓		✓		
Security		✓	✓						✓
`eval` operation								✓	✓

JavaSpaces (**JSP**), TSpaces (**TSP**), GigaSpaces (**GSP**), Tupleware (**TW**), Grinda (**GR**), Blossom (**BL**), DTuples (**DTP**), LuaTS (**LTS**), Klaim (**KL**)

Table 1: Results of the comparison

Distributed tuple space This criterion denotes whether tuple spaces are stored in one single node of the distributed 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.

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.

Domain specificity Many of implementations have specific area in which they can be used. If the implementation is domain specific it can be good because it is more suitable for it and has an advantage over other ones. On another side, this feature could be considered a limitation if one aims at generality.

Scalability This criterion implies that system based on particular Linda implementation can cope with the increasing amount of data and nodes while maintaining acceptable performance.

Security This criterion specifies whether an implementation has security features or not.

`eval` **operation** This criterion denotes whether the tuple space system has implemented the `eval` operation.

Table 1 summarises the result of our comparison: ✓ means that the implementation enjoys the property and ? means that we were not able to provide an answer, since the source code was not available.

An extra requirement to be able to compare implementations (especially in terms of time) is that they have to be written in the same language. We have chosen Java, since nowadays it is the most used language. Moreover, using a single programming language allows us to develop case studies as *skeletons*: the code remains the same for all the implementations, only the invocations of different library methods do change. This choice, guarantees also the possibility of performing better comparisons of the time performances exhibited by the different tuple systems which could be significantly depend the chosen language.

After considering the results in Table 1, to perform our detailed experiments we have chosen: TUPLEWARE which enjoys most of the selected features; KLAIM since it offers distribution, clustering of tuple spaces and code mobility. Finally, we have chosen GIGASPACES because it is the most modern among the commercial systems; it will be used as a yardstick to compare the performance of TUPLEWARE and KLAIM. 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.

In all our implementations of the case studies, we have structured the systems by assigning each process a local tuple space. Because GIGASPACES is a centralized tuple space, in order to satisfy this rule we do not use it as centralized one, but as distributed: each process is assigned its own tuple space in the GIGASPACES server.

3 Experiments

3.1 Case studies

In order to compare different tuple space systems we have chosen 3 case studies: *Password search*, *Sorting* and *Ocean model*. The first case study is a communication intensive task where the number of tuples is large and it requires doing many reading and writing operations. The second case study is computation intensive, since each node spends more time for sorting elements than for communicating with the other nodes. This case study has been considered because it needs structured tuples that contains both basic values (with primitive type) and complex data structures that impact on the speed of the inter-process communication. The third case has been taken into account since it introduces particular dependencies among nodes, which if exploited can improve the application performances. This was considered to check whether adapting a tuple space system to the specific inter-process interaction pattern of a specific class of the applications could lead to significative performance improvements. All the case studies are implemented using master-worker paradigm [6]. Now we briefly describe them.

Password search. The main aim of the distributed application for password search is to find a password using its hashed value in the predefined distributed database. We have generated that database in the form of the files containing pairs of password and hashed value, for each password. The application creates a master process and several worker processes: the master keeps asking to the workers passwords corresponding to a specific hashed values, by issuing tuples of the form:

$$\langle \text{“search\_task”}, dd157c03313e452ae4a7a5b72407b3a9 \rangle$$

Each worker first loads its part of the distributed database, and after, it 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 process, with a tuple of the form:

$$\langle \text{“found\_password”}, dd157c03313e452ae4a7a5b72407b3a9, 7723567 \rangle$$

For multi tuple spaces 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 results.

Sorting. This distributed application consists of sorting arrays of integers. The master is responsible for loading initial data and for collecting the final sorted data, while workers are directly responsible for the sorting. At the beginning, the master loads predefined initial data to be sorted and sends them to one worker to start the sorting process. Afterwards, the master waits for the sorted arrays from the workers: when any sub-array is sorted the master receives it and builds the whole sorted sequence when all sub-arrays are collected. The behavior of workers is different; when they are instantiated, each of them starts searching for the unsorted data in local and remote tuple spaces. When a worker finds a tuple with data, it checks whether it is possible to sort these data (the size of the data is less than particular threshold). If it is possible to sort them, the worker does the computation, sends the result to the master and starts searching for other unsorted data. Otherwise, the worker splits the array into two parts: one part is stored into its local tuple space while the other is processed.

