Dynamic Adaptation of Data Distribution Policies in a Shared Data Space System

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

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

Abstract. Increasing demands for interconnectivity, adaptivity and flexibility are leading to distributed component-based systems (DCBS) where components may dynamically join and leave a system at run-time.
Our research is aimed at the development of an architecture for middleware for DCBS such that the extra-functional properties of resulting systems can be easily tailored to different requirements. To this end, we proposed an architecture based on the shared data space paradigm. This architecture provides a suite of distribution strategies [16] supporting different application usage patterns.
We showed that using different distribution strategies for different usage patterns improved overall performance [17]. As is the case with other middleware for DCBS, the configuration of the selected distribution policies was fixed before run-time.
Consequently, these systems cannot adapt to changes in usage patterns that may be due to the joining of leaving of the components in the system.
In this paper, we propose a mechanism for the dynamic adaptation of distribution policies to the evolving behaviour of applications. This architecture improves over existing architectures for distributed shared data spaces by providing a mechanism for self-management.
We experimentally demonstrate the benefits that may be gained by dynamic adaptation of distribution policies.

1 Introduction

Software engineering artifacts are witnessing increasing demands for interconnectivity, adaptivity, and flexibility. Existing systems need to exchange information, even in the presence of transient connections; they need to adapt dynamically to different usage contexts; and their structure should support the addition and removal of functionality.

This leads to architectures for distributed component-based systems (DCBSes) where components may dynamically join and leave the system at run-time. The dynamic evolution of the configuration of applications poses new challenges to the balancing between resource usage and performance optimization.

Dynamic composition and reconfigurations of DCBS applications is achieved by an extra software layer, called *middleware*. Since components are subject to reconfigurations it is important that they are loosely coupled. This decoupling can be realized in two dimensions: time and space. Decoupling in time means that components do not need to be active at the same time to communicate. Decoupling in space, also called referential decoupling, means that components need not refer to each other to communicate.

R. Meersman, Z. Tari (Eds.): CoopIS/DOA/ODBASE 2004, LNCS 3291, pp. 1225–1242, 2004.

The *shared data space model* provides both types of decoupling and is thus well suited for DCBS. In the literature, several designs have been proposed for shared data space implementations. Common to those solutions is the use of a single system-wide policy for distributing data. Often, these policies are dictated by constraints that are specific to the application-domain or the type of hardware used. Thus, when those systems are used with applications extraneous to the original domain or on different hardware platforms, their performance may be dramatically affected.

The novelty of our approach resides in exploiting the Separation of Concerns (SoC) concept in a shared data space system. Our design allows to separate extra-functional concerns—in particular data distribution, security, availability—from the basic functionality of an application (for a general overview of our approach see [15]). Since the definition of a concern is not tangled with the definition of other concerns, the design, implementation and verification phases are simplified. Reusability of components is also enhanced since the changes needed for tuning the components to different requirements of the new environment can be localized in a single place in the middleware.

The way in which applications interact with the shared data space determines the resource usage for distributing data across the network. In [17] we experimentally proved that applications may benefit in performance if the middleware provides several policies to deal with data distribution. In the architecture that we propose, we provide an extensible suite of distribution policies.

However, identifying which distribution policy best suits the application behaviour it is often very difficult – if at all possible—before application deployment. To complicate matters, it might be the case that the behaviour of an application changes during its executing time due to component reconfigurations.

The contributions of this paper are as follows: (i) we propose a design that enables the middleware to monitor and subsequently adapt its distribution policy to the actual application behaviour; (ii) as proof of concept we built a prototype that employs our design; and (iii) using the prototype, we conduct a series of experiments that proves the benefits of continuous dynamic adaptation of distribution policies.

The rest of this paper is organized as follows. Section 2 focuses on our approach, providing more insights on its architectural design. The experimental results are presented in Section 3. Section 4 describes research related to the work discussed in this paper. Finally, we conclude and describe future directions of our research in Section 5.

2 GSpace

In this section we briefly introduce the basic concepts of the shared data space model. Subsequently, we explain the design of *GSpace* which is our distributed implementation of a shared data space. In particular, we focus on the mechanisms of GSpace that enable it to dynamically adapt its data distribution policies to application usage patterns.

2.1 Data Space Basic Concepts

The data space concept was introduced in the coordination language Linda [8]. In Linda, applications communicate by inserting and retrieving data through a data space. The unit

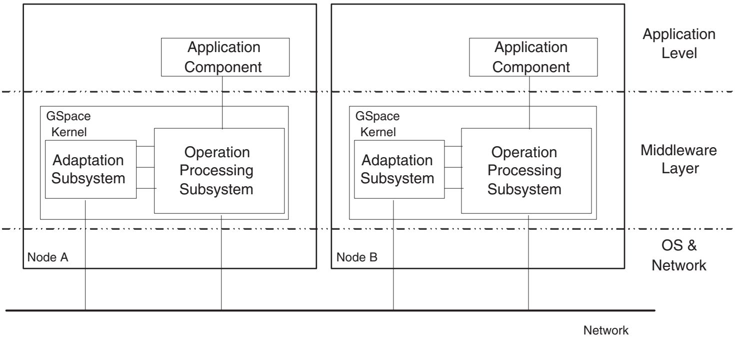

Fig. 1. The deployment of GSpace kernels in two nodes. Each kernel consists of two subsystems: the Operation Processing Subsystem and the Adaptation Subsystem.

of data in the data space is called *tuple*. Tuples are retrieved from the data space by means of *templates*, using an associative method. An application interacts with the data space using three simple operations:put, read and take.

