GSpace: Tailorable Data Distribution in Shared Data Space Systems

Giovanni Russello[1], Michel Chaudron[1], Maarten van Steen[2]

[1] Eindhoven University of Technology
[2] Vrije Universiteit Amsterdam

Abstract. The shared data space model has proven to be an effective paradigm for building distributed applications. However, building an efficient distributed implementation remains a challenge. A plethora of different implementations exists. Each of them has a specific policy for distributing data across nodes. Often, these policies are tailored to a specific application domain. Thus, those systems may often perform poorly with applications extraneous to their domain. In this paper, we propose that implementations of a distributed shared data space system should provide mechanisms for tailoring data distribution policies. Through this flexibility the shared data space system can cope with a wide spectrum of application classes. The need for this flexibility is illustrated by experiments which show that there is no single distribution policy that works well in all cases.

1 Introduction

Motivation As distributed systems scale in the number of components and in their dispersion across large networks, the need for loose coupling between those components increases. This decoupling can take place in two dimensions: time and space [4]. Decoupling in time means that communicating parties need not be active simultaneously, this decoupling is present in message-queuing systems. Decoupling in space means that communicating parties need not explicitly have to refer to each other, this decoupling is in publish/subscribe systems.

Generative communication [12], also referred to as data-oriented coordination [15], provides both types of decoupling. In the literature several implementaions generative communication using shared data space systems exist.

A drawback of those systems is their use of a single fixed strategy for distributing data. To meet extra-functional system goals, such as scalability and timeliness, these distribution strategies are often optimized for a specific application domain or technical infrastructure. For instance, [13] proposes a specific distribution strategy to obtain scalability of a distributed shared data space across a large number of components in a wide area network. As a result, those systems are not very flexible. Their reuse to different classes of applications requires intricate modification of the application code.

In this paper, our point of departure is that the trade-off between different extra-functional quality properties can be addressed by a flexible architecture. The flexibility of this architecture consist of the possibility of adapting the distribution policies to application-level characteristics of access to the shared data space. In this way, it becomes possible to provide efficient implementation for a large classes of applications.

Moreover, the distribution and replication of data items is such that an efficient distributed shared data space can be realized. To this end, we have built a system that realizes distributed shared data spaces in which each data type is distributed and replicated according to a dedicated strategy.

Another important innovation introduced in our system is that the tailorability of distribution needs of an application is carried out transparently to the application itself. This is in line with the principle of Separation of Concerns (SoC), where functional requirements of an application are separated from its non-functional requirements. In this way, it is possible to reuse the same application code in several environments where different distribution strategies are required.

Contributions We make two contributions. First, we demonstrate how this differentiation of strategies outperforms fixed strategies. We note that differentiating strategies by itself is not new and that it has been applied to distributed shared memory systems [3, 6], and to some extend proposed also for shared data space systems [19]. However, this work is the first to demonstrate also the need for differentiation in shared data spaces. Second, we show that continuous adaptation of strategies may be needed, which in turn requires a monitoring and feedback system to adjust previously chosen strategies.

Roadmap The paper is organized as follows. In section 2 we briefly introduce the shared data space model. Section 3 describes some common distribution strategies, together with a method to measure their performance. Section 4 explains our implementation of a distributed shared data space. Subsequently, in section 5 we discuss the results of the experiments. We conclude with some final remarks and future work.

2 System Model

We consider a distributed system in which nodes are connected by a communication subsystem that provides efficient multicasting facilities. Typically, such facilities are offered in local-area networks. However, efficient implementations are also offered for wide-area systems, either following traditional approaches as in PGM [20], or exploiting novel peer-to-peer networks [7, 1].

A shared data space is capable of storing **tuples**. A tuple is an indivisible, ordered collection of named values, bearing resemblance with records in databases and programming languages. Each node stores a part of the content of the shared data space. Access to this part is controlled locally. These local data spaces are hidden from applications. Instead, applications are offered a simple interface to access a shared data space, consisting of the three operations presented in Figure 1.
In this paper, we simplify matters by adopting the semantics of the corresponding operators as specified for JavaSpace [14].