Ocean model. The ocean model is a simulation of the enclosed body of water. The core of that case study was given in [1]. The two-dimensional surface of water in the model is represented as a 2-D grid and each cell of the grid represents one point. 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, and each worker owns one panel in order to compute its parameters. The aim of the case study is to simulate the body of water during several time-steps. At each time-step, in order to compute the new panel parameters, each worker has to take into account its neighbouring panels.

The mission of the master and workers are similar to the previous case studies. In the application the master instantiates the whole grids, divides it into parts and sends them to the workers. After all iterations, it receives all parts of the grid. Each worker receives its share 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; in the end it sends its data to the master.

Implementing case studies. Since we have chosen Java-based tuple space systems, all case studies are implemented in Java. Implementations of the three case studies require the use of synchronization to avoid conflicts while accessing to the same tuple space. GigaSpaces and Tupleware have built in synchronization mechanisms, while Klaim does not. To cope with it, for Klaim we implemented synchronizations, using standard Java synchronized blocks [14], at the node/process level instead of modifying the source code of the core operation and applied it to local tuple space.

There is a difference in the implementation of the search among distributed tuple spaces. Tupleware has a built in operation with notification mechanism: it searches in local and remote tuple spaces once and then waits for the notification that the wanted tuple appears in one of the tuple spaces. The implementation of this operation for Klaim and GigaSpaces requires to continuously check each tuple space until the wanted tuple is found.

3.2 Methodology

All the conducted experiments are parametric with respect to two parameters. The first one is the number of workers taken into account with values $1, 5, 10, 15$ and it tests how the different implementations scale up with concurrency. The second parameter is application specific, but its meaning is the same: testing the implementation when the workload increases. For the case study *Password search* we vary the number of the entries in the database (10000, 1000000 and 1 million passwords) where it is necessary to search the password. This parameter directly affects the number of local entries each worker has. Moreover, for this case study the number of password to find was fixed to 100. For the *Sorting* case, the second parameter is the number of elements in an array to be sorted (100000, 1 million, 10 million elements). In this case the number of elements does not correspond to the number of tuples because parts of array are transferred also as arrays of smaller size. For the case study *Ocean model* the second parameter is the grid size (300, 600 and 1200) which is related with computational size of the initial task.

Remark 1 (Execution environment). Our test were conducted on a server with 4 processors Intel Xeon E5620 (4 cores, 12M Cache, 2.40GHz, Hyper-Threading Technology) with 32 threads in total, 40 GB RAM and installed Ubuntu 14.04.3. All applications are programmed in Java 8 (1.8.0).

Measured metrics. For measurement of metrics Clarkware Profiler[2] is used. We use manual method of profiling and insert methods `Profiler.begin(label)` and `Profiler.end(label)` surrounding parts of the code we are interested into program code in order to begin and stop counting time respectively. This sequence of the actions can be repeated many times and in the end we receive report which includes the number of calls, overall and average time. For each metrics the label is different and it is possible to use several of them simultaneously. Each set of experiments was conducted 10 times with randomly generated input and average values of each metrics were computed. To extensively compare the different implementations, we have chosen the following measures:

Local writing time: required time to write one tuple into local tuple space.
Local reading time: required time to read or take one tuple from local tuple space using template. The parameter checks how fast pattern matching works.
Remote writing time: time of the writing to the tuple space plus the time of communication with process associated with tuple space.
Remote reading time: similarly to the previous one, this time is a sum of the time of the search in tuple space and the time of the communication with it.
Search time: when the application has several workers we introduce the time which is required to find a tuple in a several separated tuple spaces.
Total time: total execution time. This time does not include initial creation of tuple spaces or starting tuple space server as in the case of GigaSpaces.
Number of visited nodes: number of visited before a necessary tuple was found.

Please notice that, all plots used in the paper report results of our experiments in a logarithmic scale. When describing the outcome, we have only used those plots which are more relevant to evidence the difference between the three tuple space systems[3]

3.3 Results

Password search As shown in figure 1 GigaSpaces exhibits better performances than the other two tuple space systems.

Figure 2 depicts the local writing time for each implementation, with different numbers of workers. As we can see, by increasing the number of workers (which implies reducing the amount of local data to consider), the local writing time decreases. This is more evident in Tupleware, which really suffers when a big number of tuples (e.g. 1 million) is stored in a single local tuple space. The writing time of Klaim is the lowest among other systems and does not change significantly during any variation in the experiments.