2.2 System Overview

A typical setup consists of several *GSpace kernels* instantiated on several networked nodes. Each kernel provides facilities for storing tuples locally, and for discovering and communicating with other kernels. GSpace kernels collaborate with each other to provide to the application components a unified view of the shared data space. Thus the physical distribution of the shared data space across several nodes is transparent to the application components, preserving the simple coordination model of the shared data space.

In GSpace tuples are typed. Separate distribution policies can be associated with different tuple types. A contribution of this paper is the mechanism for dynamically adapting distribution policies at run-time. This mechanism works along the following lines. During execution the system logs the operations that are executed by the applications. After a certain number of operations has passed, the system evaluates how the available distribution policies would have performed for the most recent log of operations. The system selects the policy that performed best for this log as the policy to use after the evaluation.

In the following sections we will describe the process of dynamically identifying the best distribution policy for a given tuple type.

2.3 Physical Deployment

Figure 1 shows an example of a component-based application distributed across interconnected nodes that uses GSpace. On each node, a GSpace kernel is instantiated. A GSpace kernel consists of two subsystems: the **Operation Processing Subsystem (OPS)** and the **Adaptation Subsystem (AS)**.

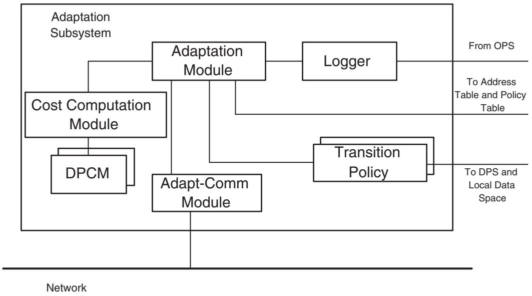

Fig. 2. Internal structure of the Adaptation Subsystem.

The OPS provides the core functionality necessary for a node to participate in a distributed GSpace: handling application component operations; providing mechanisms for communication with kernels on other nodes; and monitoring connectivity of other GSpace nodes that join and leave the system; and maintaining the information about other kernels. Finally, the OPS provides the infrastructure to differentiate distribution strategies per tuple type. The internal structure of the OPS is described in more details [16].

The adaptation subsystem is an optional addition to GSpace that provides the functionality needed for dynamic adaptation of policies. The AS communicates with the co-deployed OPS for obtaining information about the status and actual usage of the system. Periodically, the AS analyzes this information and evaluates the system performance. Based on this information, the AS may decide to change to another distribution policy.

Figure 2 shows the internal structure of an AS. It consists of the following modules:

Logger: The Logger is responsible for logging all the space operations executed on the local kernel. When the OPS receives a request for a space operation from an application component, it informs the Logger about the operation. The Logger keeps track of the number of operations that have been executed for each tuple type. When the number of operations for a particular type reaches a threshold, the logger notifies its local *Adaptation Module*. Each tuple type in the system might be associated with a specific threshold.

Adaptation Module (AM): The AM is the core of the Adaptation Subsystem. The AM is responsible for deciding when the different phases of the *adaptation mechanism* should be started. The code of the AMs on all nodes is identical. However, for each tuple type in the system one AM operates as a *master* [1] and all other AMs operate

[1] Kernels are identified by unique IDs. During the discovery phase, kernels store in an address table addresses and IDs of other kernels. Using this ID the entries in the address table are sorted. Since this ID is unique, all kernels have their own address table sorted in the same order. The tuple types that applications are going to use are listed in the policy descriptor file. A copy of this file is made available on each kernel, thus tuple types are listed in the same order in all nodes. Hence, for the i-th tuple type in the file the master is the node that appears in i-th position in the address table. the first address is designed as master for the first tuple type, and so on until all tuple types eventually get a master.

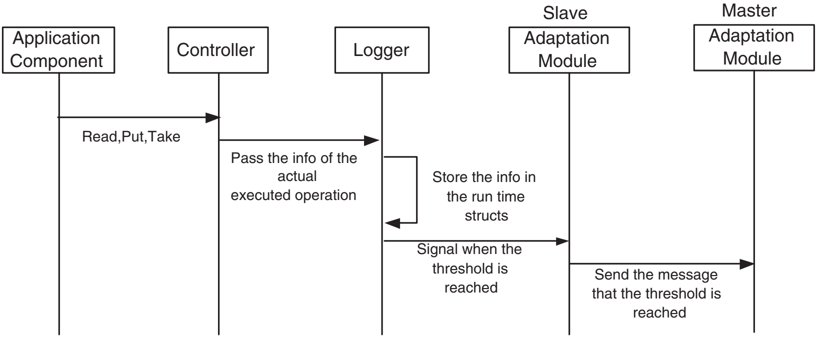
Application Component
Controller
Logger
Slave Adaptation Module
Master Adaptation Module
Read,Put,Take
Pass the info of the actual executed operation
Store the info in the run time structs
Signal when the threshold is reached
Send the message that the threshold is reached

Fig. 3. The MSC of the logging phase.

as *slaves*. The master AM is responsible for the adaptation decisions for a particular tuple type. The slave AMs follow the decisions taken by the master. Because the AMs on all nodes are identical, it is in principle possible for any slave to take over the role of master if the latter leaves the system.