3 Data Space Distribution

Background We are interested in distributing and replicating tuples such that we obtain an efficient implementation of the shared data space. In the past, researchers have sought

Operation	Description
put(tuple)	Stores a given tuple in the data space.
read(template)	Reads an arbitrary tuple that matches *template* from the data space. If no match can be found, the caller is blocked.
take(template)	Removes an arbitrary tuple that matches *template* from the data space. If no match could be found, the caller is blocked.

Fig. 1. The three data space operations.

a solution to the efficiency problem by providing an implementation that was tailored to a specific application domain. Proposed implementations differ in the distribution of tuples. Examples include:

Statically centralized: Only a single, fixed node has a local data space where *all* tuples are stored. This is the common implementation of a nondistributed, but remote accessible shared data space. Examples of systems following this approach include both JavaSpaces [11] and TSpaces [21]. This approach has the drawbacks common to all centralized designs. The single node where the data space resides may become a bottleneck under a high load of requests; and it represents a single point of failure.

Dynamically centralized: A system that follows this approach is Lime [16]. In Lime each process stores tuples in its local data space. The data space is permanently bound to the process. Processes join and leave the computational environment, together with their local data space. The local data spaces collaborate with each other to give to processes a logical view of a single shared data space. This means that the content of the space dynamically changes when local data spaces join or leave the system. In Lime both processes and data spaces have unique identifiers. If a process has to leave and wants that its tuples are still available, it has to declare the identifier of the destination data space where the tuples should be moved. This peculiarity contrasts the basic principle of the data space model of space decoupling. In fact, to transfer the tuples, the sender has to know the receiver.

Statically distributed: In this case, each tuple is stored at a single node, but different tuples may be stored at different nodes. The node n responsible for storing a tuple t is determined by a hash function H: $n = H(t)$. This approach, with some modifications, has been adopted in the run-time environment developed at York [10]. An advantage of using a hashing function for rooting tuple requests is the reduction of searching time. On the other hand, it does not support changes in the environment configuration. A reconfiguration of the system requires a costly re-mapping of the entire data space content.

Fully replicated: Each tuple is replicated to every node. This strategy has been applied in [9]. Since tuples can be found locally, the searching time is null. However, this strategy needs sophisticated mechanisms to control the consistency of the data space in the presence of removal operations. Generally, these mechanisms perform poorly when the number of nodes increases.

Structurally replicated: Tuples are replicated according to a structural schema. In [5], the schema is based on a grid of nodes formed by logical intersecting busses. Each

node belongs to one *outbus* and one *inbus*. A tuple is replicated in all nodes that form the outbus. Whereas, retrieve operations are executed on the nodes of the inbus. Since an inbus intersects all outbusses, the search is extended to the entire data space. Another schema, used in [8] follows a tree structure. The leaves of the tree represent the nodes where the processes reside. In the internal nodes of the tree the data space is distributed. A tuple is replicated along the path that starts from the leaf node where the tuple was generated, and goes up to the root node. A search follows a path that starts from a leaf node and may go up till the root node. Also in this case a search is extended to the entire set of tuples stored in the data space, because eventually a searching path will get to the root node where all tuples are stored. Both these approaches scale down the problem of consistency to a part of the data space. However, as the number of nodes increases they incurs in the same problem of the previous strategy.

We make two observations. Firstly, a shared data space system that offers just one global distribution policy is not flexible enough, as it can satisfy the distribution requirements of only a single class of applications. Secondly, in case that several distribution policies are available, it is important to determine which policy matches an application's needs. It is unclear whether application developers are always capable of making such a decision.

Another issue that needs to be addressed is the data granularity at which a policy is applied. In virtually all shared data space systems, a single distribution policy is applied to the entire data space. In our own research on Web hosting services, we observed that associating a distribution policy for each Web document separately allows to obtain close-to-optimal performance. In other words, differentiating distribution policies at a finer granularity than an entire data space may be beneficial.

We hypothesize that ideally, a shared data space system supports multiple distribution policies, and that it can automatically determine which policy should be applied to a given application. Moreover, policies should be differentiated at a much finer granularity than the entire data space. If a policy change is needed, a shared data space system should be able to detect such a need and automatically adapt to a better policy.

