Exploiting Differentiated Tuple Distribution in Shared Data Spaces

Giovanni Russello[1], Michel Chaudron[1], and 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

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 [3]. Time decoupling means that communicating parties need not be active simultaneously. Space decoupling means that communicating parties need not have an explicit reference to each other.

Generative communication [10], also referred to as data-oriented coordination [12], provides both types of decoupling. In the literature several implementations of generative communication using shared data space systems have been proposed. To meet extra-functional system properties, such as scalability and timeliness, these distribution policies are often optimized for a specific application domain or technical infrastructure. This hard-wiring of a single policy limits the ability of these systems to suit different application characteristics.

Instead, we propose to cater for a wide variety of extra-functional requirements by a using flexible architecture. This architecture provides the possibility of adapting the distribution policies to application-level characteristics of access to the shared data space. In this way, the implementation provides a means to balance extra-functional properties of a system, such as performance, resource use and scalability, for a large class of applications.

In our design of a distributed shared data space, we apply the principle of separation of concerns. This means that we address functional requirements of an application separately from its extra-functional requirements. In particular, we propose to separate the policies for distributing data between nodes from the application functionality. Through

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 579–586, 2004.

this separation, tuning the distribution policy for extra-functional properties such as low latency or low bandwith use becomes transparent to the application. Also, through application of this principle, application logic and distribution logic are separate units of implementation. In this way, both the application code and the distribution code can be reused in different environments.

To substantiate this claim, we show in this paper, that matching the distribution policy with an application's needs, yields better performance than any single distribution policy. While, differentiation of policies has been applied to distributed shared memory systems [2, 5] and in shared data space [4, 15], this paper is the first to demonstrate with concrete results the need for differentiation in shared data spaces. Furthermore, we present experimental results that suggest that continuous adaptation of policies may also be needed.

The paper is organized as follows. In Section 2 we introduce the shared data space model and common distribution schemes. We also explain succintly our distributed shared data space implementation. In Section 3 we describe the setup for our experiments, followed by a discussion of the results in Section 4. We conclude in Section 5.

2 The Shared Data Space Model

A shared data space is capable of storing **tuples**. A tuple is an indivisible, ordered collection of named values. Tuples may be *typed*. Applications can interact with the data space via the three operations described in Figure 1. In this paper, we adopt the semantics of the corresponding operators as specified for JavaSpaces [11].

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.

Various approaches have been followed for constructing distributed shared data spaces. However, the most common approach is still to build a *centralized* data space, in which all tuples are stored at a single node. Examples of this approach include JavaSpaces [9] and TSpaces [18]. The obvious drawback is that the single node may become a bottleneck for performance, reliability and scalability.

For local-area systems, a popular solution is the *statically distributed* data space, in which tuples are assigned to nodes according to a systemwide hash function [14]. Static distribution is primarily done to balance the load between various servers, and assumes that access to tuples is more or less uniformly distributed. With the distributed hashing techniques as now being applied in peer-to-peer file sharing systems, hash-based solutions can also be applied to wide-area systems, although it would seem that there is a severe performance penalty due to high access latencies.

Fully replicated data spaces have also been developed, as in [8]. In these cases, which have been generally applied to high-performance computing, each tuple is replicated to every node. Since tuples can be found locally, search time can be short. However, sophisticated mechanisms are needed to efficiently manage the consistency amongst nodes. The overhead of these mechanisms limits the scalability to large-scale networks.

There are other examples of distributing shared data spaces, but in all cases the succes of these schemes has also been fairly limited. The main reason is that shared data spaces, like relational databases, essentially require content-based searching in order to read data. This type of searching is inherently expensive in large-scale settings, as has again recently been illustrated by the research on unstructured overlay networks [6, 7].

The approach we take, is that by dynamically differentiating how tuples should be distributed in a shared data space, we can achieve significant performance gains in comparison to any static, systemwide distribution scheme. The best scheme highly depends on the applications that access the shared data space. For this reason the supporting middleware should be able to support a myriad of schemes. Eilean [15], and in its successor OpenTS [4], are the first shared data space systems where tuples can be treated differently according to the use within the applications, similar to our approach. However, those systems present a static set of distribution strategies that can not easily extended. Moreover, the programmer has to provide explicit information about which distribution strategy has to be applied to each tuple, solely based on his/her knowledge of the application. Finally, the association tuples to distribution strategies is tangled within the application code, contrary to the principle of separation of concerns.

Our solution is called **GSpace**. A GSpace system consists of several *GSpace kernels* running on different nodes. Each kernel stores a part of the overall data space (called a *slice*), as shown in Figure 2. The kernels communicate with each other to present applications with a view of a logically unified data space, thus preserving its simple programming model.

