Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 246–262
Science of Computer Programming
www.elsevier.com/locate/scico

An experimental evaluation of self-managing availability in shared data spaces

Giovanni Russello[a,*], Michel R.V. Chaudron[a], Maarten van Steen[b], Ibrahim Bokharouss[a]

[a] *Eindhoven University of Technology, The Netherlands*
[b] *Vrije Universiteit Amsterdam, The Netherlands*

Received 15 September 2005; received in revised form 15 March 2006; accepted 15 June 2006
Available online 7 November 2006

Abstract

With its decoupling of processes in space and time, the shared data space model has proven to be a well-suited solution for developing distributed component-based systems. However, as in many distributed applications, functional and extra-functional aspects are still interwoven in components. In this paper, we address how shared data spaces can support the separation of concerns. In particular, we present a solution that allows developers to merely specify availability requirements for data tuples, while the underlying middleware evaluates various distribution and replication policies in order to select the one that meets these requirements best. Moreover, in our approach, the middleware continuously monitors the behavior of application and system components, and switches to different policies if this would lead to better results. We describe our approach, along with the design of a prototype implementation and its quantitative evaluation.

Keywords: Shared data spaces; Fault tolerance; Availability; Self-management; Distributed systems

1. Introduction

The shared data space model has proven to be a useful abstraction for developing distributed applications. Notably its support for decoupling processes in space and time makes it attractive for distributed systems that require dynamic configuration of applications by adjusting (the set of) components at runtime.

In general, architectural quality properties of a system, such as performance, availability and security, are largely determined by the strategies that the components of the system use for exchanging and storing data. In many distributed applications, the extra-functional quality properties are interwoven with functionality inside components.

The coordination paradigm advocates the principle of separating computation concerns from coordination concerns. Applying this principle to the architectural design suggest that a system can be organized as a collection of components that contain application functionality together with a layer that coordinates these components. This design can be seamlessly implemented via a shared data space system. The shared data space enables components to exchange and store data while abstracting from specific strategies through which this may be realized. Through

* Corresponding address: Department of Computing, Imperial College London, South Kensington Campus, 180 Queen's Gate, London SW7 2RH, United Kingdom. Tel.: +44 (0)20 759 48385; fax: +44 (0)20 7581 8024.
E-mail address: russello@doc.ic.ac.uk (G. Russello).

0167-6423/$ - see front matter © 2006 Published by Elsevier B.V.
doi:10.1016/j.scico.2006.06.007

this abstraction it becomes possible to use different strategies for data exchange and storage through which quality objectives can be managed.

From an implementation perspective, the shared data space implements a middleware layer underneath application components. In particular, we believe that this middleware should provide the mechanisms for specifying and enforcing extra-functional concerns. For example, if availability is required, the middleware should ideally offer mechanisms that allow a selection from different replication policies that can be enforced at runtime. If necessary, new policies can be developed and deployed during execution.

Somewhat surprisingly, research on shared data spaces has been largely ignoring the support for this separation of concerns. A plethora of solutions have been proposed to distribute data items, without giving the application developer a choice on *how*, *where*, and *when* data should be distributed or replicated. To solve this problem, we have proposed an extension of the shared data space model with a mechanism for separating the distribution and replication of data items from their strict functional usage by application components. Moreover, by monitoring the behavior of application components, we are able to dynamically adapt data distribution to the needs of an application. We have thus effectively created a closed feedback-control system, now often coined as a self-managing or autonomic system.

Previously, we have considered adaptation for performance, focusing on metrics such as application perceived latency and consumed network bandwidth. For this paper, we concentrate on data availability. Assuming that nodes may unpredictably fail, particular care has to be taken for shared data items to remain available to components.

A well-known solution to this problem is data replication. By replicating data on several nodes, the system can statistically guarantee that a data item is available even if the node where the item was inserted is no longer available. However, replicating for availability may conflict with replicating for performance. For example, high performance requirements may dictate that only weak data consistency can be supported, whereas high availability requires updating all replicas simultaneously.

Such tradeoffs generally require application-specific solutions. However, instead of imposing a single solution, we propose a framework that offers to the application developer a suite of availability policies. Each policy incurs costs with respect to performance, availability, consistency, etc. In our approach, a developer is offered a simple means to weigh these different costs such that the system can automatically choose the policy that meets the various (and often conflicting) objectives best. Moreover, through continuous monitoring of the environment the system can dynamically and automatically switch to another policy if it turns out that this would reduce overall costs.

We make the following contributions. First, we provide a simple mechanism that allows for separating concerns regarding functionality and availability in shared data space systems. Second, we demonstrate how possibly conflicting objectives can be dealt with in these systems, such that the selection of a best policy can be done dynamically and in a fully automated fashion. Third, we show that the input needed from an application developer to support these optimal adaptations can be kept to a minimum, allowing the developer to concentrate on the design and implementation of functionality.

This paper is organized as follows. In Section 2 we present our proof-of-concept called GSpace and its (optional) adaptation mechanism. To prove the soundness of our framework we conducted some experiments, of which the outcomes are discussed in Section 3. Section 4 focuses on related work. We conclude in Section 5 and give directions for future research.

2. GSpace

In this section, we first provide some background information on the shared data space model. Thereafter, we concentrate on our implementation of a shared data space, called *GSpace*. We describe the internal modules that compose GSpace.

2.1. Architectural design

The data space concept was introduced in the coordination language Linda [7]. Applications can insert data into, and read or delete data from the data space using the operations: put, read and take. The unit of data in the data space is called *tuple*. Tuples are selected in an associative manner by matching against *templates*. For example, a tuple can be specified as ⟨ 2, 4.5, “a” ⟩. A process querying the data space can now provide a template such as ⟨ 2, ?, “a” ⟩, which would match the example tuple, and which can then be returned as a result to the reader. Multiple instances of the same tuple item can coexist.