Policy evaluation A distribution policy incurs various costs. For example, full replication may make read operations cheap in terms of latency, but update operations such as put and take may be expensive. Likewise, a higher price needs to be paid for storage usage when comparing full replication to hash-based distribution of tuples, or centralized solutions.

We need a means to compare the performance of policies. To this end, where performance is expressed in terms of different metrics such a client-perceived latency, bandwidth usage, storage usage, etc. Following the approach described in [17], we adopt the use a **cost function** CF which is a linear combination of n metrics $m_{1,p},\ldots,m_{n,p}$ produces by a policy p:

$$CF(p) = w_1 * m_{1,p} + w_2 * m_{2,p} + \ldots + w_n * m_{n,p} \quad (1)$$

Here, $w_1, \ldots, w_n$ are weights that determin the relative influence of each metric, that is, $\sum w_i = 1$ with $w_i \geq 0$. Given a set of weights, the policy p that minimizes $CF(p)$ is considered the best. We return to the use of this cost function below.

4 GSpace: A System for Distributed Shared Data Spaces

For this paper, we are interested in two questions: (1) what can we gain from tailoring tuple distribution to applications characteristics, and (2) to what extent do we need to dynamically adapt policies while executing an application?

To answer these questions, we have built a prototype of a distributed shared data space that supports a variety of distribution policies. We have used this prototype to emulate different applications on a real local-area network. In other words, rather than conducting simulations, we decided to measure the effect of different policies on actual resource usage. In this section we briefly describe our prototype, which we have named **GSpace**.

4.1 GSpace's Features

GSpace presents some distinct and novel qualities with respect to previous work:

- GSpace is a distributed shared data space. This means that a typical GSpace set-up consists of several *GSpace kernels* running on different nodes. Yet, from the application point of view GSpace is a single data space, preserving its simple programming model.
- GSpace's design allows applications to separate the functionality from its extra-functional concerns, such as data distribution. In GSpace, tuple distribution requirements are declared separate from the application code. This separation facilitates the reuse of the same application code in different environments that require different data distribution.
- GSpace embeds a suite of distribution policies that can be used to tailor the behavior of different applications.
- GSpace's suite of distribution policies can easily be extended. Users can develop their own distribution policy and download it in the middleware, without having to change application code.

4.2 Architectural View

Figure 2 shows the internal structure of a GSpace kernel and the modules where the sensors for measuring resource usage are placed. A more detailed description of the kernel modules can be found in [18]. Here, we just provide a concise description of the modules relevant to the discussion of this paper.

- **Controller and Latency Sensor**. The Controller provides the API of GSpace to application components. This API consists of three operations: put, read, and take. In this module the sensor for timing the latency of operation execution is placed.

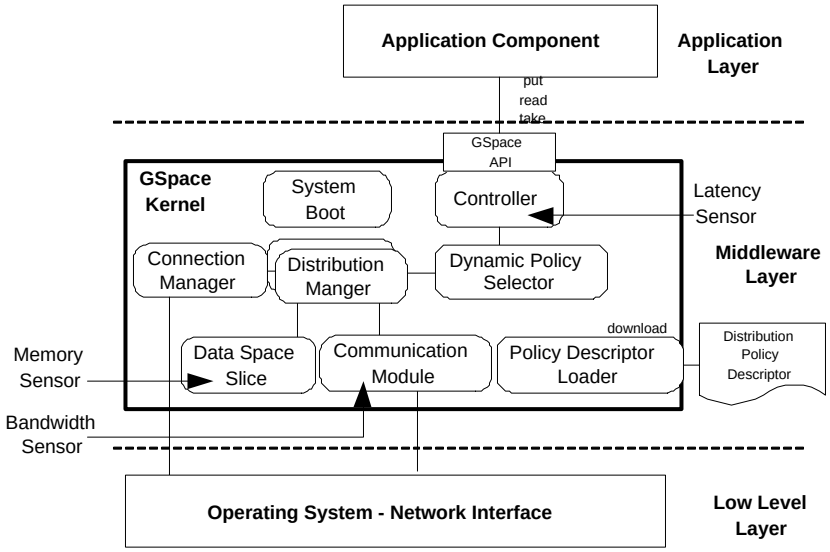

Fig. 2. Internal view of a GSpace kernel and sensor placement.