Cost Computation Module (CCM): This module performs a simulation of a log. It obtains the logs from the AM. For all operations in the log it asks the DPCM (described next) to provide the cost of execution of this operation. The CCM aggregates the cost over a complete log. The CCM passes the results of this simulation to the AM.

Distribution Policy Cost Models (DPCM): In order to enable adaptation, a distribution cost policy model must be provided for every distribution policy available in to the GSpace system. The task of the DPCM is to compute the cost incurred by the corresponding distribution policy for a given log of operations. When a run-time extension of the suite of distribution policies available to a GSpace system is required, a DPCM must be provided for every new distribution policy.

Transition Policies: When the distribution policy for a tuple type is adapted, it is possible that tuples of that type are present in the shared data space. We refer to these tuples as *legacy tuples*. A transition policy prescribes how to handle legacy tuples in order for them to be placed at locations where the new distribution policy expects to find them. For each tuple type, the application developer can specify which transition policy to apply.

Adapt-Comm Module (ACM): This module provides communication channels between the ASes on different nodes in the system.

The adaptation mechanism allows GSpace to select the best distribution policy for a given tuple type during run time. These actions can be grouped into three phases.

The first phase is called *logging phase*. During this phase, statistical data is collected about the operations that application components perform for each tuple type. Based on the data collected during this phase, the system will determine the distribution policy that best fits the application distribution requirements for a given tuple type. In Figure 3 a message sequence chart shows the actions executed during this phase. The Controller,

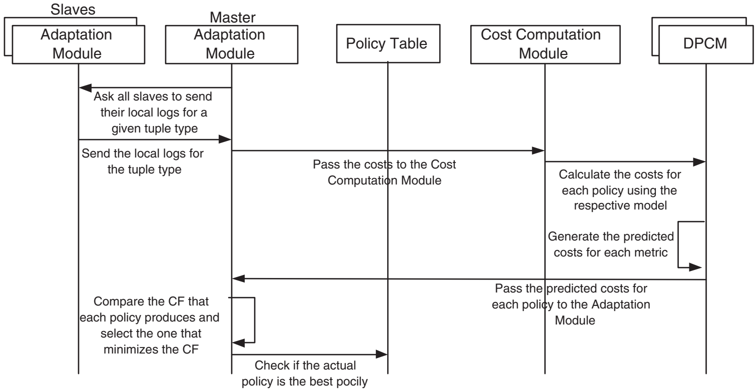
Slaves
Adaptation Module
Master
Adaptation Module
Policy Table
Cost Computation Module
DPCM
Ask all slaves to send their local logs for a given tuple type
Send the local logs for the tuple type
Pass the costs to the Cost Computation Module
Calculate the costs for each policy using the respective model
Generate the predicted costs for each metric
Pass the predicted costs for each policy to the Adaptation Module
Compare the CF that each policy produces and select the one that minimizes the CF
Check if the actual policy is the best pocily

Fig. 4. The MSC of the evaluation phase.

who receives the requests for space operations from the application components, passes the data about the current operation to the Logger. This data contains:

- Operation type: the space operation executed (either a read, take or put)
- Tuple type: the type of the tuple or template passed as argument with the operation
- Location: the address of the kernel where the operation is executed
- Tuple ID: a unique id provided to each tuple that enters the shared data space
- Tuple size: the size of the tuple inserted through a put operation or returned by a read or take operation
- Template size: the size of the template passed as argument of a read or a take operation.
- Timestamp: the time when the operation is executed

Once the number of operations for a tuple type reaches a threshold in one of the nodes, the system starts an *evaluation phase*. The exchange of message in this phase is shown in the message sequence chart in Figure 4.

In this phase the master AM asks all slave AMs to report their local logs for the tuple type. The timestamps in the operation logs are compensated for clock drift. Subsequently, when all logs are gathered by the master, the CCM at that node sorts the aggregated log in chronological order.

For each distribution policy available in the kernel at the time when the evaluation phase is executed, the CCM feeds the logs to the respective DPCM. The DPCM generates the predicted costs that the system would have incurred if that distribution policy had been applied to the tuple type. The CCM collects the costs from the DPCM and passes them to the AM. The AM combines the predicted costs for each policy in a cost function value (more on this in section 2.4). The AM compares this values and selects the *best* policy, that is the one that minimizes this value. The AM checks whether the actual policy associated with the tuple type (it retrieves this information from the policy table) is also the best policy. If this is the case, no further actions are undertaken. Otherwise, the AM starts the *adaptation phase*. If the node in which the threshold was reached is not the

