Block Placement Strategies for Fault-Resilient Distributed Tuple Spaces: An Experimental Study

Roberta Barbi, Vitaly Buravlev, Claudio Antares Mezzina, Valerio Schiavoni

Block placement strategies for fault-resilient distributed tuple spaces: an experimental study

(Practical experience report)

Roberta Barbi[1], Vitaly Buravlev[2],
Claudio Antares Mezzina[2], and Valerio Schiavoni[1]

[1] Université de Neuchâtel, Switzerland {roberta.barbi,valerio.schiavoni}@unine.ch
[2] IMT School for Advanced Studies Lucca, Italy
{claudio.mezzina,vitaly.buravlel}@imtlucca.it

Abstract. The tuple space abstraction provides an easy-to-use programming paradigm for distributed applications. Intuitively, it behaves like a distributed shared memory, where applications write and read entries (tuples). When deployed over a wide area network, the tuple space needs to efficiently cope with faults of links and nodes. Erasure coding techniques are increasingly popular to deal with such catastrophic events, in particular due to their storage efficiency with respect to replication. When a client writes a tuple into the system, this is first striped into k blocks and encoded into $n > k$ blocks, in a fault-redundant manner. Then, any k out of the n blocks are sufficient to reconstruct and read the tuple. This paper presents several strategies to place those blocks across the set of nodes of a wide area network, that all together form the tuple space. We present the performance trade-offs of different placement strategies by means of simulations and a Python implementation of a distributed tuple space. Our results reveal important differences in the efficiency of the different strategies, for example in terms of block fetching latency, and that having some knowledge of the underlying network graph topology is highly beneficial.

1 Introduction

We are currently observing a deluge of data originated by our personal devices. Distributed applications must be able to efficiently collect, store, process and expose data. When dealing with such applications, developers need to settle on a specific programming model, to i) facilitate the implementation of such systems and ii) retain user-friendliness and ability to scale, both horizontally and geographically. Distributed storage systems are one prominent example of such applications. They are typically operated across wide area networks, such as Amazon AWS, which currently spans across 15 geographical regions.[3] In such deployment scenarios, applications must transparently tolerate faults, a common threat for distributed systems.

A trivial strategy to tolerate faults is to rely on replication. Block replication obviously entails a huge storage overhead. A state-of-the-art solution to decrease such overhead while providing the same level of fault-tolerance is to use erasure coding

[3] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html

techniques [15]. With a systematic (n,k) linear code, each codeword (an element of the linear code) consists of n blocks: k source blocks for the original data, and $n-k$ redundant blocks. The storage overhead is $\frac{n-k}{k}$, and if the code is Maximum Distance Separable (MDS) [15], any k of the n blocks are necessary and sufficient to recover the original data.

From a fault tolerance point of view, it is optimal to place the n blocks of a codeword on different logical units (with respect to failures), so that the MDS code can tolerate up to $n-k$ failures. A logical unit can be a single node (in this case for the optimum it is sufficient to place different blocks of a codeword on different nodes), but it can also be a cluster of nodes (e.g. a set of machines physically hosted in a single room can go down at the same moment if the cooling system of the room fails). In this second scenario, one is tempted to spread different blocks of a codeword into separate and faraway clusters. Although being optimal with respect to fault tolerance, this solution affects negatively the latency to fetch the required blocks.

The case of distributed tuple spaces. A programming model can be made of two separate pieces: the *computation* model and the *coordination* model. The computation model allows programmer to build a single computational unit, while the coordination model is the glue that binds separate activities into an ensemble [10]. The tuple space paradigm, based on this idea, offers a flexible technique to program parallel and distributed systems, by providing the abstraction of a shared space where all the processes can access. In this model, communication between processes is *indirect* and *anonymous* as it is done through the shared (distributed) space. Moreover, data exists in a tuple space and do not belongs to any process. Despite the simplicity of the model, very few implementations of tuple spaces offer fault tolerant facilities usually in the form of data replication ([16, 4]), with the drawbacks of space overhead and consistency maintenance. In this paper, we consider an extended, distributed tuple space system with erasure-coding capabilities. A tuple to be inserted in the tuple space is *erasure-coded* and its blocks are placed across the nodes joining the tuple space group.

Contributions. First, we study how to distribute the encoded blocks of single codewords over a large-scale network, in order to decrease the fetch latency. We do so by designing and evaluating several different block placement heuristics, over synthetic and real-world network topologies. Second, we evaluate how the proposed heuristics behave with respect to data loss when injecting faults into the topology. Third, we leverage the results of our simulations to identify two suitable placement strategies that we deploy atop a simple distributed tuple space system with the aim of evaluating their performance in a practical setting.

This paper is organized as follows. First, we present the related work (Section 2). Next, Section 3 introduces the tuple space paradigm. In Section 4 we describe the block placement heuristics. Section 5 presents some modeling results that we leverage to drive the prototype implementation. Section 6 presents its implementation details and the extensions done to support both erasure-coding techniques and a pluggable mechanism to choose among the different placement strategies. We present the evaluation of the complete prototype in Section 7. We conclude in Section 8.