- **Data Space Slice and Memory Sensor**. Each GSpace kernel is supplied with a local tuple storage called Data Space Slice. The Memory Sensor measures the amount of memory that is used for storing tuples on each kernel.
- **Communication Module and Bandwidth Sensor**. The Communication Module supplies the support for exchanging tuple between the kernels using different forms of communication, such as point-to-point and multicast messages. In this module the Bandwidth Sensor is installed. The task of this sensor is to measure the amount of bandwidth that is used for sending tuples and synchronization messages on the network.

GSpace provides a suite of distribution policies that supports several strategies for distributing tuples across GSpace kernels. In this way, GSpace can differentiate distribution strategies per tuple type. In fact, we assume that in GSpace tuples are *typed*, and for each tuple type within an application it is possible to apply a different distribution policy.

A distribution policy is implemented by a dedicated **Distribution Manager** (DM), installed in each GSpace kernel (see Figure 2). A DM carries out the execution of the data space operations according to the policy implemented. Every time an operation is executed the **Dynamic Policy Selector** (DPS) inspects the type of the tuple or template and retrieves the associated distribution policy. After that, the DPS passes the control of execution to the corresponding DM.

The association between tuple types and distribution policies is obtained from a **Distribution Policy Descriptor**. This is a file that is made available to all locations in

which a GSpace kernel is instantiated. At kernel start-up time, the **Policy Descriptor Loader** downloads the information in the internal data structure of GSpace to make the information available at run-time to the DPS. Currently, the foolowing distribution policies are supported by GSpace:

Store locally (SL): A tuple is always stored on the slice that excutes its put operation. Likewise, read or take operations are performed locally as well. If the tuple is not found locally then a request is forwarded to other nodes.

Full replication (FR): Tuples are inserted at all nodes. The read and take operations are performed locally. However, a take has to be forwarded to all nodes by means of a totally-ordered broadcast, in order to remove all copies.

Cache with invalidation (CI): A tuple is stored locally. When a remote location performs a read operation, a copy of the tuple is subsequently cached at the requester's location. When a cached tuple is removed through a take operation then an invalidation message is sent to invalidate all other cached copies of that tuple.

Cache with verification (CV): This policy is similar to CI, except that invalidations are not sent when performing a take. On reading a cached tuple, the reader verifies whether the cached copy is still valid, that is the original has not been removed.

5 Experimental Results

In this section we discuss the experiments that we set up for investigating the effect of using different distribution policies for several application behavior.

5.1 Application Model Description

To validate our research we conducted a series of experiments. The experiments consist of executing an application composed from several components that exchange data via GSpace. The application is designed to simulate several application usage patterns. A usage pattern of an application is characterized by the order and the ratio in which the application's components execute data space operations.

Figure 3 shows the experiment setup. On each of several node one GSpace kernel was instantiated together with one application component. All application components are equal, except for the component called *coordinator*. The coordinator is a special component that acts as a director in an orchestra, directing and coordinating the actions of each application components. The coordinator resides in one of the nodes used for the experiment. It is connected directly with each component, thus communication between coordinator and application components is external to GSpace.

During an experiment run, the coordinator executes the following steps:

1. generate a sequence of data space operations in a certain ratio
2. generate the schedule in which components have to execute the operation sequence
3. execute the schedule, dispatching each operation in the sequence to the components
4. when all the operations of the sequence have been executed, change the distribution policy associated with tuples, reset the data space content, and repeat step 3. If no more distribution policy is available, then stop.

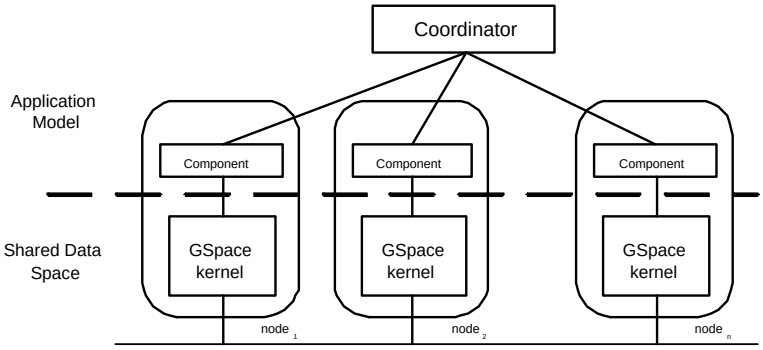

Fig. 3. The setup used for the experiments.