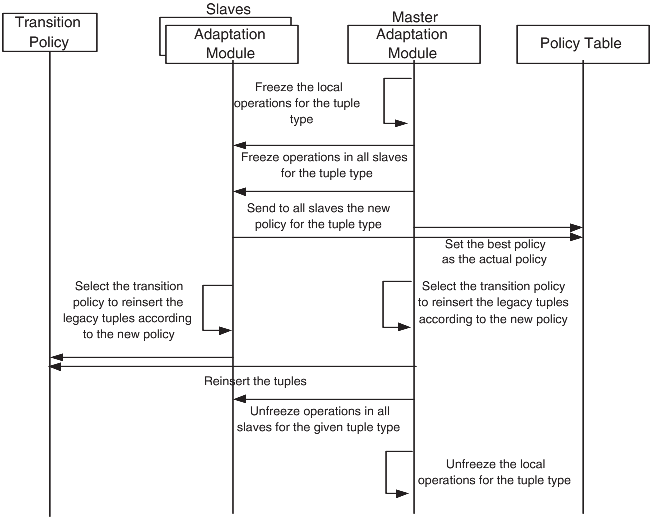
Transition Policy
Slaves
Adaptation Module
Master
Adaptation Module
Policy Table
Freeze the local operations for the tuple type
Freeze operations in all slaves for the tuple type
Send to all slaves the new policy for the tuple type
Set the best policy as the actual policy
Select the transition policy to reinsert the legacy tuples according to the new policy
Select the transition policy to reinsert the legacy tuples according to the new policy
Reinsert the tuples
Unfreeze operations in all slaves for the given tuple type
Unfreeze the local operations for the tuple type

Fig. 5. The MSC of the policy adaptation phase.

node where the master AM is allocated, then the slave AM has to notify its master. After that, the master AM proceeds to the evaluation phase as explained above.

Figure 5 shows the message sequence chart for the adaptation phase. The master AM starts by freezing the operations for the tuple type in the system. This means that during the adaptation phase, the adaptation subsystem will block all incoming requests from application components[2]. The master AM updates its local policy table and then commands to each slave to update their local policy table. The update consists of setting the best distribution policy as the distribution policy for the tuple type.

At this point, each AM retrieves the transition policy associated with the tuple type. The transition policy takes care of possible legacy tuples present in the local data space. Subsequently, the master unblocks the operations for the tuple type and the normal activity is resumed.

Note that all communication between AS modules located on different nodes is mediated through the ACM.

2.4 Distribution Policy Evaluation

Next we discuss the method that we use in the evaluation phase to quantify the efficiency of each distribution.

In a distributed shared data space, such as GSpace, finding the location of a matching tuple might be an expensive task. To minimize tuple access time, tuples should

[2] Notice that this solution of freezing system's activity was taken just for its simplicity and easiness to implement. More complex solutions are available in which the system does not need to block its activity and in the next version of the prototype we are plannig to include one of them

```
readModel(Vector putLogs, Log readLog) {

  while (putLogs.hasMoreElements()) {
    put = putLogs.next();
    if (put.location.equals(readLog.location)) {
      readLatency += Profiler.readLocalLatency(readLog.templateSize);
      return;
    }
  }

  while(addrTable.hasMoreElements()) {
    msgSize = Profiler.sendingPacketSize(readLog.templateSize);
    bandwidthUsage += msgSize;
    readLatency += Profiler.networkLatencyTCP(msgSize);

    addr = addrTable.next();
    while (putLogs.hasMoreElements()) {
      put  = putLogs.next();
      if (addr.equals(put.location)) {
        msgSize = Profiler.sendingPacketSize(put.tupleSize);
        bandwidthUsage += msgSize;
        readLatency += Profiler.networkLatencyTCP(msgSize);
        return;
      }
      msgSize = Profiler.nullReplyPacketSize();
      bandwidthUsage += msgSize;
      readLatency += Profiler.networkLatencyTCP(msgSize);
    }
  }
}
```

Fig. 6. The readModel operation in the SL-DMM

ideally be stored locally on the same node where the application component consumes it. However to provide this, the system has to pay some costs in terms of network access and storage space for moving the tuple to the consumer location. Moreover, some extra communication may be needed for maintaining consistency of the space across all nodes.

Currently, a number of distribution policies is available for GSpace. Each policy strikes a different balance between tuple access time and resource usage. Together with the patterns of tuple accesses by application components these factors determine the performance of a distribution policy.

To compare the performances of distribution policies we follow an approach inspired on [12]. We define a *cost function* as a linear combination of metrics that capture different aspects of the costs incurred by a policy. The cost function combines these costs in an abstract value that quantifies the performance of a distribution policy. We used the following metrics in the cost function: rl and tl represent the cumulative latency for the execution of read and take operations, respectively; bu represents the total network bandwidth usage; and mu represents the memory consumption for storing the tuples in each local data space. 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 burden incurred because of put operations is captured by the latency experienced by read and take operations, which are included in the cost function. The w_i's control the relative contribution of individual cost parameters to the overall cost. The metrics are such that a lower value indicates a better performance.

Periodically, the master AM for a tuple type evaluates the cost function value for each distribution policy. These evaluations are performed by means of simulation using policy models.

Currently in GSpace the following distribution policies are available: Store locally (SL), Full replication (FR), Cache with invalidation (CI), and Cache with verification (CV). Details on these policies can be found in [17]. For each of these policies, we developed the respective DPCM.