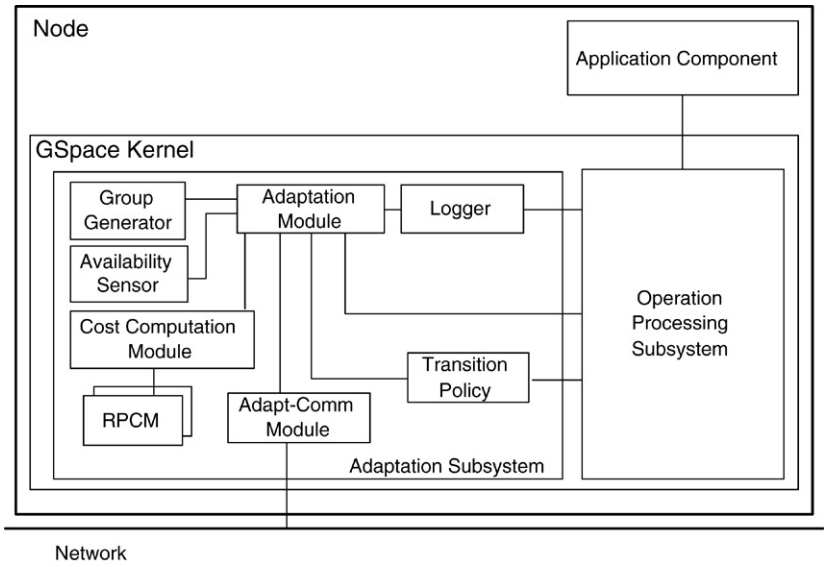

Fig. 1. Internal structure of a GSpace kernel deployed on a node.

GSpace is an implementation of a distributed shared data space. A typical set-up of GSpace consists of several *GSpace kernels* instantiated on several networked nodes. Each kernel provides facilities for storing tuples 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 its simple coordination model. In GSpace tuples are typed. This allows the system to associate different availability policies with different tuple types.

Fig. 1 shows a GSpace kernel deployed on a networked node. A GSpace kernel consists of two subsystems: the *Operation Processing Subsystem (OPS)* and the *Adaptation Subsystem (AS)*.

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; 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 [19].

The adaptation subsystem is optional to GSpace. It 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. In particular the *Logger* is responsible for logging all the space operations executed on the local kernel. When the number of operations for a particular type reaches a threshold, the logger notifies its local *Adaptation Module (AM)*. The AM is the core of each AS. The AM coordinates the different phases of the *adaptation mechanism*. The code of the AMs on all nodes is identical. However, for each tuple type in the system one AM operates as a *master* and all the others as *slaves*. The master AM takes decisions concerning which replication policy should be applied to a tuple type. The slaves AM follow the master's decisions. The *Cost Computation Module (CCM)* and *Replication Policy Cost Models (RPCM)* are responsible for computing the costs incurred by the replication policies for a given set of operation logs. The *Transition Policy* prescribes how to handle legacy tuples in order for them to be placed at locations where the new replication policy expects to find them. The *Adapt-Comm Module (ACM)* provides communication channels between the ASes on different nodes in the system.

The new modules that deal with availability are the following:

Availability Sensor: This module is responsible for measuring the availability of the node on which it is deployed.

Group Generator: Generating groups of nodes is the task of this module. Once the availability values for all the nodes have been collected the master AM passes this information to its local Group Generator. The Group Generator will aggregate nodes following some given strategy. For instance, in the experiments that we discuss in Section 3 the Group Generator selects the best three nodes in term of availability and it provides the group to replication policies.

In the following section we describe in more detail how the different modules in the AS contribute to the mechanism that allows GSpace to select the replication policy that best suits the application behavior.

2.2. Autonomic behavior in GSpace

In a distributed system such as GSpace, nodes may fail. As a result, part of the shared data space may not be reachable. A common solution to this problem is the use of replication. By replicating tuples across several nodes we increase the probability of accessing a tuple even if some nodes are down. However, replication requires consumption of extra resources, such as extra memory and bandwidth for storing tuple replicas and keeping replicas consistent.

Instead of proposing a one-size-fits-all solution, our approach sets flexibility as its primary goal. We included in GSpace a suite of availability policies each with its own tradeoff between provided availability, resource consumption, and performance. The problem we are tackling is finding the replication policy that (a) minimizes resource consumption while (b) fulfilling the availability requirements. These conditions are generally in conflict with each other. As we will show, our simple mechanism is able to deal with such conflicting situations in a fully automated fashion.

As the environment's conditions change over time, a static assignment of replication policy to tuple type could eventually fail to provide the required performance of the system. As a solution to this issue, we monitor the environment. Application patterns are detected by logging each data space operation. Moreover, to guarantee that availability requirements are fulfilled, sensors are placed in each node to measure node availability in real time. By combining these data, our mechanism can automatically detect when to switch to another availability policy if it turns out that availability is at risk, or when resource consumption can be improved.

We identify three phases in our mechanism, that we briefly explain in turn. A more detailed description can be found in [22].

- Monitoring phase.
- Evaluation phase.
- Adaptation phase.

2.2.1. Monitoring phase

During the first phase GSpace collects statistical data regarding its environment. This data consists of information about the availability of nodes and the usage profile of application components.

For collecting node availability, the GSpace kernel is instrumented with a sensor that monitors the availability of the node where it is running. For computing the availability of a node, the sensor installed in the node collects starting and ending times of a failure. When the system is started for the first time, the sensor writes into a file the starting time of the system. Periodically (every 100 ms), the sensor is activated and writes timestamps into the same file. Writes to disk are synchronous. Actually, a timestamp is just the time at which the sensor is active. After a node experiences a failure, at rebooting time the sensor detects that the system was down (since the timestamp file is stored persistently). The starting time of a failure is then considered as the time at which the last timestamp was written whereas the time at which the system is up again is considered as the end-of-failure time. GSpace simply calculates the down time as the difference between the new starting time and the time of the last recorded timestamp.

For collecting data on the application behavior, we employ the same method as described in our previous work [21]. Each data space operation that application components execute is logged and stored per tuple type. Fig. 2 shows the message sequence chart during the operation logging. When the number of executed operations on a node reaches a given threshold the system starts the next phase.

2.2.2. Evaluation phase

The evaluation phase consists of collecting data from all nodes and comparing the cost of different replication policies.