During the experiments, we measured the actual values of costs that each distribution policy produced. The same sequence and the same schedule are repeated for all distribution policies. This ensures that the comparison between the distribution policy is unbiased by randomization effects.

5.2 Application Usage Patterns

In this section we list several application usage patterns that were simulated in our application model. This does not pretend to be an exhaustive collection of usage patterns.

Local Usage Pattern (LUP): In this case, tuples are retrieved from the slice on the same node where they have been inserted. This could be the case if components store some information for their own use or if producer and consumer of a tuple type are deployed on the same node.

Write-many Usage Pattern (WUP): In this usage pattern applications on different nodes need to frequently and concurrently update the same tuple instance. This is problematic for the consistency of distributed shared-memory systems, since extra mechanisms are needed for mutual exclusion.

Read-mostly Usage Pattern (RUP): In this usage pattern, application components execute mostly read operations on remote tuples. We distinguish two variants of this pattern: 1) RUP(i), where applications might execute tuple updates between sequences of read operations. An example could be of a tuple type representing a list-of-content. 2) RUP(ii), between the insertion of a tuple and its removal only read operations are executed. This could be the case of tuple type representing intermediate-result data in a process-farm parallel application.

5.3 Results

In our experiments, we tested each application usage patterns with the currently available distribution policies in GSpace, described in section 4.2. During runtime, the sensors placed in the system measure the following cost parameters:

– The latency for the execution of a read operation (rl)
– The latency for the execution of a take operation (tl)
– The network bandwidth usage (bu)
– The memory consumption for storing the tuples in each local data space (mu)

For this specific set of parameters, the cost function (for policy p) becomes:

$$CF(p) = w_1 * rl_p + w_2 * tl_p + w_3 * bu_p + w_4 * mu_p \quad (2)$$

Since put operations are executed without blocking the application components, we decided not to use the latency of put operations as a parameter for the CF. For the calculation of the CF values we use the same value for all weights w_i, that is 0.25.

The optimal policy for the application usage pattern is represented by the policy that produces the lowest CF value. It should be noticed that the proposed approach implies that the optimal policy is identified through a comparison of their execution performance, rather than via a prediction. Once the performance profile for a distribution policy is known, it can be used in future cases to match it to a given application profile.

All experiments were executed on 10 nodes of the DAS-2 cluster [2] allocated exclusively for a GSpace kernel and an Application Component. Each DAS-2 node is a Dual Pentium-III workstation interconnected by Myrinet, a multi-Gigabit LAN. Since this is a multi-user environment each experiment was executed several times and at different times of the day to avoid that the execution of other user tasks could influence our measurements.

Each usage pattern was simulated using sequences of 500, 1000, 2000, 3000, 5000 operations.

For brevity reasons, the histograms in Figure 4 only illustrate the results obtained during the experiments of 5000 operations. The rest of the experiment results can be found in the Appendix of this paper. The results collected with sequences of operations with shorter lengths follow the same trend. The only differences are in the relative CF values. For each histogram, the X-axis shows the distribution policies and the Y-axis represents the respective CF values.

Figure 4-(a) shows the results collected when the LUP was simulated. Under these conditions, the policy **SL** produces the lowest CF value. In fact, SL guarantees low cost for the execution of space operations that take place on local tuples. Instead, other policies provide more sophisticate strategies to distributed tuples at the cost of using more resources.

Figure 4-(b) shows the results obtained when the WUP was simulated. In this scenario, several components need to access and modify the same instance of a tuple. The lowest CF value was output by policy **FR** In fact, the extra resources needed for replicating tuples lower the access time, since no search time is required for finding a matching tuple. The other policies perform equally bad, since for all of them read and take operations require a global search, when the tuple is not stored locally.