2 Related work

The tuple space coordination model is very appealing for distributed systems thanks to its space and time decoupling and its synchronization power. As a consequence, researchers have tried to add fault-tolerance and security to tuple spaces.

One recent result is DEPSPACE [3], a Byzantine fault-tolerant coordination service, which employs process replication for handling crashes and providing fault tolerance.

An alternative to process replication is block replication which entails the problem of block placement.

Block placement policies have been mainly studied in MapReduce contexts such as Hadoop [18]. The main purpose of Hadoop's data placement policy is to provide good balance between reliability, write bandwidth, read performance and load balancing [19]. Placing all replicas on a single node incurs the lowest write bandwidth penalty but it lacks redundancy: if the node fails, data is lost. On the other hand, placing replicas in different data centers maximize redundancy, but at the cost of bandwidth.

Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first, chosen at random. The third replica goes to the same rack as the second, but on a different node. Further replicas are placed on random nodes on the cluster, although Hadoop's block scheduler avoids placing too many replicas on the same rack. Our *cluster-aware* and *distance-aware* strategies share some similarity with this approach, in that they take into account zones of the system that are more sensitive to simultaneous failures. Several enhancement were introduced in Hadoop with respect to block placement policies, such as pluggable policies (since v0.21) or guarantees of even distributions across the cluster (since v0.17). We envision a similar technique to rebalance blocks of the tuple space according to the announced load ratio.

COHADOOP [7] is a lightweight Hadoop extension that gives applications a fine-grain control of data location. Similarly, our scheduling policies allow deployers to choose the destination of the blocks according to different performance criteria.

ADAPT [13] introduced a strategy to mitigate availability heterogeneity issues in non-dedicated distributed computing environments. ADAPT dynamically dispatches data blocks according to hosts' storage capacities. Through simulations, this strategy is shown to reduce the application runtime by more than 30%, increasing data locality and reducing data migration cost, even though the improvement of performances is less significant for environment with higher network connectivity.

3 Tuple Spaces in a Nutshell

The tuple space paradigm, made popular by Linda [9], is an abstraction of shared associative memory for parallel and distributed computing. A tuple space is a repository of tuples that processes can concurrently access via *pattern-matching*. Processes create new tuples (`out` or `write` operation), test the existence of a tuple (`read`) and consume a tuple (via the `in` operation). The simplicity of this coordination model makes this model intuitive and easy to use, also for distributed applications. In fact, some synchronization

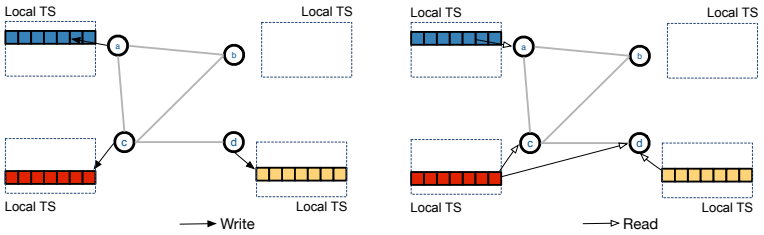


Fig. 1: Example of distributed tuple space: each node writes tuples in its own local tuple space (left) and read tuples from local and remote nodes (right).

primitives (e.g. semaphores, synchronization barriers) can be easily implemented [6] leveraging this coordination model. Tuple space interaction model provides time and space decoupling, in that tuple producers and consumers remain anonymous with respect to each other [8]. Moreover a tuple has to survive its producer's termination, which can be caused by a node crash or due to the ending of the normal execution. In a distributed tuple space, each node writes tuples in its own local space, but it can read tuples also from remote ones. For example, in Figure 1 node **D** reads also the tuple produced by node **C**.

Despite the wide development of tuple space implementations [5], very few of them offer support for distribution. While some systems use replication to guarantee data availability [16] or to be resilient to Byzantine faults [4], no existing system handle link or node faults to guarantee availability of data via erasure-coding. The extensions presented in Section 6 fill this gap.

4 Block Placement Strategies

In this section we describe several heuristics for block placement. Data is stored in the nodes of a graph representing a distributed storage system adding redundancy via standard $[14, 10]$ Reed-Solomon code. The aim of the code is to map 10-blocks-inputs into 14-blocks-codewords in such a way that any 10 encoded blocks are sufficient to recover the original 10. In other words, this linear code can withstand loss of any 4 blocks of a codeword. Then the code provides the same level of fault-tolerance as 5 times replication while entailing a storage overhead of 40% only.

In this configuration, from a fault-tolerance point of view, it is optimal to place the 14 blocks of a codeword on units failing independently, such as geographically remote nodes. In reality, nodes hosted in the same data center have a higher likelihood to fail or being unreachable at the same time. Indeed there are several threats that can lead a data center to a power outage. We can mention cyber attacks, UPS system failures, air conditioner failures or human errors [11].