The DCPM contains a model of a specific policy. This model predicts the cost for executing a data space operation. This cost is expressed in terms of the variables that occur in the cost function (latency, bandwidth use and memory use). For each DPCM, the CCM iterates through the logs and for each log the CCM invokes the respective operation model.

As an example, Figure 6 shows the pseudo-java code for the readModel operation in the SL-DPCM. The operation takes two parameters: 1) the set of logs for put operations that insert tuples that the read operations can match (this set of logs is maintained by the CCM), and 2) the log for the read operations.

According to the Store-locally policy, the read operation first has to search on the local node for a matching tuple. In lines 3-7, the readModel iterates through the set of logs of put operations searching for a put executed on the same location of the read operation. If such a put has been logged, then the read can return a copy of the matching tuple. In this case, just the latency for accessing the local data space is accounted (line 6). Otherwise, the read operation has to send the request to the other kernels. As for the real operation, the readModel goes through the addresses in the address table in search of the location of a matching tuple (lines 13-21). For each request sent to a node, the readModel accounts the bandwidth usage (line 15). This value is given by the size of the requested message. The message contains the header and the payload, which contains the size of the template, given as argument to the read operation (line 14). Furthermore, the network latency for sending this request using TCP is accounted (line 16). If a put operation has been logged in the current location (meaning that a matching tuple is in this node), then a copy of the tuple is returned. The readModel accounts the bandwidth for sending the reply message with the matching tuple (line 22-23) and the latency for sending the message back to the requester (line 24). If in the current location no put operation have been logged, then a message with a null reply is send back to the requester. Also in this case, the readModel accounts the bandwidth usage (line 27-28) and the network latency (line 29).

Information about the latency for network accesses and for local data space accesses is provided by the profiler module. When GSpace is deployed for the first time in a new environment, the profiler creates these network and data space profiles. For the network profile, the profiler sends a number of packets of different sizes to a remote echo server[3] (for both TCP and UDP packets) and measures the time for the round-trip. This data is used for building a function that for a given packet size returns the latency for sending the packet. For profiling the access to the local space, the profiler executes a number of read and take operations on a local data space with templates of different sizes measuring the time to complete each operation. Also in this case, the data collected is used for building a function that for a given template size returns the access latency. The parameters to build those functions are stored in a file, called **profile.inf**. At booting time, the system tries to load the file. If the file is present, then the environment was already profiled. Otherwise, the profiler of a kernel is chosen to start the profiling phase. Once the necessary data has been collected and processed, the profiler stores the data in the profile.inf file and makes the file available to the other kernels.

2.5 Adapting the Data Space Content

According to the semantics of read and take operations, when a matching tuple is inside the shared data space it should be returned. Since GSpace is a *distributed* shared data space, each distribution policy has its own strategy for searching a matching tuple across the nodes during a read or take. This strategy is influenced by the modality in which tuples are inserted through the put operations of that distribution policy. When the system changes the policy associated with a tuple type as consequence of an adaptation, it is most likely that legacy tuples are still inside the data space. If the searching strategies of old and the new policy are different, then the system cannot guarantee that a matching tuple inside the shared data space is always returned. Depending on the particular application, it could be the case that those tuples could be ignored since new tuples will be soon available. However, this is not the case in general.

For this reason we introduce the transition policy. A transition policy lets the application designer specify the actions to take for the legacy tuples of a given tuple type when an adaptation is executed. If for a given tuple type the transition policy is not specified, a *Default Transition Policy (DTP)* is available. This DTP removes all legacy tuples and reinserts them according to the new policy. This ensures that the space is kept consistent, and reduces the effort of the developer of distribution policies (who does not have to invent a transition policy). The default policy may be costly. To provide the possibility to reduce these cost, GSpace provides the option to define specific policies for making a transition from existing policies to the new policy.

As an example, let us assume that the system has to change policy from SL to FR. The read and take operations in FR search in the local data space for a matching tuple since all tuples are replicated. If a tuple is not found in the local data space then matching should fail. When switching from SL to FR, we need to guarantee that these operations will behave correctly. Therefore, upon switching policy, it is necessary to replicate the

[3] Note that a more precise model can be build by pinging more than one remote server and averaging the results.

legacy tuples to all local data spaces. The DTP simply first removes all legacy tuples and subsequently reinserts them according to, in this case, the FR policy. As a result, tuples are replicated across the entire system.

The execution of a transition policy may involve extra costs. These costs should be taken into account when switching policy. Depending on the number of legacy tuples that needs to be reinserted, the costs of redistribution could be too high compared to the actual gain that the system achieves by adopting the best policy. However, for a long period of execution the best policy may reduce the overall costs to such a level that the extra costs for the redistribution actually pay off. This problem falls in the category of *Online Decision Making with Partial Information* problems, of which the *The Ski Rental Problem* is a classic formalization [9]. Currently, we are working on the adoption of an algorithm to deal with this problem during the adaptation phase.

3 Experiment

In this section we present the results of the experiments. These show that a significant gain in performance can be achieved through the use of dynamic adaptation of data distribution policies. Measurement of the overhead introduced by the adaptation mechanism shows that this overhead is small compared to the gain in performance.