Fig. 3 shows the message sequence chart of the evaluation phase. The master AM requests all slave AMs to send their local data (logs and node availability). This data is combined and the costs for each policy are calculated by means of simulation.

For capturing the performance of the different distribution policies we use a *cost function*. Our cost function is a linear combination of various parameters. The values of these parameters are combined in an abstract value that

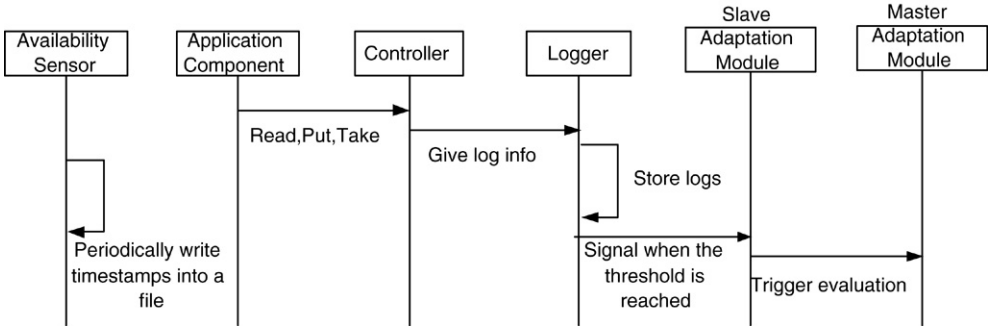

Fig. 2. The MSC of operation logging.

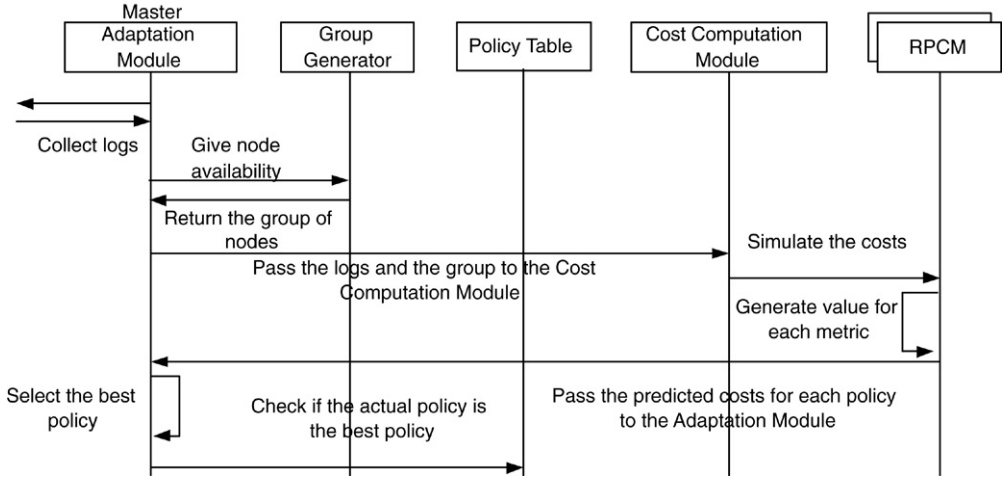

Fig. 3. The MSC of the evaluation phase.

quantifies the tradeoff between performance versus resource usage for a given replication policy. The parameters are defined in such a way that a lower value indicates lower costs (and thus better behavior). The replication policy that leads to the lowest costs is considered the best policy for the application.

In this work, we apply the same method as described in [21] but with the focus on data availability. Therefore, we use a different cost function: *bu* represents the bandwidth usage; *mu* represents the accumulative memory usage; and *da* represents the *derived availability*. The latter is calculated as follows:

$$da(p) = \begin{cases} 100 - availability(p) & \text{if } avail(p) \geq required\_avail, \\ MaxValue & \text{if } avail(p) < required\_avail \end{cases}$$

where $avail(p)$ is the availability for policy p and *required_avail* is the required availability that the application developer specifies. If the availability provided by a replication policy p does not satisfy the user's requirements then the value for *da* is set to *MaxValue* so that the calculated costs will become very high and the system will automatically reject this policy. The cost function is defined as follows:

$$CF_p = w_1 * bu(p) + w_2 * mu(p) + w_3 * da(p). \tag{1}$$

The weights w_i tune the relative contribution of each parameter to the overall cost.

Once the costs are calculated for each replication policy, they are passed to the AM that selects the best replication policy. The AM checks whether the current policy is still the best one. If this is the case, no further actions are undertaken. Otherwise, the AM starts the phase described next.

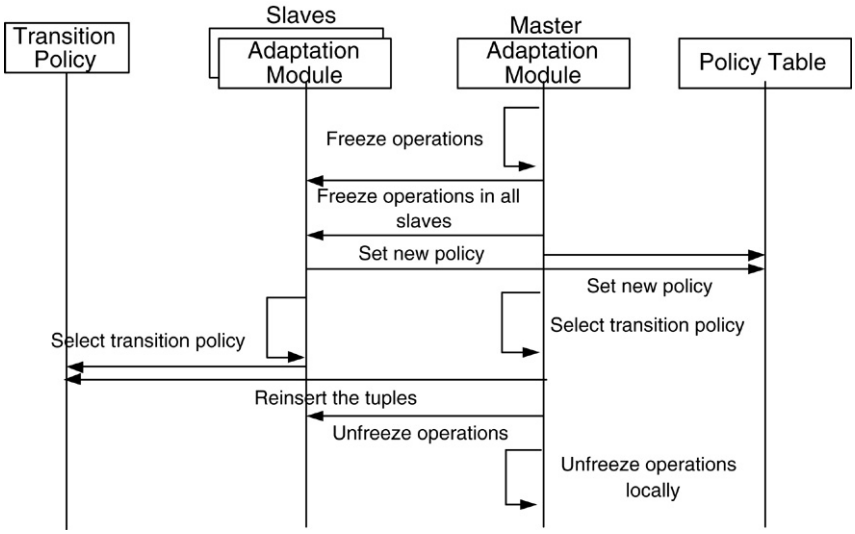

Fig. 4. The MSC of the policy adaptation phase.