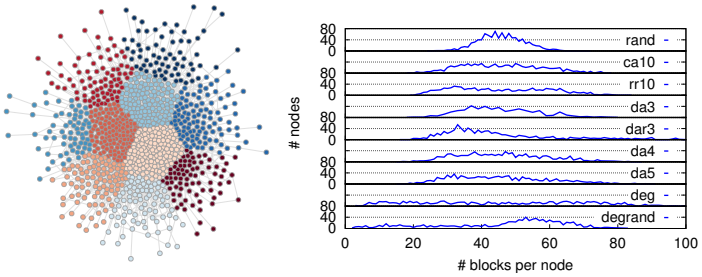


Fig. 2: Left: random graph used in this experiment. Center and right: blocks distribution induced by the placement strategies under study.

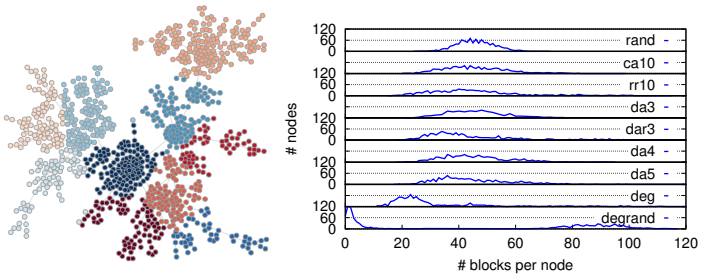


Fig. 3: Left: scale-free graph used in this experiment. Center and right: blocks distribution induced by the placement strategies under study.

The proposed strategies must consider a trade-off between:

- *Latency efficiency*: placing blocks apart from each other negatively affects the fetch latencies;
- *Failure resiliency*: if related blocks are placed geographically close to each other, a failure affecting a wide geographical area will affect several blocks at once.

With the aim of understanding experimentally this trade-off, we study 5 different placement heuristics. They take into account several structural graph properties (e.g. the clustering degree) with the objective of minimizing the latency for fetching blocks.

Round-robin (`rr`). The graph is divided into $\mathcal{K}$ clusters $C_1, \ldots, C_{\mathcal{K}}$ using $\mathcal{K}$-means algorithm [12]. We place the first block in a random node inside cluster C_1, the second block in a random node in cluster C_2 and so on. We proceed until all blocks are placed.

Degree proportional (`deg`). This strategy places more blocks in nodes with higher degree. Intuitively, it let nodes with higher network capacity serve more blocks, irrespectively of their geographical location.

Cluster-aware (`ca`). This strategy assumes knowledge of the clustering of the network and places blocks in the cluster hosting the emitting node and two neighboring clusters. Using $\mathcal{K}$-means, we divide the graph in $\mathcal{K}$ cluster $C_1, \ldots, C_{\mathcal{K}}$. We say that cluster C_i is at distance 1 from cluster C_j if there is an edge of the graph with source/target in C_i and target/source in C_j. For each C_i, we compute all clusters being at distance 1 from C_i.

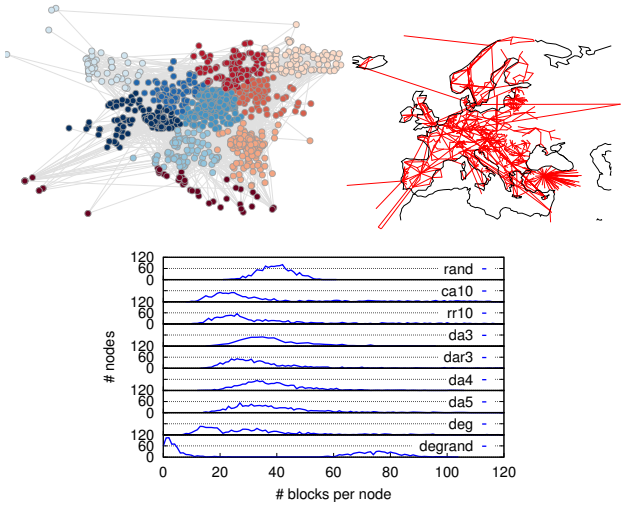


Fig. 4: NREN topology and its blocks distribution.

We say that clusters C_i and C_j are at distance δ from each other if we must cross $\delta - 1$ clusters to go from C_i to C_j and this is the smallest number possible. For each C_i, we compute all clusters being at distance 2 from C_i.

In our simulations, we statically precompute the distances between clusters. We select a first cluster C at random for each codeword. Then, we extract at random 8 nodes from C, 4 different ones from a cluster at distance 1 from C, and finally 2 more from a cluster at distance 2. The chosen nodes receive the 14 blocks of the codeword. Notice that this heuristic needs at least 3 clusters to work.