3.1 Experiment Setup

For the execution of the experiments we used the application model described in [17]. Using this model, we are able to simulate several application usage patterns. 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. We generated a set of runs (sequence of operations) in which the pattern in which the application uses the data space changes a number of times. All experiments were executed on 10 nodes of the DAS-2 [2] distributed computer.

Distribution Policy	Actual cost	Predicted cost
SL	49974	**49963**
FR	184977	185815
CI	90779	93132
CV	**49482**	52248

Fig. 7. The predicted and actual values for the cost function for each policy.

3.2 Accuracy of the Model

As we explained in Section 2, the adaptation mechanism uses models for predicting the metrics used for the calculation of the cost function.

For evaluating the accuracy of our models we performed the following experiments. We executed several runs of operations. For each run, we collected both the measured values and the values predicted by the models.

Figure 7 shows a comparison between the actual and predicted cost function values for each policy. These values are to be considered indicative of the complete set of data that we collected during our experiments. The values indicate that the prediction of the model is extremely close to the actual measurements.

In this example, it is worth noticing the result of the selection for the best policy, which occurs during an evaluation phase. Policy SL and policy CV perform quite similar according to both actual and predicted values. Still, the selection of the best policy according to the model is not the same as for the actual measurements. According to the actual measurements, the policy with the lowest cost function value is policy CV. However, the policy with the lowest cost predicted by our model is policy SL. Thus, the adaptation system (that uses predicted costs) would have chosen policy SL instead of policy CV, leading to a sub-optimal solution. However, this sub-optimal choice is not so dramatic since the cost prediction is quite accurate (for this specific case the loss is less than 1%).

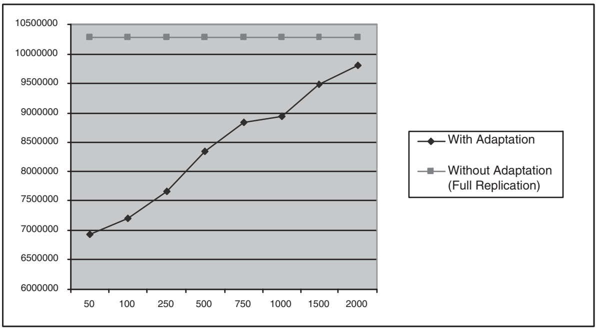

Fig. 8. Accumulated cost function values for different threshold values compared to the cost function of the best static policy.

3.3 Performance and Overhead

To measure the performance gain when adaptation is used, we executed the following experiments. We produced a set of operation runs in which the application model changes behavior during execution. In each run, at least 500 operations are executed according to the same application usage pattern. We refer to this part of a run where the same application usage pattern is used as a *run-phase*.

Firstly, we instantiated GSpace without the adaptation mechanism. For each policy we executed each operation run, collecting the operation logs. At the end of the run, we executed the simulation on the logs for each policy, obtaining the cost function values for each policy. Out of these values, we selected the best cost function values. Subsequently, we executed the same runs but this time GSpace used the adaptation mechanism. We employed different threshold values used for triggering the evaluation phase, expressed as the number of operations before a next evaluation takes place. Every time the evaluation phase was terminated we stored the best cost function value. In the end of the execution these values were summed together, producing an aggregated cost function value. This value represents the total cost incurred during the execution of the run with the adaptation.

Figure 8 shows the graph where the threshold values are placed on the X-axis and on the Y-axis the cost function values. In the graph, the aggregated cost function values for different thresholds are compared with the best cost function value produced during the first phase of the experiments. For all threshold values, the performance of the system with adaptation outperforms the performance of the policy that performs best without adaptation. In particular there is a gain of 30% when the threshold is 50, which reduces to 5% when the threshold is 2000. The graph shows that the smaller the threshold the better the performance. When the threshold value is much smaller then the length of a run-phase, the system can detect more quickly when there is a change in application usage. Hence GSpace can decide sooner to switch to the best policy. Therefore, the total aggregated costs for small threshold values are lower then the costs for larger threshold values.

This is also confirmed by the graph in Figure 9. This graph shows on the Y-axis the percentages of executed adaptations (change of policy) for each threshold value (X-axis). For small threshold values the system has to adapt less often, since once the best policy is determined it has to change only during the next run-phase. As the graph shows, the percentage of adaptations per evaluation increases up to a threshold-value of 1000 and then starts to decrease. This is due to the fact that the simulation is executed with a large number of logs. In such a large number of operations, more run-phases are captured, leading to a random usage pattern behavior. In earlier work [17] we showed that such behavior is best matched by the Full Replication distribution policy. Thus the system just employs such policy for most of the execution. This explanation is also supported by the fact that the performance of the system for larger threshold values is very close to the static case, where the cost function value is obtained by the Full Replication policy (see Figure 8).

The costs incurred by doing adaptation comes from two factors:

1. the costs in performing evaluation. This leads to additional network traffic for collecting logs and to additional computation time for simulating the policies for the logs.
2. the costs of making a transition from one policy to a new one. These costs depend on the particular transition policy.

For the default transition policy (DTP) we performed a number of measurements. Figure 10 shows the time needed for evaluation and transition to a next policy for increasing thresholds.