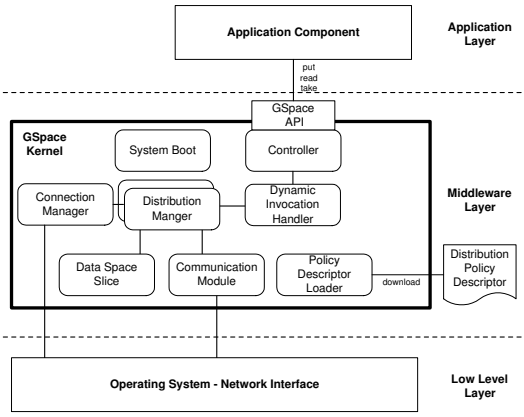

Fig. 2. The internal organization of a GSpace kernel.

Each kernel contains several *distribution managers* that are responsible for distribution of tuples. These modules each employ a different distribution policy for different tuple types, and are completely separated from application components. In other words: data distribution is carried out without specifing any details in the application code. Moreover, the set of policies is extensible such that new distribution policies can be defined. Distribution policies can be inserted in the middleware either at design or at run-time. Further details on GSpace internals can be found in [16].

3 Experiment Setup

To investigate the effect of using different distribution policies for different applications, we set up the following experiments.

We defined a number of patterns that characterize how distributed applications use the data space. Such a usage pattern consists of (1) the ratio of read, put and take operations, (2) the ordering in which these operations are executed, and (3) the distribution of the execution of these actions across different nodes. To avoid randomization anomalies, we generate a set of *runs* that comply with specific usage patterns. We execute the set of runs for different distribution policies. During execution of a run, we measure system parameters that are indicators of costs produced by a distribution policy.

We examined the following application usage patterns, which we considered to be representative for a wide range of applications.

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.

As we mentioned, we are interested in examining how differentiating distribution policies can improve performance. To this end, we designed and implemented four different policies, which we subsequently applied to each of the three application usage patterns. The four different policies are the following:

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.

To compare the distribution policies we follow the approach described in [13]. We define a **cost function** (*CF*) as a linear combination of metrics that represent different aspects of the cost incurred by a policy. We used the following metrics in the cost function: *rl* and *tl* represent the average latency for the execution of read and take operations; *bu* represents the total network bandwidth usage; and *mu* represents the memory consumption for storing the tuples in each local data slice. For these parameters, the cost function for a policy p becomes:

$$CF_p = w_1 * rl_p + w_2 * tl_p + w_3 * bu_p + w_4 * mu_p \quad (1)$$

Because put operations are non-blocking, application components do not perceive any difference in latency for different distribution policies. Therefore, the put latency is not used as a parameter for the cost function.

The w_i's control the relative contribution of an individual cost metric to the overall cost. An application engineer can set these parameters to match his preference of the relative importance of the different cost metrics. For the experiments in this paper, we take $w_i = 0.25$ for all i. Clearly, the settings of these weights determine the performance of the policies. The relevance of these results is not to identify the best setting of the weights, but to illustrate that different policies can be ranked according to a cost-criterium and that for different application characteristics different policies perform best (this holds for any setting of weights).

In our experiments, we simulated all application usage patterns with the policies described previously. The best policy for an application usage pattern is the one that produces the lowest cost value.

4 Results

All experiments were executed on 10 nodes of the DAS-2 [1]. Each usage pattern was simulated using runs of 500, 1000, 2000, 3000, and 5000 operations. For brevity reasons, the histograms in Figure 3 only illustrate the results obtained using runs of 5000 operations. In each histogram, the X-axis shows the distribution policies and the Y-axis represents the respective *CF* values. The results of shorter sequences of operations follow the same trend. The complete results of these other experiments can be found in the extended version of this paper [17].

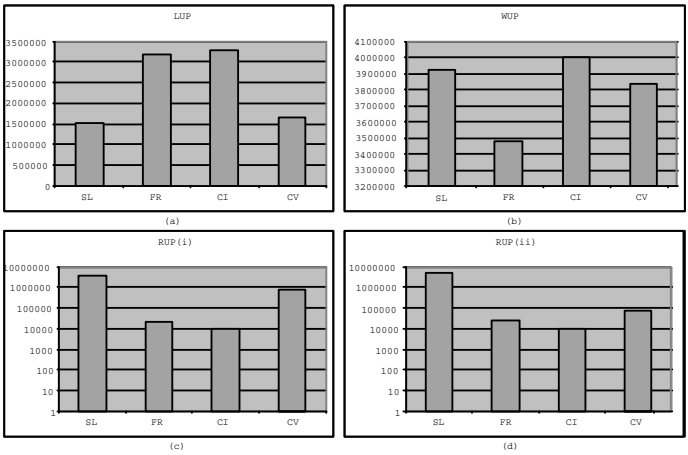

Fig. 3. Histograms showing the cost incurrend by each policy per application pattern.