[2] The profiler was written by Mike Clark; source code is available in GitHub.com: `https://github.com/akatkinson/Tupleware/tree/master/src/com/clarkware/profiler`.
[3] Plots with more detailed (numeric) information are reported as bar charts at `http://sysma.imtlucca.it/coord16_appendix/`.

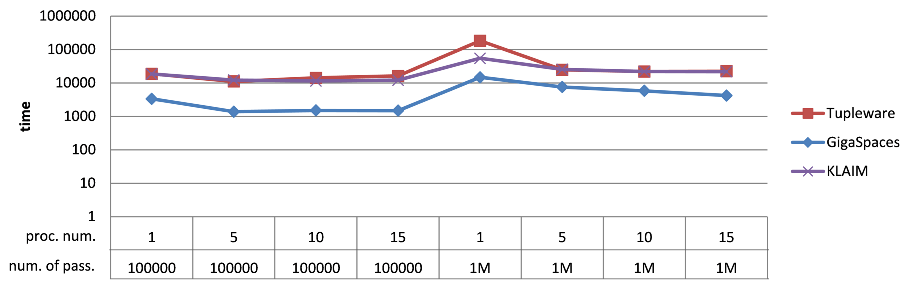


Fig. 1: Password search. Local total time

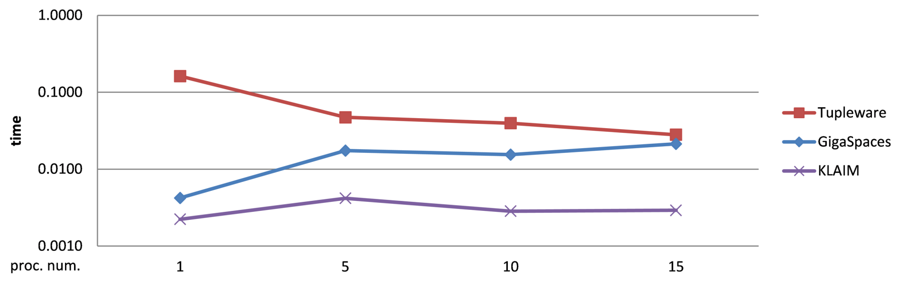


Fig. 2: Password search. Local writing time (1 million passwords)

Local reading time is shown in Figure 3 and KLAIM is the one that exhibits the worst performance for searching in local space. Indeed, if there is just one worker, the local reading time is 10 times greater than TUPLEWARE. We conjecture that the pattern matching mechanism of KLAIM is less effective than others. By increasing the number of workers the difference becomes less evident, even if it remains four times bigger than TUPLEWARE. Since this case study requires little synchronization among workers, performance improves when the level of parallelism (the number of workers) increases.

Search time is similar to local reading time, but takes into account searching in remote tuple spaces. When considering just one worker, the search time is the same as the reading time in local tuple space, however, when the number of workers increases the search time of TUPLEWARE and KLAIM grows faster than the time of GIGASPACES. Figure 4 shows that GIGASPACES is more sensitive to the number of tuples than to the number of accesses to the tuple space.

It is worth to remark that the local tuple spaces of the three systems exhibit different performances depending on the operation on them: the writing time of KLAIM is always significantly smaller than the others, while the pattern matching mechanism of TUPLEWARE allows faster local searching.

Sorting Figure 5 shows that GIGASPACES exhibits significantly better execution time when the number of elements to sort is 1 million. When 10 million elements

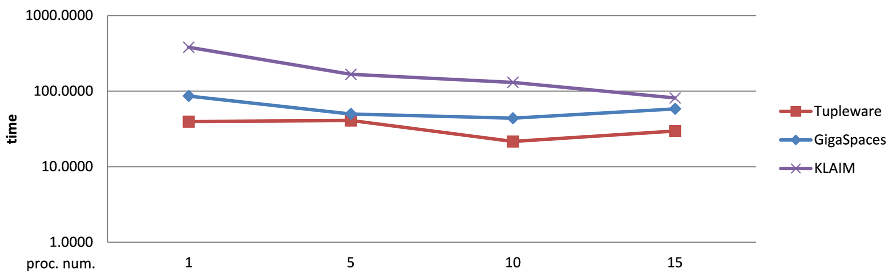


Fig. 3: Password search. Local reading time (1 million passwords)