The results in Figure 4-(c) and (d) are obtained when the RUP(i) and RUP(ii) were simulated. The values in the histograms are reported in logarithmic scale. In both scenarios, application components execute multiple reads on the same tuple instance and

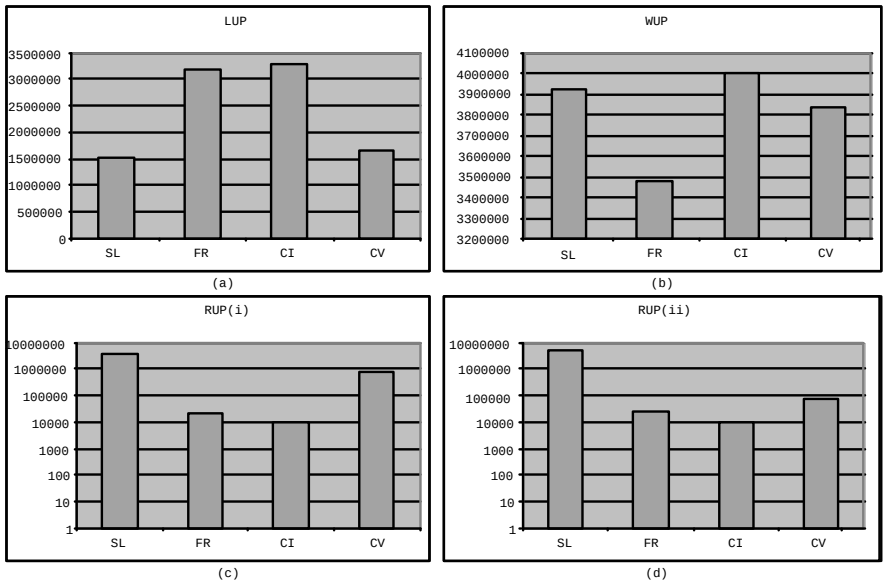

Fig. 4. For each application usage pattern, a histogram shows the cost incurred by different distribution policies.

few take operations. In both cases, the lowest *CF* values were produced by the **CI** strategy, since caching allows to execute most of the read operations locally. However, policy CV performs always considerable worst than policy CI due to the validation message that the reader has to send for each operation executed on the cache. Thus, both latency time and bandwidth usage for read operations result higher than in CI. The performances of the FR policy are close to the one of CI. However, the FR strategy uses more memory for storing replicas in all node; whether CI stores the cached tuples only in those nodes where it was requested.

5.4 A Case for Dynamic Adaptation

Figure 5 shows some unanticipated results collected for a set of experiments with the Read mostly usage pattern RUP(i). Here, the ratio of *number of read operations* to *number of take operations* differs from the experiment in 4-(c). The *X*-axis shows the length of the run; i.e. number of operations. The *Y*-axis shows -on a logarithmic scale- the cost incurred by the distribution policies. The experiments described before suggest that the best policy for RUP(i) is CI. Instead, the graph shows that only for shorter runs, cost is minimized by the CI policy. As the number of the operations increases policy FR outperforms policy CI.

The reason for this changing of policy performances is due to the increased number of take operations executed for each run. This fact has two effects that jeopardize the

performance of policy CI. Firstly, the execution of more take operations reduces the benefits introduced with caching since cached tuples are more often invalidated. Thus, read operations have to search for a matching tuple, increasing latency time and bandwidth use. On the other hand, policy FR replicates tuples at every insertion thus replicas are already available locally. Secondly, for each take operation policy CI uses point-to-point messages for cache invalidation. Instead, policy FR exploits the more effective atomic multicast technique for removing replicas, that reduces resource usage.

What we see is that even given the behaviour of an application, it is difficult to predict which policy it fits best. One solution is to make more accurate models for predicting the cost of policies from behaviour. Building these models is quite intricate. For one thing, it is quite complex to determine all the parameters needed for such a model. An alternative approach is to let the system itself figure out which policy works best. In [17] an approach is reported in which a system automatically selects the best strategy for caching Web pages. This approach works by internally replaying and simulating the recent behaviour of the systems for a set of available strategies. Based on this these simulations, the system can decide which policy works best for the current behaviour of the system. We are extending GSpace to include such a such a mechanism that can dynamically select the best available distribution strategy.