2.2.3. Adaptation phase

In this phase the system switches replication policy and adapts the data space content. In Fig. 4 the actions executed during this phase are presented in a message sequence chart. The master AM freezes application operations for the given tuple type in all nodes. Afterward, each kernel updates its own data structure and redistributes the tuples still in the space according to the new replication policy. When this transition period ends the master AM resumes the operations in all nodes.

3. Implementation and experiments

This section describes the experiments that we performed using a full-scale GSpace set-up deployed together with a benchmarking application. It should be noted that the results described in this section are to be considered as an evaluation of our *proof-of-concept* prototype.

Our previous experiments focused on distributed systems in which application components dynamically join and leave a system during execution (but in which the nodes were always available). In [20] we showed that there is no single distribution policy that is best for this dynamic type of application behavior. Furthermore, in [21] we showed that dynamically adapting the distribution policy outperforms any static policy.

In this paper we do not only consider changes in the application behavior, but also in the underlying hardware infrastructure. In particular, we consider that the availability characteristic of nodes in the network may change due to failures.

In [22] we simulated a GSpace deployment over 10 nodes in a LAN environment showing the impact of changing infrastructure on sustaining a level of availability: without adaptation, no single static policy is able to sustain a given level of availability. Moreover, we showed that the dynamic adaptation of the policy provides a better level of availability in the case of changing infrastructure. Furthermore, we showed that the adaptation mechanism can handle situations where both the infrastructure as well as the application behavior change dynamically.

For this paper, we fully implemented all modules to enable a GSpace kernel to be availability aware. The experiments were executed on a cluster of nodes connected by a gigabit LAN. We deployed application components and GSpace kernels in several nodes (ranging from 3 up to 17) of the cluster. While application components were executing we performed controlled failures on the nodes. The goals of this set of experiments are:

- Firstly, we want to corroborate the results obtained via simulation in [22] using an actual implementation. In particular, we want to show that the system is able to dynamically adapt to the application behavior changes as much as to the infrastructure changes.

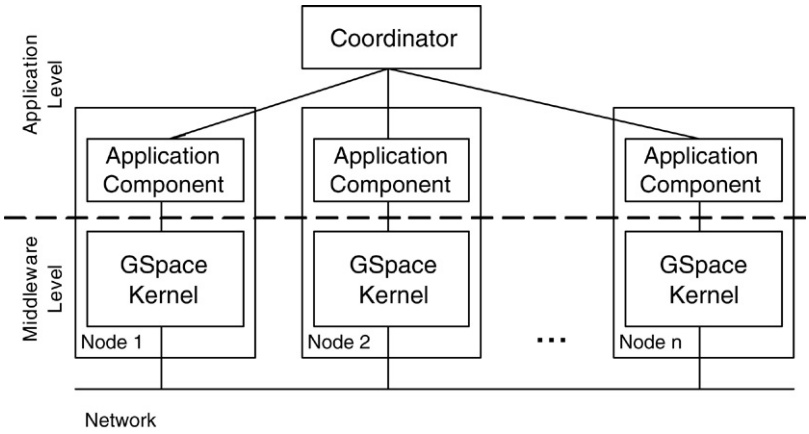

Fig. 5. The deployment of the benchmark application and GSpace kernels in the experiment set-up.

- Secondly, we want to measure timing properties of the Adaptation Subsystem under several circumstances. In particular, we want to measure the tradeoff between the overhead and the gain that the system obtains when adaptation is enabled.
- Moreover, we are interested in how well the system scales in terms of frequency of evaluation and number of nodes.

In the following, we first describe the set-up used for the experiments.

3.1. Set-up of the experiments

We run the experiments in the Sandpit cluster hosted by the Eindhoven University of Technology. The nodes used for the execution of the experiments are 3.06 GHz Pentium4 with 2 GByte memory and are connected via a 1 Gigabit LAN.

For benchmarking our system, we used an application benchmark that we used in earlier work [20]. Fig. 5 shows the deployment of such an application together with GSpace kernels. In each node, an application component is deployed together with a GSpace kernel.

All application components insert in the space the same tuple instance. The template that is used for the read and take operations is the same and matches all the tuples that are in the system.

The core unit of the application is the **Coordinator**. The main functions of the Coordinator are the following:

- Generate sequences of operations, which we refer to as *runs*, according to predefined application behavior.
- Generate sequences of failures according to a specific pattern (per node).
- The ability of replaying exactly the same run of operation.

The Coordinator avoids the introduction of randomized anomalies in our experiments. The Coordinator orchestrates the execution of a run by dispatching operations to components according to a given sequence. The Coordinator is connected directly with each application component and does not interact with any GSpace kernels. Thus, Coordinator–Component interaction does affect any decision taken by the Adaptation Subsystem.

For this set of experiments we also want to control availability of nodes over time. More precisely, we want to control when and for how long a failure should occur on each node.[1] For this reason, we extended our benchmark application by adding a set of special operations for starting and ending a kernel's failure.

Although GSpace is integrated with a mechanism able to deal with failures during a blocking operation, we do not allow that failures can occur while a blocking operation is being performed. Allowing such failures would introduce more complexity in the experiments, making the evaluation of the system's performance unnecessarily more complicated.

[1] A failure means that a kernel does not respond to a request.

3.2. Replication policies

The set of replication policies for GSpace is extensible. For the experiments in this paper, we use the following set of replication policies:

- **Full Replication (FlR)**. Tuples are replicate in all available nodes.
- **n-Fixed Replication (n-FxR)**. Tuples are replicated to a group of n nodes.
- **Dynamic Consumer Replication (DCR)**. This policy replicates tuples to all nodes that host an application component that is a consumer of this type of tuple. In case the availability of the group does not meet the requirements nodes are added according to the Availability Manager.
- **Dynamic Producer Replication (DPR)**. This policy replicates tuples to all nodes that host an application component that is a produced of this type of tuple. Also here, new nodes can be added if the availability requirements are not met.

For maintaining tuple replicas consistent, the availability policies collaborate using atomic multicasting supported by the Group Communication Protocol (GCP) described in [10]. The nodes on which tuples are replicated are joined in a group where operations are atomically broadcasted. This is especially appealing for the execution of put and take operations because with one single message an insertion or removal operation can be executed in all nodes of the group.