As expected, the threshold value does not influence the total evaluation time. This is because the number of evaluated logs (directly proportional to the threshold value)

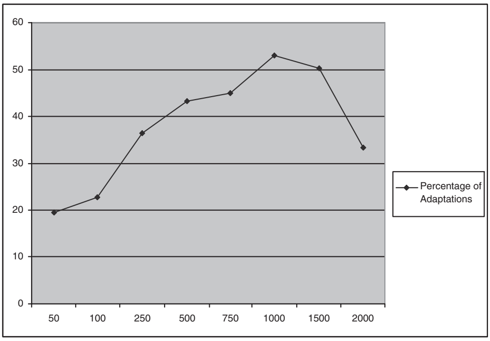

Fig. 9. The percentages of adaptation phases compared to the number of evaluation phases for different threshold values.

is inversely proportional to the frequency of evaluation. Instead, the total time spent on changing policy using the DTP decreases when the threshold value increases.

Notice that during the evaluation phase on the master node, the system is still able to serve the application requests. Only during the transition phase the system does not accept requests until termination of this phase in all nodes. Thus, choosing small threshold values has the advantage of increasing system performance, but increases the cumulative time waiting for transitions to complete. Finally, the last row in the table shows the percentage of the total time spent in evaluating and making transition with respect to the total execution time of the run. We argue that this extra 14% overhead due to the adaptation mechanism is worthwhile to pay compared to the gain in performance that the system achieves.

3.4 Choosing the Appropiate Threshold

The basic assumption behind this research is the possibility to predict the future behaviour of an application through the analysis of its near-past behaviour. Research described in [12] proves that this assumption holds for Web applications. Although we have achieved promising results in our own work, we need to use real-world traces to validate our basic assumption.

Related to the prediction of application behaviour is the selection of an appropriate threshold value for a given application. If the threshold is too small then the system

	50	**100**	**250**	**500**	**750**	**1000**	**1500**	**2000**
Total evaluation time	201810	192820	192572	192851	193389	133408	195261	193584
Total transition time	3735	3092	1626	669	650	1131	21	205
% of total exec. time	15%	14%	14%	14%	14%	9%	14%	14%

Fig. 10. The evaluation time, transition time and their percentage respect to the total execution time. Time is in milliseconds.

will not have enough information for a correct prediction. In addition, adaptation may introduce a high overhead to the entire system. In contrast, a larger threshold reduces the frequency of evaluation at the cost of lower performance due to late identification of a better distribution policy.

The correct threshold should be such that the system gain introduced through adaptation balances the overhead of the adaptation itself. For example, consider Figure 8. For the given run the choice of the threshold can make a difference between 30% and 5% performance improvement in comparison to a static deployment. However, the adaptation itself introduce a 14% of overhead. To be effective, the performance improvements should outweigh the overhead costs. In our case, the threshold value should be selected between 50 (with an effective gain of 16%) up to 500 (with an effective gain of 5%).

A general solution to selecting a threshold could be to introduce a warm start period in which an application deploys a default distribution policy, and logs are collected to capture its behavior in an attempt to identify run phases. Using the techniques described previously, we can then find a reasonable threshold value from which point on adaptive distribution is deployed. Note that a similar continuous external analysis of an application's usage patterns would allow us to even dynamically adjust the threshold value. We plan to investigate these matters in our future research.

4 Related Work

4.1 Shared Data Spaces

Several different approaches for realizing shared data space systems have been proposed. The most common approach is to build a centralized data space in which all tuples are stored at a single node. The main advantage of such an approach is its simplicity. Examples of this approach include JavaSpaces [7] and TSpaces [21]. 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 system-wide 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 across nodes and across time. 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.

The shared data space has been used also in highly dynamic environments, such as in home networks. Those environments are characterized by devices that unpredictably join and leave the network. An approach for coping with such dynamic environments is to dynamically distribute the data space. A system that follows this approach is Lime [11]. In Lime, the shared data space is divided into several transient data spaces that are located on different nodes that form a network. The content of the shared data space changes dynamically upon connection and disconnection of devices. Tuples generated on a device are stored in the local transient data space. When a device connects to the network, the content of its local data space is made available to the entire shared data space. If the device is disconnected the content of its local data space is no longer available unless special actions are taken upon departure time.

A somewhat similar yet simpler approach is followed in SPREAD [6], which is a shared data space system tailored towards mobile and embedded computing. SPREAD follows a store-locally strategy and take operations can be performed only by the node that stored the tuple. However, read operations can be carried out by any node that is in range of a tuple.

Fully replicated data spaces have also been developed, as in [5]. 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.

Much research has been done on developing distributed shared data space systems that are fault tolerant. Notable work in this area is FT-Linda [1] and LiPS [18]. FTLinda provides a data space that guarantees persistence of tuples in the presence of node failures. It also guarantees atomic execution of a set of data space operations. LiPS provides mechanisms that allows the system to recover from data loss and process failures.

Eilean [19,4] is a distributed shared data space system that explicitly addresses scalability issues. Together with GSpace, Eilean is the only example of a shared data space system that provides multiple tuple distribution policies. Like GSpace, Eilean is able to differentiate distribution policies on a per-tuple-type basis.