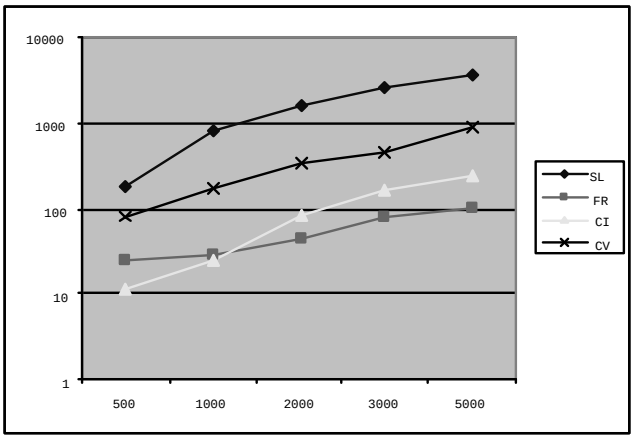

Fig. 5. Results of the simulation for the RUP(i) with different operation lengths.

5.5 Discussion

The experiments presented here show that there is a benefit in having different distribution policies. This characteristic makes the system flexible enough to cater efficiently for different application usage patterns. Thus, it is possible to minimize the resource usage for tuple distribution when the distributed policy that fits the needs of the application is used. Secondly, the unpredicted behavior discussed in section 5.4 stresses

the importance of the impact that GSpace's flexibility has at application level. In designing GSpace we adopted SoC principles to keep extra-functional concerns (such as data distribution) separated from application functionality. This allows us to change distribution strategy without the need to modify one single line of application code. As consequence, the system could switch from one distribution strategy to another at run-time, without interrupting the application execution.

6 Conclusion and Future Work

In this paper we presented GSpace, a distributed shared data space system. The architecture of GSpace provides option of tailoring distribution policies to needs of applications with different usage patterns. We also described a method that allows us to measure the performance of distribution policies during the execution of applications. Finally, we discussed a series of experiments, where we measured the performances of several distribution policies used by applications with different distribution needs.

The results show how dramatically the performance of a distribution policy changes when the application behavior changes. There is no single distribution strategy optimal for all application usage patterns. Instead, having several distribution strategies allows the system to efficiently cater for the requirements of several applications.

Another important result of our experiments is the urge to have in the system a mechanism able to monitor at run-time the application behavior. In this way, the system is aware when the actual distribution policy is no more the most efficient one. When this happens, the system can adapt dynamically to the new needs of the application by switching distribution policy. Moreover, since the data distribution concerns are defined outside the code of the application, switching to a different policy does not require any changes in the application code.

Indeed, GSpace can be considered a flexible distributed shared memory system. However, GSpace differs from other systems because its flexibility is extendable. In other words, GSpace is designed in such a way that new distribution policies can be downloaded in system at any time.

As future work, we are currently implementing the mechanism that allows GSpace to dynamically change distribution policy. Another future development aims to extend the extra-functional concerns that GSpace supports, such as exception handling and timing behavior of an application.

References

1. L. O. Alima, A. Ghodsi, P. Brand, and S. Haridi. "Multicast in DKS(N,k,f) Overlay Networks." In *Proc. 7th International Conference on Principles of Distributed Systems*, Lecture Notes on Computer Science, pp. 80–91, Dec. 2003. Springer-Verlag, Berlin.
2. H. Bal et al. "The Distributed ASCI Supercomputer Project." *Oper. Syst. Rev.*, 34(4):76–96, Oct. 2000.
3. H. Bal and M. Kaashoek. "Object Distribution in Orca using Compile-Time and Run-Time Techniques." In *Proc. 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 162–177, Sept. 1993. Washington, DC.