The GCP takes care of consistency issues that could arise from the failure of some of the nodes in the group. When a node recovers from a crash, its kernel can rejoin the group. During the rejoining phase, the GCP ensures that the kernel's status is updated with the rest of the nodes in the group. Additionally, when a node crashes during the execution of a blocking operation, such as a read or a take, the GCP guarantees that the correct semantics of the operation is respected.

The availability of a given replication policy is determined by the availability of the group of nodes where tuples are replicated to. In particular, a group of nodes is considered available if at least one node of the group is available. Then, the group availability, GA, equals 1 minus the probability that all nodes within the group fail:

$$GA = 1 - P_{all\_nodes\_down}. \tag{2}$$

We assume that failures of nodes are independent. Then the probability that all nodes fail is equal to the product of the probabilities of failure f_i of the individual nodes:

$$P_{all\_nodes\_down} = \prod_{i=1}^{n} f_i. \tag{3}$$

3.3. Dealing with a changing environment

In this section, we discuss the results obtained when both application behavior and node availability changes at runtime.

We set up an experiment that is similar to the one described in [22]. The experiment consists of changing the application component behavior during execution according to the following phases:

- Phase 1 (cycles 0–32): all application components are consumers and producers;
- Phase 2 (cycle 32–64): only the application components deployed on nodes n_9 and n_{10} act as consumers, all the other components act as producers;
- Phase 3 (cycle 64–95): only application components on nodes n_9 and n_{10} act as producers, the other components act as consumers.

Moreover, the nodes n_9 and n_{10} sustain most of the failures, therefore the group formed by these two nodes is not always able to sustain the required level of availability, which is fixed to 70%.

Fig. 6 shows the graphs of cost function values obtained during each evaluation phase. In this experiment all parameters of the cost function have equal weight. Graph (a) was obtained by execution of the real system and graph (b) was obtained by simulation in [22]. In both graphs, the cost functions exhibit a similar trend, although their values are different.

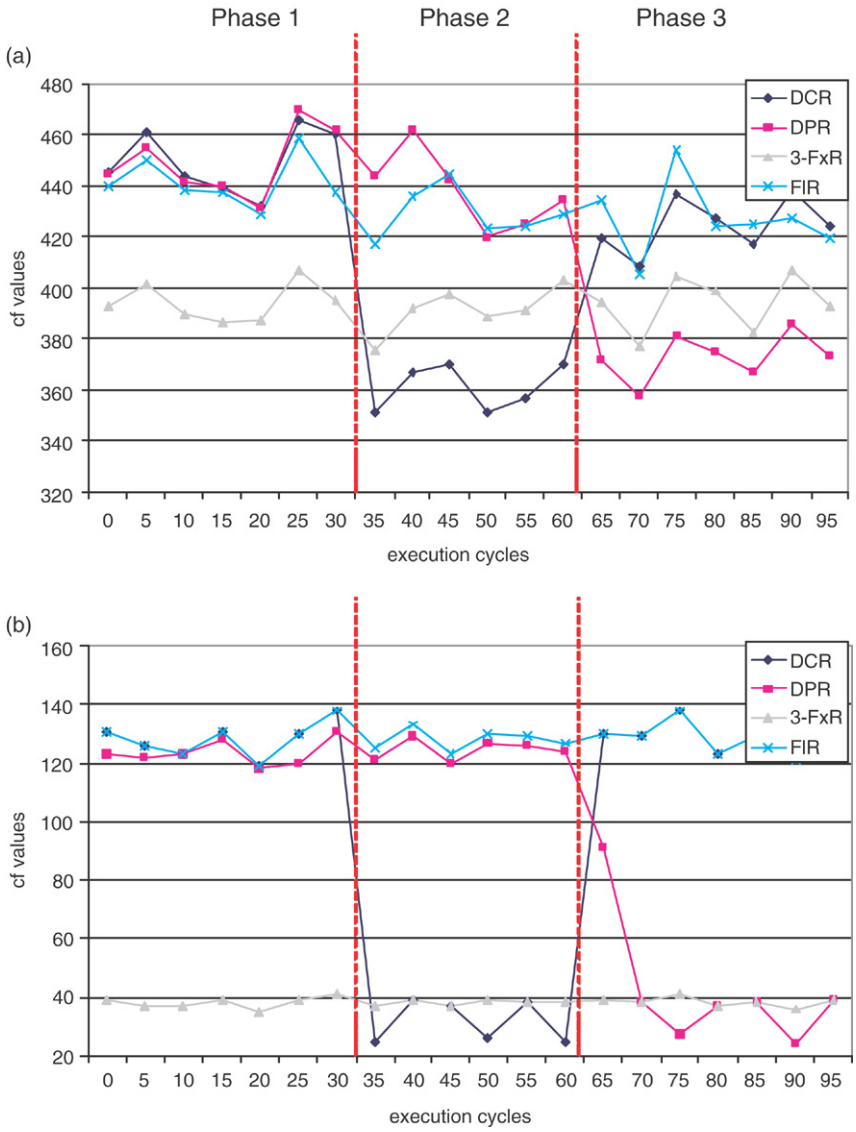

Fig. 6. Cost Function values for the real system (a) and for the simulation (b).

In particular, during the first phase of operations the policy that has the lowest cost function values is 3-FxR. This is due to the fact that 3-FxR uses the smallest number of nodes for replicating the tuples to. At the same time, the nodes used are able to sustain an availability level that satisfies the required 70% availability.

During Phase 2, the best policy is DCR. DCR uses only consumer nodes for replicating the tuple type. During this interval there are only two consumer nodes, that is n_9 and n_{10}. Thus DCR uses the smallest group of nodes with respect to the other policies for replicating tuples. Moreover, with this group DCR can sustain the required availability.

In the last interval the best policy is DPR. This is similar to the case of DCR. Only here, the two producer nodes are used as replication group for the tuple type.