Distance-aware (`da`). This strategy takes into account the distance between the node emitting the block and the other nodes in the graph. It assumes the knowledge of the diameter of the graph (*d_max*), and proceeds as follows. First, 3 ranges of node-to-node distances (3 being a parameter of the algorithm) must be fixed: *short* (from the minimum to the 33^{rd} percentile of *d_max*), *mid* (from the 33^{rd} to the 66^{th} percentile of *d_max*), and *long* from the 66^{th} to *d_max*. Then, for each codeword the algorithm picks a node N at random, and respectively 7 short-range nodes, 4 at mid-range and 2 from long-range nodes, for a total of 14 target nodes. Finally it places the 14 blocks of the codeword in such nodes. We report results for 3 ranges (`da3`), for 4 ranges (`da4`, for which the percentiles are 25^{th}, 50^{th}, 70^{th} and the number of blocks are 6, 4, 2, 1 for each range, respectively) and finally for 5 ranges (`da5`, using the percentiles 20^{th}, 40^{th}, 60^{th}, 80^{th} and the number of blocks are 5, 5, 1, 1, 1 for each range respectively).[4]

Random-Degree (`drnd`). This strategy combines a naive random strategy with `deg`. Each strategy contributes for the placement of half of the blocks.

[4] The number of blocks assigned to each class of range nodes (`da3`, `da4`, `da5`) has been experimentally proved to work better in practice.

5 Simulation results

This section presents the results of our simulations with the aim of evaluating how the different placement strategies perform with respect to fetch-latency and data loss.

Load Balancing. We begin by studying how the strategies spread blocks on 4 different graph topologies. First, we consider a random graph of 1000 nodes, as depicted in Figure 2 on the left, where we highlight the 10 clusters computed by $\mathcal{K}$-means using the Euclidean distance between nodes. The distribution of blocks among nodes is presented in Figure 2 (right). As expected, the `rnd` strategy produces a Gaussian distribution, while the other approaches tend to flatten and/or shift the bell.

Figure 3 shows topology and block distribution for a scale-free graph of 1000 nodes built using the preferential attachment method [2]. This topology closely maps a real Internet topology, yet is simple to study and analyze. We observe that `deg` and `drnd` produce a long-tail block distribution: several nodes have few blocks (right side of the figure), while few nodes store plenty of blocks (left side of the figure).

Finally, we consider two real-world topologies. The first is the Full European NREN network [14]. This graph has 1157 nodes and 1465 edges. When computing 10 clusters, we observe 1170 inter-cluster edges (i.e. source and destination nodes belong to different clusters). Topology and block distributions are presented in Figure 4. As an empirical confirmation that scale-free graphs are well-suited for representing Internet topologies, we underline the similarity between the two block distributions.

The second real-world topology, depicted in Figure 5, is the Cogent network [14]. It is smaller than the NREN topology (197 nodes, 245 edges) nevertheless it extends across Europe and US. This topology presents trans-oceanic links, with 13 edges to connect nodes across the Europe and North America. Different ranges in the block distribution with respect to other graphs are due to the much smaller number of nodes (while we distribute the same amount of data blocks).

Overall, block distributions generated by the `da` and `rr` strategies tend to be bell-shaped, while `dar` and `deg` entail left-sided pick and long tail corresponding to few blocks in many nodes and few nodes hosting many blocks respectively.

For NREN and Cogent, we know the geographical coordinates of the nodes. To take into account of the curvature of the Earth and place more precisely the centroids of the clusters, we use the Haversine distance [17] as $\mathcal{K}$-means distance function.

We fix the number of cluster $\mathcal{K} = 10$ in our simulations except for Cogent topology which is split in $\mathcal{K} = 2$ clusters corresponding to USA and Europe. For the same reason, results of `ca` are not available for Cogent, since the heuristic requires at least 3 clusters.

Fetching latency. We continue by evaluating how the proposed strategies differ in terms of block recovery latency, as observed by the clients wishing to reconstruct matching tuples. We assume that the fetch-latency is proportional to the distance between nodes. Hence, we measure the length of the minimum paths between the node hosting the target block and the client.

We observe that a node storing a lot of blocks we necessarily need to fetch only few ones to reconstruct tuples. Hence, for each topology and each placement heuristic, we distinguish 3 types of clients based on the number of blocks they store. The *lucky* and the *unlucky* node stores the greatest and the smallest amount of blocks respectively.

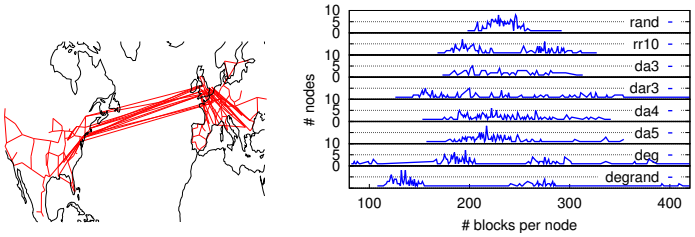
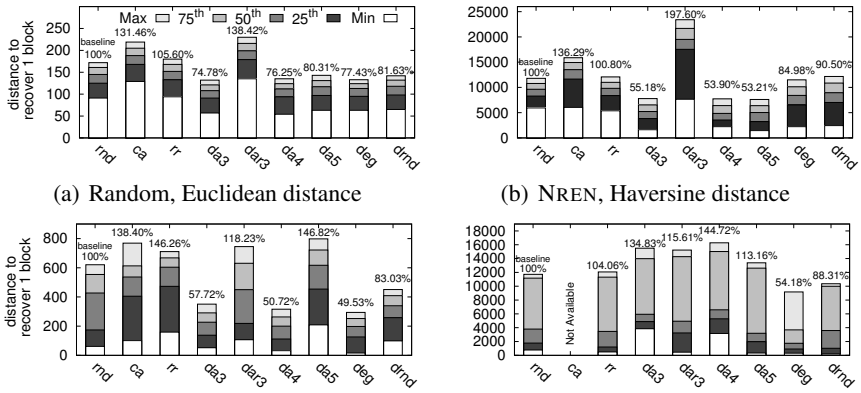


Fig. 5: Cogent topology and its blocks distribution.