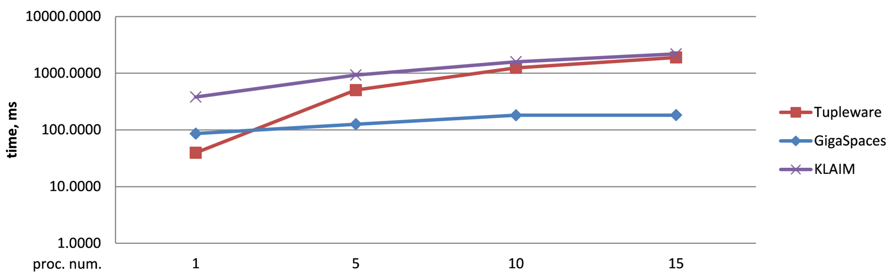


Fig. 4: Password search. Search time (1 million passwords)

are considered and several workers are involved, Tupleware exhibits a more efficient parallelization and thus requires less time.

This case study is computation intensive but requires also exchange of structured data and, although in the experiments a considerable part of the time is spent for sorting, we have that performances do not significantly improve when the number of workers increases.

The performance of Klaim is visibly worse than others even for one worker. In this case, the profiling of the Klaim application showed that a considerable amount of time was spent to transmit initial data from the master to the worker. Inefficient implementation of data transmission seems to be the reason the total time of Klaim differs from the total time of Tupleware.

By comparing Figures 2 and 6, we see that, when the number of workers increases, GigaSpaces and Klaim suffer more from synchronization in the current case study than in the previous one; there no other operation was performed in parallel to writing and thus no conflict handling was required.

In addition to experimenting with case studies, we measured the time required by reading and writing operations on remote tuple space for all three systems. For Klaim and Tupleware these times were similar and significantly greater than those of GigaSpaces. Klaim and Tupleware communications rely on TCP and to handle any remote tuple space one needs to use exact addresses and ports. GigaSpaces, that has a centralized implementation, most

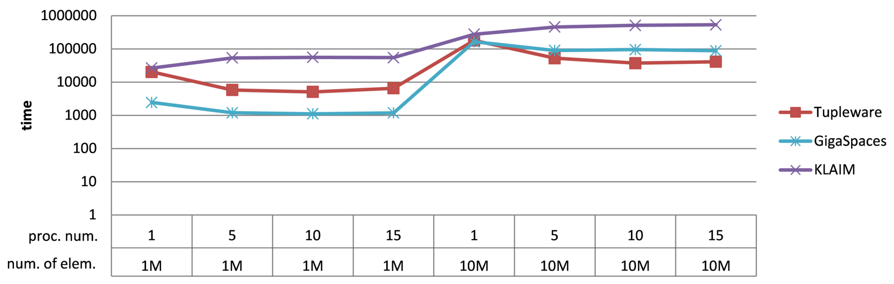


Fig. 5: Sorting. Total time

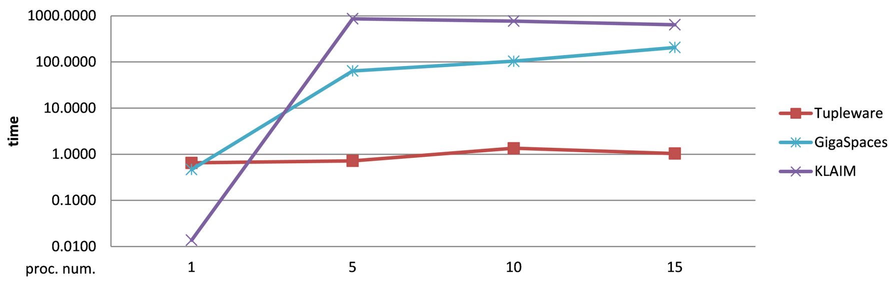


Fig. 6: Sorting. Local writing time (10 million elements)

likely does not use TCP for data exchange but relies on a more efficient memory-based approach.

As shown in Figure 7, search time directly depends on the number of the workers and grows with it. Taking into account that KLAIM and TUPLEWARE spend more time accessing remote tuple space, GIGASPACES suffers more because of synchronization. KLAIM has the same problem, but its inefficiency is hampered by data transmission cost.

Ocean model This case study was chosen to examine behavior of tuple systems when specific patterns of interactions are used. Out of the three considered systems, only TUPLEWARE has a method for reducing the number of visited nodes during search operation which helps in lowering search time. Figure 8 depicts the number of visited nodes for different grid size and different number of workers. The curve depends only weakly on the size of the grid for all systems, and much more on the number of workers. Indeed, from Figure 8 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[4].