Although the trends on both graphs show similar behaviors, the values of the cost function on the Y-axis are different. The reason for this can be explained by the contribution that the availability and bandwidth metrics give in the cost function during the experiment for the real implementation and the simulation:

- Concerning availability: the mechanisms for controlling node availability in the real implementation are different than in the simulation. In the implementation, the only way of controlling the availability of a node is by controlling the amount of crash operations that are sent to the node during the experiments. Instead, in the simulation, node

availability is just another parameter of the simulations defined in a configuration file. This provides a much easier way of controlling the value during each evaluation: a crash of node n is simulated by reducing the availability value of n with some percentage.

- Concerning bandwidth: the discrepancy between the implementation and the simulation is a consequence of the way in which logs are ordered for the evaluation. In the implementation, the evaluation is based on logs that are collected by each node. When the threshold is reached, the logs are assembled on the master AM for evaluation. The order in which the logs are assembled is based on timestamps that are assigned to a log when the request arrives at the kernel. Although the mechanism takes care of synchronizing the clocks during log assembly, it could be the case that the order in which the logs are assembled is not exactly the same as the order in which operations are executed by the application components. This comport a different contribution of the bandwidth metric to the overall costs. While, in the case of the simulation, logs were generated by the Coordinator and passed directly to the evaluation mechanism.

Since the trends of both graphs show a similar behavior we can conclude that the Adaptation Subsystem in both simulations and real executions adapts according to the same pattern.

In the following section, we will present results on the overhead and scalability of the Adaptation Subsystem.

3.4. Adaptation mechanism performance

We used an actual implementation of GSpace together with a synthetic benchmark application to perform some performance measurements of the adaptation mechanism. Here, the application behavior pattern is set as all nodes are consumers and producers and this does not change over time. What changes is the availability of nodes. In this set of experiments, failures are equally distributed over all nodes in the system.

The results that we obtained can be divided into three groups. The first group concerns the application perceived latency. In particular, we analyzed the tradeoff between the overhead and the gain in terms of latency that application components perceived due to the execution of the adaptation mechanism.

The second group of results concerns the evaluation phase of the adaptation mechanism. In particular, we analyzed how the time used for evaluating is influenced by:

- The length of a series of operations (in the system as a whole) that triggers the evaluation phase. We refer to this length as L_e.
- The number of nodes used for conducting the experiments.

The last group focuses on the correctness of the decisions taken versus the specific overhead of the adaptation phase.

3.4.1. Overhead versus gain of the adaptation mechanism

For estimating the overhead, we measured the application perceived latency[2] in executing blocking operations, such as read and take. We used the same run of 10 000 operations equally composed of reads, puts and takes. The run was executed over 10 nodes.

During the experiments, failures could occur simultaneously on several nodes. This means that all nodes in the group where tuples are replicated could be down at the same time. As a consequence, a read or a take operation could be blocked until one of the nodes in the group recovers from the failure and can serve the operation.

We executed the same run of operations using the following settings:

- net-latency no-adapt (*nlna*): in these settings, down time and recovery time of a failure were not included in the latency measurements of components executing read and take operations. Moreover, the Adaptation Subsystem was not activated.
- net-latency with adapt (*nlwa*): as in *nlna*, but this time the Adaptation Subsystem was activated.
- total-latency no-adapt (*tlna*): in this case, application latency was measured taking into account the down time and recovery time due to a failure. The Adaptation Subsystem was not activated.

[2] The clock starts when the application component issues a request and it stops when the tuple is retrieved and available to the application.

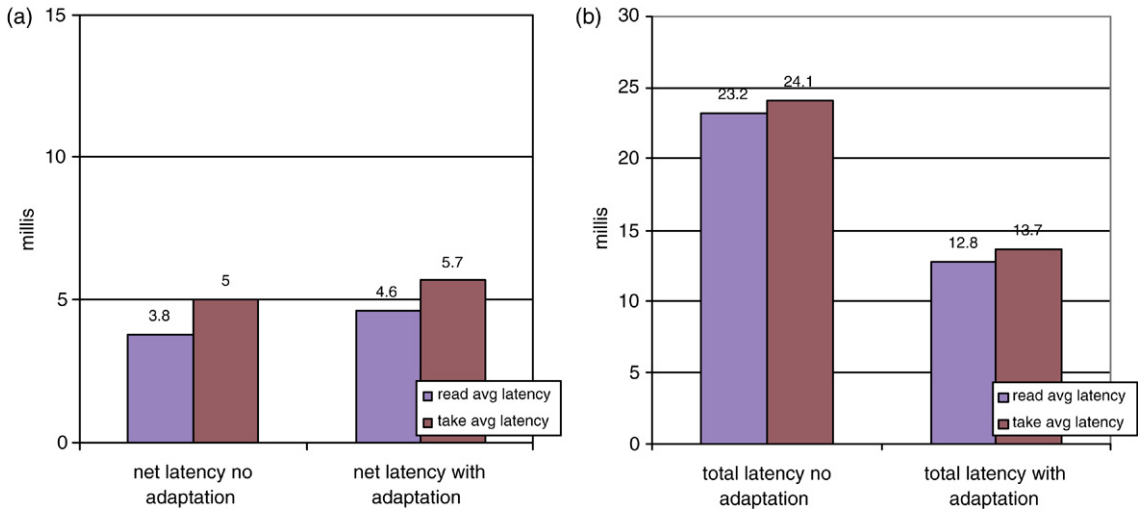

Fig. 7. Read and take latency values in which (a) the group failure latency is not considered. In (b) group failure latency is taken into account.

- total latency with adapt (*tlwa*): as in *tlna*, but with the Adaptation Subsystem activated.

During the experiment we measured for a component c_i the values $avg\_read\_t_i$ and $avg\_take\_t_i$. Averaging again these values for the number of components we obtain the average time for read and take operations. The results in milliseconds are shown in Fig. 7.

Fig. 7(a) compares the measurements for *nlna* and *nlwa*. There is a slight increment of latency in *nlwa* compared to the latency in *nlna*. This increment is due to the Adaptation Subsystem, and in particular when the adaptation phase is executed. During this phase, all the operations in the system are frozen for avoiding inconsistency issues. This affects the latency time perceived by application components.