(a) Random, Euclidean distance

(b) NREN, Haversine distance

(c) Scale-free, Euclidean distance

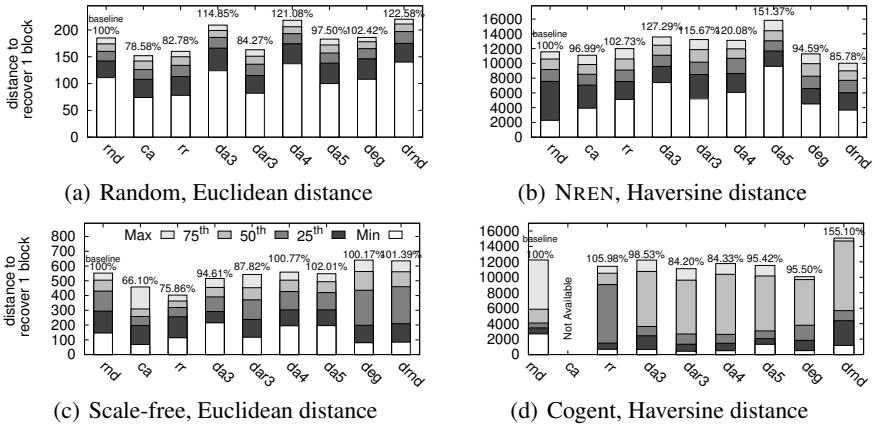
(d) Cogent, Haversine distance

Fig. 6: Distance for fetching blocks (*lucky node*).

We use a representation based on stacked percentiles throughout the reminder of this section. The white bar at the bottom represents the minimum value, the pale gray on top the maximal value. Intermediate shades of gray represent the 25th, 50th –the median– and 75th percentiles. We compare the results against a baseline `rnd` strategy that randomly places blocks across the graph. Figure 6 and 7 presents the case of the *lucky* and *unlucky* node respectively.

These results validate the intuition that the number of blocks the client is storing greatly affects the observed fetch-latency. For instance, `da3` performs better than other heuristics in 3 out of 4 topologies when the client is *lucky*. However, this is not the case for the *unlucky* case, where `deg` and `rr` perform better instead. These observations suggest that no strategy wins in all possible topologies, and that deployers need to carefully consider the different trade-offs for their applications and workloads.

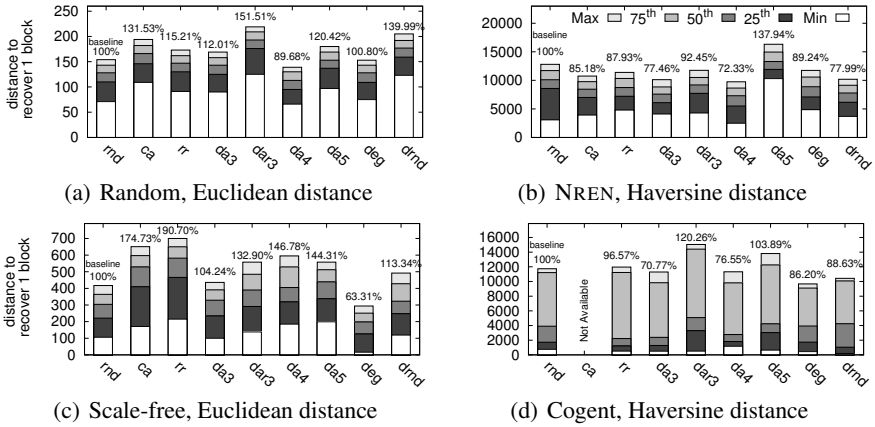
Fetching latency under faults. Next, we perform a set of experiments that faults into the graph. For each graph, we select the most populated among the 10 clusters and we crash 1% of its nodes. This setting simulates a catastrophic event occurring to nodes geographically close to each other. Once the faults are injected, we use the *lucky* nodes (Figure 8) and *unlucky* nodes (Figure 9) to try to reconstruct all data stored.



(a) Random, Euclidean distance (b) NREN, Haversine distance

(c) Scale-free, Euclidean distance (d) Cogent, Haversine distance

Fig. 7: Distance for fetching blocks (*unlucky node*).