4. G. Cabri, L. Leonardi, and F. Zambonelli. “Mobile-Agent Coordination Models for Internet Applications.” *Computer*, 33(2):82–89, Feb. 2000.
5. N. Carriero,and D. Gelernter. “The snet’s linda kernel.” *ACM Transaction on Computer System*, 4(2):110-129, 1986.
6. J. Carter, J. Bennett, and W. Zwaenepoel. “Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems.” *ACM Transactions on Computer Systems*, 13(3):205–244, Aug. 1995.
7. M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. “Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure.” *IEEE Journal on Selected Areas in Communication*, 20(8):100–110, Oct. 2002.
8. A. Corradi,F. Zambonelli, and L. Leonardi. “A Scalable Tuple Space Model for Structured Parallel Programming.” *Pro.* Conference on Massively Parallel Programming Models, IEEE CS Press, Pages 25-32, Oct. 1995. Berlin.
9. A. Corradi, L. Leonardi, and F. Zambonelli “Strategies and protocols for highly parallel linda servers.” *Software: Practice and Experience*, 28(14), 1998.
10. A. Douglas, A. Wood, and A. Rowstron “Linda implementation revisited” In *Transputer and occam developments*, pp. 125-138. ISO Press, 1995.
11. E. Freeman, S. Hupfer, and K. Arnold. JavaSpaces principles, patterns, and practice. Addison-Wesley, Reading, MA, USA, 1999.
12. D. Gelernter. “Generative Communication in Linda.” *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
13. R. Menezes, R. Tolksdorf, and A. Wood. “Scalability in Linda-like Coordination Systems.” In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, (eds.), *Coordination of Internet Agents*. Springer-Verlag, Berlin, 2001.
14. S. Microsystems. *JavaSpaces Service Specification*, Oct. 2000.
15. G. Papadopoulos and F. Arbab. “Coordination Models and Languages.” In M. Zelkowitz, (ed.), *Advances in Computers*, volume 46, pp. 329–400. Academic Press, New York, NY, Sept. 1998.
16. G. P. Picco, A. L. Murphy, and G.-C. Roman. “Lime: Linda Meets Mobility.” In *Proc.* 21st International Conference on Software Engineering (ICSE’99), ACM Press, ISBN 1-58113-074-0, pp. 368-377, Los Angeles (USA), D. Garlan and J. Kramer, eds., May 1999.
17. G. Pierre, M. van Steen, and A. Tanenbaum. “Dynamically Selecting Optimal Distribution Strategies for Web Documents.” *IEEE Transactions on Computers*, 51(6):637–651, June 2002.
18. G. Russello, M. Chaudron, and M. van Steen. “Customizable Data Distribution for Shared Data Spaces.” In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2003.
19. J. G. Silva, J. Carreira, and L. Silva. “On the design of Eilean: A Linda-like library for MPI.” In *Proc. 2nd Scalable Parallel Libraries Conference*, IEEE, October 1994.
20. T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano. “PGM Reliable Transport Protocol Specification.” RFC 3208, Dec. 2001.
21. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. “T Spaces.” *IBM System Journal*, 37(3):454-474, 1998.

A More Results

The complete set of results is shown in the table of Figure 6. Each row represents the execution of an experiment run. The first column specifies the usage pattern and the

number of operations executed for a run. The rest of the columns show the cost values for each distribution policy. The cost values displyed in bold font represent the lowest cost per run.

To make things clear, let us consider as an example the first row of values. This row represents the execution of the usage pattern LUP for 500 operations. The lowest cost value is located in the second column, meaning that is output by policy SL. Thus, policy SL is the optimal policy for this scenario.

Usage Pattern	SL CF value	FR CF value	CI CF value	CV CF value
LUP 500	**151032**	331335	324324	163060
WUP 500	4015551	**362997**	531881	392769
RUP(i) 500	180623	23744	**11095**	79318
RUP(ii) 500	212878	30211	**17067**	87033
LUP 1000	**299019**	638544	642107	322832
WUP 1000	802147	**695475**	1047568	772172
RUP(i) 1000	815637	27379	**23774**	171240
RUP(ii) 1000	509192	28872	**14524**	161038
LUP 2000	**616917**	1301450	1324753	666047
WUP 2000	1552213	**1384384**	2087047	1522862
RUP(i) 2000	1577363	695518	**78588**	348454
RUP(ii) 2000	636314	50532	**42407**	328539
LUP 3000	**930552**	1958648	2012760	1012300
WUP 3000	2327112	**2026492**	3086643	2261594
RUP(i) 3000	1909641	22064	**6681**	466869
RUP(ii) 3000	2145815	20648	**3670**	463654
LUP 5000	**1523109**	3182077	3270685	1644407
WUP 5000	3924117	**3483025**	5278553	3837958
RUP(i) 5000	3565590	23438	**9907**	767482
RUP(ii) 5000	4782720	25042	**10601**	76787

Fig. 6. The complete set of values produced during the execution of each usage pattern with all distribution policies.