In contrast to GSpace, the tuple-distribution policy association in Eilean can only be statically defined as part of the application. The programmer uses his knowledge of the application access pattern to define the association. In previous work [17] we demonstrated that this static association is not enough for providing an efficient distribution of tuples. With the adaptation mechanism described in this paper, GSpace is able to monitor the application behavior and dynamically adapt the distribution policy for each tuple type. Another difference between Eilean and GSpace is that in GSpace the set of distribution policies can be extended and new distribution policies can be downloaded in the system even during execution.

4.2 Adaptive Shared-Object Systems

We are not aware of any shared data space systems that are able to dynamically adapt to the application needs. Systems with this type of adaptive capability do exist in the the domain of shared objects.

One of the first systems that adopted a form of automatic differentiation was Orca [3]. This system provides support for physically distributed objects. An object can be in one of two forms: fully replicated or as single copy. By monitoring the read-write ratios, the run-time system can dynamically switch an object between the two forms.

Further differentiation is offered by fragmented objects [10], and Globe's distributed shared objects [20]. Both systems separate functionality from distribution aspects by subdividing objects into at least two subobjects. One subobject captures functional behavior and can be replicated across multiple nodes. Each copy of such a subobject is accompanied by a subobject that dictates when and where invocations can take place, similar to the role of distribution manager in GSpace (as part of the OPS [16]). The main difference between GSpace and these two systems, is GSpace's more evolved approach towards run-time adaptations. With fragmented objects, distribution strategies were more or less static; in Globe, dynamic adaptation has only been partly implemented.

For sake of completeness, we also mention the support for differentiating distribution in distributed shared memory systems, notably Munin and later Treadmarks (for an overview, see [13]). In these cases, distribution strategies have mostly been static and needed to be fixed at compile time.

5 Conclusion and Future Research Direction

In this paper we presented a middleware system that has a mechanism for self-optimization of data distribution policies. The middleware allows application developers to specify distribution policies per tuple type. A basic suite of policies is available and can be extended, during execution, with new policies. Additionally, and this is a unique feature among distributed shared data space systems, the middleware adjusts the distribution policy used for tuple types to the usage pattern of applications during execution.

We demonstrated by means of experiments, that a drastic gain in performance can be obtained when the middleware adapts the distribution policy to the actual needs of applications.

Our adaptation mechanism is based on models to predict cost values for system parameters (latency, bandwidth use and memory use). We provided a mechanism by which these models are calibrated automatically. The comparison between the predicted values and the measure values show that a reasonable accuracy of these models is obtained. The automatic calibration alleviates the burden on the system designer by avoiding the need for obtaining detailed measurements about the environment in which the application will be deployed.

As future research, we are investigating possible cost-optimization of the adaptation phase. Furthermore, we are currently extending the extra-functional concerns that GSpace is able to handle, such as real time constraints and fault-tolerant properties.

References

1. D. Bakken and R. Schlichting "Supporting Fault-Tolerant Parallel Programming in Linda." *IEEE Trans. on Prallel and Distributed Systems*, 6(3):287–302, March 1995.
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. Eighth OOPSLA*, pp. 162–177, Sept. 1993. Washington, DC.
4. J. Carriera. *Researching the Tuple Space Paradigm in Parallel Programming*. PhD thesis, University of Coimbra, 1998.
5. A. Corradi, L. Leonardi, and F. Zambonelli. "Strategies and Protocols for Highly Parallel Linda Servers." *Software – Practice & Experience*, 28(14):1493 – 1517, Dec. 1998.
6. P. Couderc and M. Banatre. "Ambient Computing Applications: An Experience with the SPREAD Approach." In *Proc. 36th Hawaii Int'l Conf. System Science*. IEEE, Jan. 2003.
7. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns and Practice*. Addison-Wesley, Reading, MA, 1999.

8. D. Gelernter. "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.
9. S. Irani and A. Karlin. "Online Computation." From *Approximations for NP-Hard Problems*. ed. Dorit Hochbaum, PWS Publishing Co,1995.
10. M. Makpangou, Y. Gourhant, J.-P. le Narzul, and M. Shapiro. "Fragmented Objects for Distributed Abstractions." In T. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 170–186. IEEE Computer Society Press, Los Alamitos, CA, 1994.
11. 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.
12. 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.
13. J. Protic, M. Tomasevic, and V. Milutinovic. *Distributed Shared Memory, Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1998.
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. G. Russello, M. Chaudron, and M. van Steen. "Separating Distribution Policies in a Shared Data Space System." Internal Report IR-497, Department of Computer Science, Vrije Universiteit of Amsterdam, May 2002.
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 (PDPTA 2003)*, June 2003.
17. G. Russello, M. Chaudron, and M. van Steen. "Exploiting Differentiated Tuple Distribution in Shared Data Space." *Proc. Int'l. Conf. on Parallel and Distributed Computing (Euro-Par)*, Pisa, Italy, August 2004.
18. T. Setz and T. Liefke "The LiPS Runtime System based on Fault-Tolerant Tuple Space Machines." Technical Report TI-6/97, Darmstadt University, 1997
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. M. van Steen, P. Homburg, and A. Tanenbaum. "Globe: A Wide-Area Distributed System." *IEEE Concurrency*, vol. 7, no 1, pp. 70–78, Jan. 1999.
21. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. "T Spaces." *IBM Systems J.*, 37(3):454–474, Aug. 1998.