(a) Random, Euclidean distance (b) NREN, Haversine distance

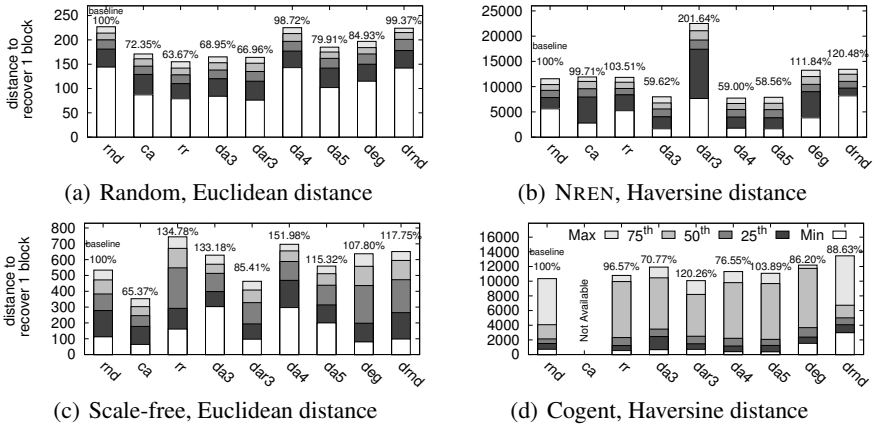
(c) Scale-free, Euclidean distance (d) Cogent, Haversine distance

Fig. 8: 1% crashes in one cluster, lucky node.

During these simulations, we did not observe any data loss. Hence, the heuristics are spreading blocks sufficiently apart from each other to tolerate crashes within the same cluster.

However, when injecting faults the fetch-latency highly depends on the particular failing nodes. In the case of the Cogent topology, the `deg` strategy greatly improves the results produced by the `rnd` placement, while on the scale-free graph performance degrades for the unlucky client. The `da3` strategy outperforms the other heuristics in the NREN topology. More in general, distance-aware heuristics seem to be well-suited for the random graph.

Statistical analysis. Finally, to evaluate the statistical significance of the differences recorded by the simulations between the various heuristics, we perform two sets of *t-tests* [12] on fault-free graphs. First, we build the dataset with one entry for every node. In this entry we compute the cumulative distance, that is, the sum of the length



(a) Random, Euclidean distance (b) NREN, Haversine distance

(c) Scale-free, Euclidean distance (d) Cogent, Haversine distance

Fig. 9: 1% crashes in one cluster, unlucky node.

of all minimum paths covered to retrieve all data in the system from that particular client. We fix a topology and compare different heuristics against each other. We find the following p-values:

	scale-free graph	random graph	NREN	Cogent
t.test(`rnd`,`deg`)	0.2486	0.03055	0.00761	0.7828
t.test(`rnd`,`da3`)	0.4805	0.3242	0.3774	0.2203

These p-values answer the question: "what is the probability that the means of the cumulative distances covered by the two heuristics are equal?". For every graph we found an heuristic between `da3` and `deg` such that the probability is less the 25%. We consider this a low evidence that the two means are the same but still such a value does not provide a decisive response.

For this reason, instead of using cumulative distances, we create a dataset of the distances covered to fetch every block by each node in the graph (e.g. in the case of the scale-free graph the dataset has 1000 entries times the number of blocks fetched, *i.e.* 3276000 entries). We run t-tests on random 1000-entries-samples from this dataset to compare different heuristics against each other. We find the following p-values:

	scale-free graph	random graph	NREN	Cogent
t.test(`rnd`,`deg`)	0.1661	$6 \cdot 10^{-6}$	0.0004	0.6475
t.test(`rnd`,`da3`)	0.4215	0.0406	0.2936	0.1042

So for every topology we can find a heuristic between `deg` and `da3` with support less than 16% for the hypothesis that the distance covered is the same as the one covered by `rand`. We take into account the modeling and statistical results to implement `deg` and `da3` into in a real tuple space and evaluate how they perform in a practical setting.

6 Implementation

We implement and deploy three of the described blocks placement strategies (`da3`, `deg` and `rnd`) atop SIMPLETS,[5] a tuple-space implemented in Python (v3.4.0). The original

[5] https://github.com/jmbjorndalen/SimpleTS

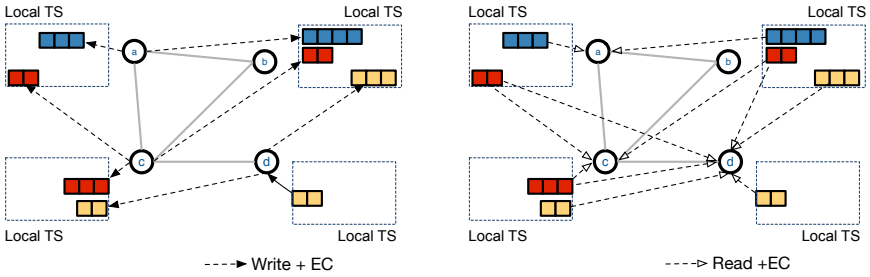


Fig. 10: Distributed tuple space with erasure code: `write` ops. spread blocks apart driven by a specific strategy; `read` ops. fetch blocks from remote nodes.

implementation of SIMPLETS did not support remote tuple space nodes. Therefore, we first extended it to support a distributed scenario, leveraging PYRO (v4.0),[6] a remoting library for Python. Overall, our modifications to the SIMPLETS source code consist of only 250 additional lines of code.