Figure 3-(a) shows that **SL** is the best policy for the local usage pattern. Store-Locally guarantees low cost for the execution of space operations on local tuples. Figure 3-(b) shows that **FR** produces the lowest cost for the write-many usage pattern. This is because the extra resources spent on replicating tuples, reduce the time required for finding a matching tuple. Figure 3-(c) and (d) show the results for RUP(i) and RUP(ii), respectively. Note that a logarithmic scale is used. In both cases, the **CI** policy produces the lowest cost. This is because caching allows to execute most of the read operations locally. However, the CV policy performs considerably worse than CI policy because the former sends a validation message for each read executed on the local cache.

Figure 4 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* is decreased respect to the one used in the experiment in 3-(c), meaning that a greater number of take operations are executed. 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.

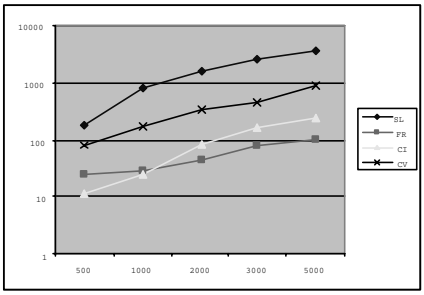

Fig. 4. Results of the simulation for the RUP(i) with different operation lengths.

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 [13] 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 mechanism that can dynamically select the best available distribution strategy.

5 Conclusion and Future Work

In this paper we discussed the use of a flexible architecture for distributed shared data space systems in which the strategy for distributing data amongst nodes can be configured without affecting application functionality. This flexibility enables the tailoring of distribution policies to balance the different extra-functional needs of applications. The separation of extra-functional concerns from application functionality enhances code reuse. Both application code and distribution policies are unit of reuse ready to be deployed in several enviroments.

The need for this flexibility is motivated by a series of experiments. These experiments show that there is no distribution policy that is best for different types of application behaviour.

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.

For future work we are currently optimizing migration strategies needed to dynamically change from one distribution policy to another, and are concentrating on developing accompanying mechanisms. At the same time, we are working on supporting real-time constraints in the same fashion as we are doing with distribution requirements.

References

1. H. Bal et al. "The Distributed ASCI Supercomputer Project." *Oper. Syst. Rev.*, 34(4):76–96, Oct. 2000.
2. H. Bal and M. Kaashoek. "Object Distribution in Orca using Compile-Time and Run-Time Techniques." In *Proc. Eighth OOPSLA*, pp. 162–177, Sept. 1993. Washington, DC.
3. G. Cabri, L. Leonardi, and F. Zambonelli. "Mobile-Agent Cooordination Models for Internet Applications." *IEEE Computer*, 33(2):82–89, Feb. 2000.
4. J. Carreira, J.G. Silva, K. Langendoen,and H. Bal. "Implementing Tuple Space with Threads". *In International Conference on Parallel and DistributedSystems (Euro-PDS97)*, 259–264, Barcelona, Spain, June 1997.
5. J. Carter, J. Bennett, and W. Zwaenepoel. "Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems." *ACM Trans. Comp. Syst.*, 13(3):205–244, Aug. 1995.
6. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. "Making Gnutella-like P2P Systems Scalable." In *Proc. SIGCOMM*, Aug. 2003. ACM Press, New York, NY.
7. E. Cohen, A. Fiat, and H. Kaplan. "Associative Search in Peer-to-Peer Networks: Harnessing Latent Semantics." In *Proc. 22nd INFOCOM Conf.*, Apr. 2003. IEEE Computer Society Press, Los Alamitos, CA.
8. A. Corradi, L. Leonardi, and F. Zambonelli. "Strategies and Protocols for Highly Parallel Linda Servers." *Software – Practice & Experience*, 28(14):1493 – 1517, Dec. 1998.
9. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns and Practice*. Addison-Wesley, Reading, MA, 1999.
10. D. Gelernter. "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.
11. S. Microsystems. *JavaSpaces Service Specification*, Oct. 2000.
12. 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.
13. G. Pierre, M. van Steen, and A. Tanenbaum. "Dynamically Selecting Optimal Distribution Strategies for Web Documents." *IEEE Trans. Comp.*, 51(6):637–651, June 2002.
14. A. Rowstron. "Run-time Systems for Coordination." In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, (eds.), *Coordination of Internet Agents: Models, Technologies and Applications*, pp. 78–96. Springer-Verlag, Berlin, 2001.
15. 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.
16. G. Russello, M. Chaudron, and M. van Steen. "Customizable Data Distribution for Shared Data Spaces." In *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, June 2003.
17. G. Russello, M. Chaudron, and M. van Steen. "GSpace: Tailorable Data Distribution in Shared Data Space System." Technical Report 04/06, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, Jan. 2004.
18. P. Wyckoff et al. "T Spaces." *IBM Systems J.*, 37(3):454–474, Aug. 1998.