[4] Figure 8, the curves for KLAIM and GIGASPACES are overlapping and purple wins over blue.

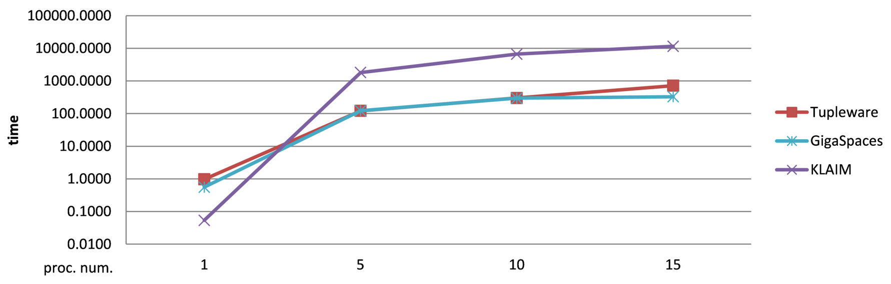


Fig. 7: Sorting. Search time (10 million elements)

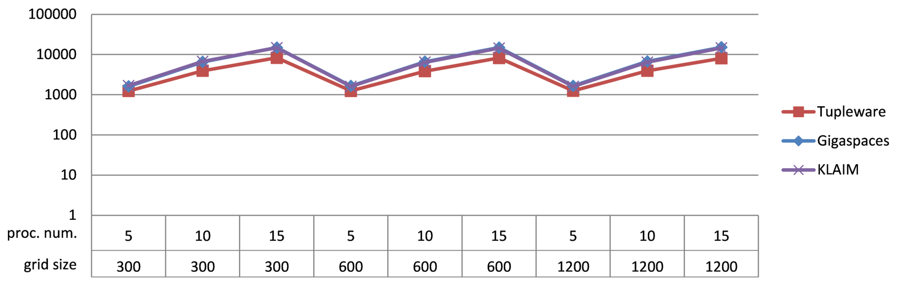


Fig. 8: Ocean model. Number of visited nodes

The difference in the number of visited nodes does not affect significantly the total time of execution (Figure 9) mostly because the case study requires many read operations from remote tuple spaces (Figure 10). But, as it was mentioned before, GigaSpaces implements read operation differently from Tupleware and Klaim and it is more effective when working on a single computer.

Figure 9 provides evidence of the effectiveness of Tupleware when its total execution time is compared with the Klaim one. Indeed, Klaim visits more nodes and spends more time for each read operation, and the difference increases when the grid size grows and more data have to be transmitted.

This case study suggests that devising an appropriate mechanism for taking advantage of the underlying communication pattern can make cooperative work of distributed tuple spaces more effective.

4 Conclusions

Distributed computing is getting increasingly pervasive, with demands from various applications domains and highly diverse underlying architectures from the multitude of tiny things to the very large cloud-based systems. Tuple spaces certainly feature valuable characteristics to help develop scalable distributed applications/systems. This paper has first surveyed and evaluated a number of tuple space systems, then it has analyzed more closely three different systems. We