To add erasure coding and block placement techniques, we extend the tuple space APIs with additional operations to properly handle writing, reading, and deletion of encoded tuples. For example, using a $[14, 10]$ Reed-Solomon code, the `out(t)` operation that emits the tuple `t` in the tuple space, becomes `out_ec(t)`. This version encodes the tuple, splits it into 14 blocks and, according to the chosen strategy, distributes these blocks among the other nodes. To this end, from the original tuple a list of tuples of the following form is created: `<tupleUID, blocksAndIndicesList, nodeList>` where `tupleUID` is a unique identifier of the original tuple `t`, `blocksAndIndicesList` is a list of pairs (b_i, i) indicating that b_i is the i-th block of the codeword and `nodeList` is a list of nodes containing the remaining blocks. Figure 10 shows the extended version of SIMPLETS with erasure coding abilities.

In this configuration, reading a tuple only require to fetch 10 out of the 14 existing blocks. The tuple space programming paradigm requires the reading operations to operate via pattern-matching [9]. In the case of encoded tuples, the tuple space needs to decode the tuple. Therefore, this operation sequentially reads a tuple with blocks from the tuple space. Specifically, it leverages the `nodeList` index to discover and retrieve the missing blocks from other nodes in order to reconstruct the tuple. Then it checks whether the reconstructed tuples matches the template. Clearly, in the worst case to find a matching tuple the system has to decode the entire tuple space. We assume the existence of an up-to-date indexing service that serves the purpose of speeding up the process of discovering the location of the required blocks. In our evaluation, we assume to know the location of the nodes storing the blocks required to decode the tuple. It is out of the scope of this work how to efficiently maintain this index.

We implement both `da3` and `deg` strategies on our tuple space and test them on a *scale-free* network made of 100 nodes. We emulate a large-scale network deployment using Docker (v1.13.1). We map each SIMPLETS node (with its local tuple space) to

[6] https://pythonhosted.org/Pyro4/

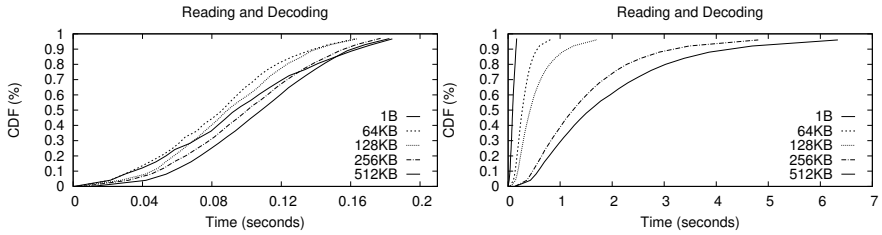


Fig. 11: Distribution (CDF) of tuple's reading performance for increasing tuple size. Left: without erasure-coding. Right: with erasure-coding.

a standalone container. The latency between two nodes, say i and j, is proportional to their minimum distance on the graph. Latency (by mean of a `sleep` system call) is then interposed by the proxy interface of the Pyro service exposed by each tuple space process. In practice, when node i contacts node j to read (or write) a tuple, node j sleeps $latency_{i,j}$ milliseconds before replying. An alternative method is be to add latency at the OS level, e.g. by implementing a software router.[7]

7 Prototype evaluation

This section presents our evaluation with the extended SIMPLETS system. Due to the lack of hardware resources, we are limited to a cluster of 100 node mimicking a *scale-free* network. Each node is executed by a SIMPLETS Docker container. In this evaluation, only communication delays among nodes are emulated.

Erasure-coding overhead. To evaluate the overhead of erasure coding, we execute an initial set of microbenchmarks for reading times. In this experiment, we vary the size of data stored in each tuple, from 1 byte to 512KB. At the beginning, we randomly distribute 1000 tuples across 100 nodes. Then, 10 random nodes read all the 1000 tuples. We measure the time for reading each tuple, and we report them as Cumulative Distribution Function (CDF). As shown in Figure 11 (left), the size of the tuple only modestly affects the reading time from the tuple space without encoding.

When erasure coding is enabled, Figure 11 (right), the reading time is more sensitive to the tuple size: it grows from milliseconds for the tuples containing 1 byte to several seconds for the size of 512KB. For bigger tuples, the time for encoding and decoding is significantly higher. We believe that a highly optimized erasure-code library, such as Intel ISA-L [1], would greatly reduce the overhead and make it more practical.

Experiments with different strategies. This experiment evaluates the performances of the tuple space using different block placement strategies. At the beginning, each of the 100 nodes writes 10 tuples. The tuples are encoded and split into blocks. Those are

[7] We report on our failed attempt in using Linux `tc`'s traffic shaping (using *delay.sh* https://gist.github.com/arr2036/6598137) to emulate network latencies. In particular, the current Docker networking layer does not cope well with this approach, where all nodes in a given network class (such as all the Docker containers running in the same host) apply the same delay, preventing the emulation of more complex graph topologies.