However, when down time and recovery time are taken into account things change drastically. Fig. 7(b) shows that when down time and recovery time are taken into account, the average latency that a component perceives with adaptation is almost half of the time when adaptation is not activated. This is due to the adaptation that dynamically changes the group of nodes where the tuples are replicated taking into account the availability of each node. This is further confirmed by counting the numbers of group failures that occurred during the experiments. In fact, when adaptation was not enabled (*tlna*) the number of group failures adds up to 45. In the case of adaptation enabled (*tlwa*), only 19 group failures occurred.

In the next section, we will investigate the influence that length L_e and the number of nodes used have on the evaluation phase.

3.4.2. Evaluation phase measurements

Each evaluation phase quantifies the past behavior of the system by simulating it. A simulation mimics the past system behavior using traces of the operations that were executed up until the moment of evaluation. The higher L_e the longer the traces that are used for the evaluation. The traces are collected by each kernel in the system by logging the operations that it executes. When the evaluation is triggered, the logs from each kernel are collected at a single kernel (the Master AM) where the evaluation is executed. As a consequence, the larger the number of nodes the longer the Master AM takes for aggregating and assembling the logs from the each node. Our question is how well the evaluation execution time scales with respect to L_e and the number of nodes.

To find this out, we executed two sets of experiments. For each set the execution time of each evaluation phase was measured. In the first experiment set, we executed the same run of 10 000 operations on a 10-nodes set-up for different values of L_e. Fig. 8(a) shows the evaluation execution time in milliseconds (Y-axis) for different values of L_e. As the graph shows, the evaluation time scales linearly with respect to the threshold.

In the second set of experiments, we used a run of 10 000 operations and a fixed threshold value of 250. Here, we use a different number of nodes for each execution. Fig. 8(b) shows the evaluation times obtained for the different

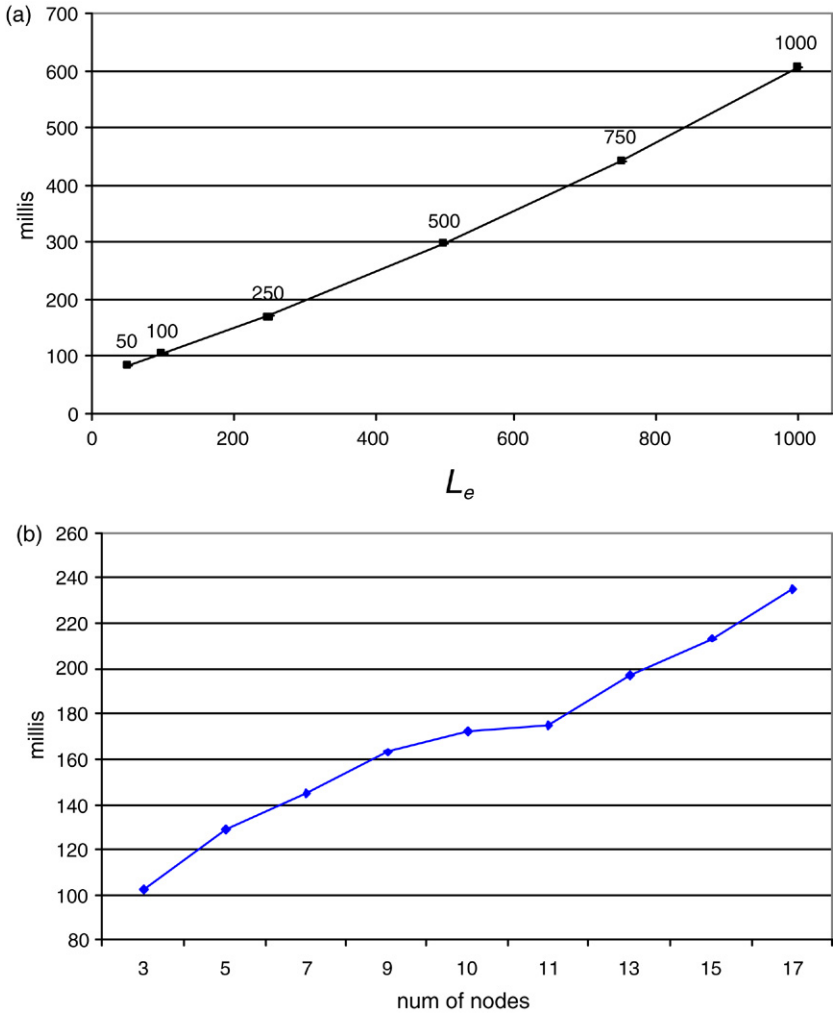

Fig. 8. Average execution time for evaluation phase when (a) L_e and (b) number of nodes change.

numbers of nodes. The results show that the aggregation time is linearly dependent on the number of nodes. In fact, the Master AM sends point-to-point messages to the other kernels asking for their logs.

We conclude that the evaluation time scales well for a local-area network.

3.4.3. Node awareness versus node adaptation penalty

In order to assess the efficiency of our approach, we consider how L_e influences the quality of evaluations. At each evaluation we look at the most recent L_e operations and at the availability of nodes. Based on this a combination of an availability policy and a set of nodes is chosen that together provide a certain level of availability as well as minimize the costs of replication. If this combination differs from the current settings the policy or group of nodes will be changed.

We focus on the part of the adaptation mechanism that makes a selection of nodes. We analyze the impact of the frequency of node evaluations on the performance of the adaptation mechanism.

The problem of determining an appropriate value for L_e requires finding a proper balance between the following concerns:

(1) Taking L_e too large may cause the mechanism to fail to adapt timely to changes in the node availability.
(2) Taking L_e too small may cause a performance penalty because evaluations are performed too often.

Firstly, we investigate how L_e influences the percentage of adaptations. Secondly, we show how L_e relates to the execution penalty of the adaptation mechanism.