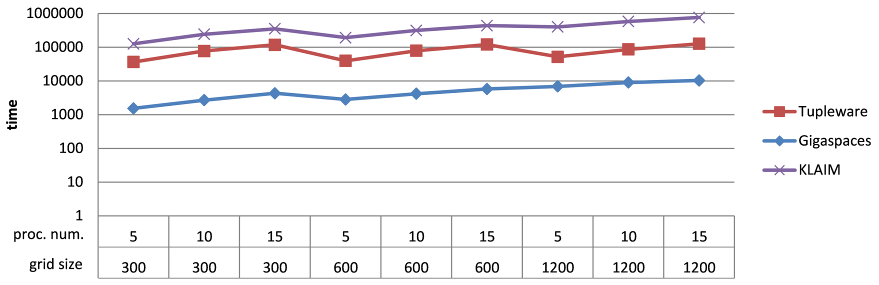


Fig. 9: Ocean model. Total time

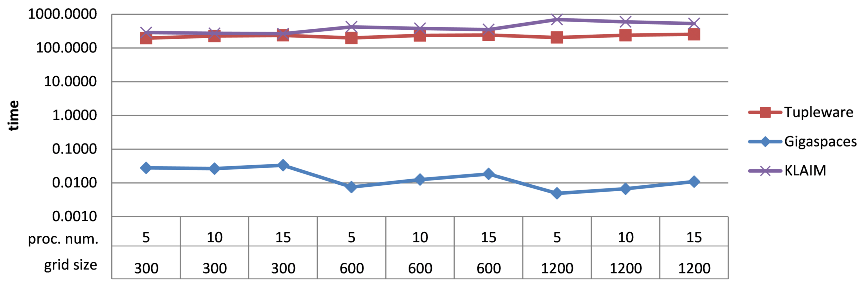


Fig. 10: Ocean model. Remote reading time

considered GigaSpaces, because it is one of the few currently used commercial products, Klaim, because it guarantees code mobility and flexible manipulation of tuple spaces, and Tupleware, because it is the one that turned out to be the best in our initial evaluation. We have then compared the three system by evaluating their performances over three case studies: a communication-intensive one, a computational-intensive one, and one with a specific communication pattern.

Our work follows the lines of [16] but we have chosen more recent implementations and conducted more extensive experiments.

The commercial system GigaSpaces differs from the other two systems for the use of a memory based interprocess communication for data exchange, that guarantees considerably smaller access time to data. Therefore, using this mechanism in the scope of one machine can increase effectiveness of work when different tuple spaces are needed. When working with networked machines, it is not possible to use that mechanism and we need to use approaches to reduce the number of inter-machine communication (e.g. Tupleware approach) and to make that communication effective. Another issue to which we need to pay to attention is related to the implementation of local tuple spaces including pattern matching algorithms and mechanisms to prevent conflicts when accessing the spaces.

Performances of a tuple space system vary depending on the chosen system architectures and on the type of interaction between their components. We did not consider different architectures but we noted problems (data transmission, synchronization, etc.) which may occur in different systems for different types of interaction. We plan to use the results of this work as the basis to design an efficient tuple space system which offers programmer the possibility of selecting (e.g. via a dashboard) the desired features of the tuple space according to the specific application. In this way, one could envisage a distributed middleware with different tuple spaces implementations each of them devised with the best characteristic, in terms of efficiency, to perform the required tasks.

References

1. A. 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.
2. B. Barker. Message Passing Interface (MPI). In *Workshop: High Performance Computing on Stampede*, 2015.
3. L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Rovereto, Italy, February 9-14, 2003, Revised Papers*, pages 88–150, 2003.
4. L. Bettini, R. De Nicola, and M. Loreti. Implementing mobile and distributed applications in X-Klaim. *Scalable Computing: Practice and Experience*, 7(4), 2006.
5. S. Capizzi. *A Tuple Space Implementation for Large-Scale Infrastructures.* Department of Computer Science Univ. Bologna thesis. University of Bologna, 2008.
6. N. Carriero and D. Gelernter. *How to write parallel programs - a first course.* MIT Press, 1990.
7. M. Ceriotti, A. L. Murphy, and G. P. Picco. Data sharing vs. message passing: Synergy or incompatibility? an implementation-driven case study. *Proceedings of the 2008 ACM symposium on Applied computing*, pages 100–107, 2008.
8. D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
9. GigaSpaces. Concepts - XAP 9.0 Documentation - GigaSpaces Documentation Wiki. `http://wiki.gigaspaces.com/wiki/display/XAP9/Concepts`. [Online; accessed 25-January-2016].
10. Y. Jiang, Z. Jia, G. Xue, and J. 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.
11. M. A. Leal, N. Rodriguez, and R. Ierusalimschy. Luats a reactive event-driven tuple space. *Journal of Universal Computer Science*, 9(8):730–744, 2003.
12. T. Lehman, S. McLaughry, and P. Wyckoff. Tspaces: The next wave. 8, 1999.
13. S. Microsystems. JS - JavaSpaces Service Specification. `https://river.apache.org/doc/specs/html/js-spec.html`. [Online; accessed 25-January-2016].
14. Oracle. The Java Tutorials. `https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html`, 2015. [Online; accessed 25-January-2016].
15. R. Van Der Goot. *High Performance Linda Using a Class Library.* PhD thesis. Erasmus Universiteit Rotterdam, 2001.
16. G. Wells, A. Chalmers, and P. G. Clayton. Linda implementations in java for concurrent systems. *Concurrency - Practice and Experience*, 16(10):1005–1022, 2004.