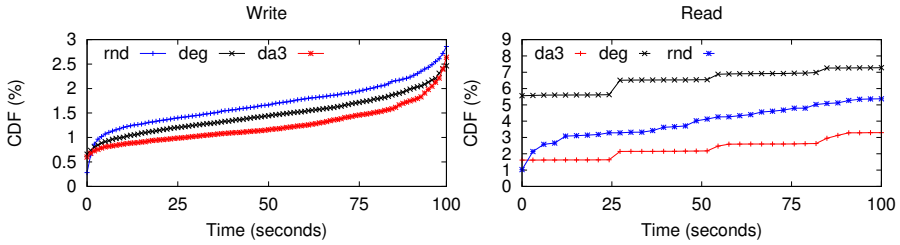


Fig. 12: Writing and the reading times for different strategies of blocks placement

dispatched to remote nodes according to the given strategy. Finally, a node is chosen to fetch and reads the blocks of its own tuples.

Figure 12 presents our results for write and read operations.[8] The plot shows the CDF of the timings to write/read the tuples into/from the tuple space. The `da3` strategy achieves the best performances for writes, because the writing time depends on the number of nodes used to spread each tuple's blocks. Random placement offers the worst performance as it involves a high number of nodes. The reading time depends on the distribution of the blocks among the nodes. The distributions obtained reflect the ones shown in Figure 3 and we do not report them here due to lack of space. For the distance-aware and random strategies, distributions are more uniform and the times are low. For the degree-aware strategy, nodes with the higher degree have considerably more blocks and the reading time vary significantly. As consequence, the reading time depends also on the order in which tuples are written. In the case of SIMPLETS, the tuple space is implemented as a list, hence the reading time will be greater for the tuples which were written toward the end.

8 Conclusion

The problem of data block placement in a wide-area network setting is of paramount importance. Several distributed applications rely on a random strategy. In this paper we considered a scenario where distributed applications are implemented via the tuple space paradigm. These systems need to efficiently cope with network faults to avoid losing tuples, while at the same time being storage efficient and allow fast fetching time. We extended an open-source Python-based tuple-space implementation with distribution capabilities and erasure-coding features. We presented a study of several block placement strategies to dispatch blocks over the nodes of a distributed tuple space. We considered synthetic and real-world graph topologies, up to thousands of nodes. Our modeling, statistical analysis and system performance results, also based the evaluation of our full working prototype, shed some light on the trade-offs that one need to accept when deploying such systems. Our results reinforce the believe that it is important to gather structural informations about the underlying network topology to wisely choose the appropriate block placement heuristic.

[8] We omit the results for withdrawing operations. They show similar trends to read results plus a small overhead due to the fetching of all the 14 blocks.

In this work we considered the distributed tuple space as practical use-case. We stress that our strategies are general purpose and can be deployed in other distributed systems such as distributed key-value stores.

Acknowledgments

The authors are grateful to Hugues Mercier and Pascal Felber for invaluable discussions during the preliminary phases of this work. We are grateful to Rocco De Nicola for fruitful discussions around tuple spaces. This research was partially supported by the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 653884.

References

1. Intel's ISA-L. https://github.com/01org/isa-l.
2. A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
3. A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. Depspace: a byzantine fault-tolerant coordination service. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 163–176. ACM, 2008.
4. A. N. Bessani, M. Correia, J. da Silva Fraga, and L. C. Lung. An efficient byzantine-resilient tuple space. *IEEE Trans. Computers*, 58(8):1080–1094, 2009.
5. V. Buravlev, R. De Nicola, and C. A. Mezzina. Tuple spaces implementations and their efficiency. In *COORDINATION 2016*, volume 9686 of *LNCS*, pages 51–66. Springer, 2016.
6. N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, Cambridge, MA, USA, 1990.
7. M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: flexible data placement and its exploitation in Hadoop. *Proceedings of the VLDB Endowment*, 4(9):575–585, 2011.
8. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
9. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.
10. D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, Feb. 1992.
11. P. Institute. 2013 cost of data center outages. 2013.
12. G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 6. Springer, 2013.
13. H. Jin, X. Yang, X.-H. Sun, and I. Raicu. Adapt: Availability-aware mapreduce data placement for non-dedicated distributed computing. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 516–525. IEEE, 2012.
14. S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.
15. F. J. MacWilliams and N. J. A. Sloane. *The theory of error-correcting codes*. Elsevier, 1977.
16. L. I. Patterson, R. S. Turner, and R. M. Hyatt. Construction of a fault-tolerant distributed tuple-space. In *SAC'93*, pages 279–285, New York, NY, USA, 1993. ACM.
17. C. C. Robusto. The Cosine-Haversine Formula. *The American Mathematical Monthly*, 64(1):38–40, 1957.
18. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.
19. T. White. *Hadoop: The definitive guide*. "O'Reilly Media, Inc.", 2012.