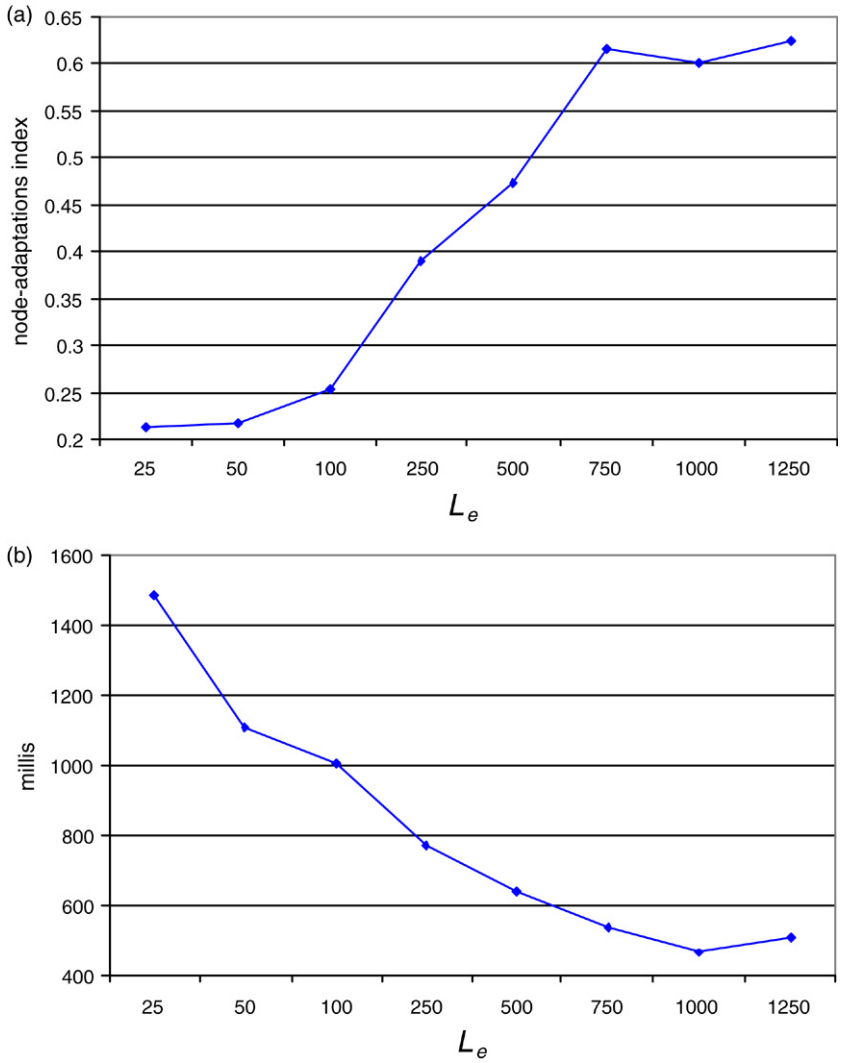

Fig. 9. Node adaptation index (a) and execution penalty (b) for the adaptation phase.

For measuring the influence that L_e has on the adaptation decisions we performed the following experiment. We executed a run of 10 000 operations on a set-up of 10 nodes using different values for L_e. During the execution only node availability is changed. In this case, failures are uniformly distributed over all nodes. The application behavior is maintained the same through the entire execution. In particular, all the application components behave as consumer and producer of tuples.

For each run, we measured the number of times that the evaluation phase was triggered and the number of times that such evaluation leads to an adaptation of the node group. If an adaptation happens, this means that the previously selected nodes were no longer the best. Consequently, the fewer nodes we need to change, the better our mechanism was capable of keeping up with actual availability. We define the *node_adaptation_index* = *#node_adaptations* / *#evaluations*. The lower the index, the better the system is able to keep track of changes in node availability. The results of these experiments are shown in Fig. 9(a). It can be seen that for smaller values of L_e the system is better able to select the nodes with highest availability.

However, higher responsiveness comes at a price. For the aforementioned experiments, we measured the total execution time spent in changing nodes in a group (excluding evaluation time). The results of these measurements are shown in Fig. 9(b). The time values on the Y-axis are given in milliseconds. The graph clearly shows how small thresholds incur a higher penalty. During these adaptations all application requests are frozen to avoid the introduction of inconsistency. Hence this causes a increase in application perceived latency.

3.5. Discussion

The experiments show that the autonomic mechanism of GSpace helps in improving the performance of the system in environments where node availability and application behavior dynamically change. The measurements show that the mechanism's resource footprint is negligible compared with the increment in performance that the application perceives.

The mechanism is capable of autonomously balancing the tradeoff between conflicting goals. On one hand, the mechanism tries to sustain a required availability by replicating tuples across several nodes. On the other hand, the mechanism must keep resource usage to a minimum. Through the use of a simple cost function as a linear combination of different performance metrics, the mechanism requires minimal input from the application developer.

However, it is necessary to select a value for L_e to set the pace at which the mechanism has to be activated for monitoring the environment. As the last set of measurements show, large L_e values mean a low degree of awareness of the mechanism. This means that the mechanism cannot adapt in time and the benefits are minimal for the overall system. Conversely, for small values of L_e the mechanism is more responsive to environment changes at the cost of higher resource usage. To complicate matters further, choosing the *right* L_e value is strictly dependent on the pattern according to which the environment characteristics change. If the application behavior does not change too much or the node availability is stable then choosing a small L_e only means wasting resources.

As it is now, the choice of the appropriate L_e value is given to the developer or system administrator, who is expected to have some knowledge of the environment characteristics where the system is deployed. Further research is needed for investigating extensions of our mechanism to autonomously extrapolate these patterns and select an appropriate L_e value.

4. Related work

Shared data spaces have a had a relatively long history ever since their introduction in the 1980s as a convenient means to program parallel applications. There has been much research on how to optimize the distribution and replication of tuples for parallel applications. A generic approach that balances the broadcasting of read and write operations is described in [1] and exemplifies tuple spaces for parallel applications.

Parallel applications often exhibit strong regular behavior, for which reason good distribution strategies can often be devised. Matters become complicated for general distributed applications, in which case usage patterns can often be very irregular. Various groups have explored the use of shared data spaces for such applications, but only relatively few have explicitly addressed fault tolerance.

4.1. Fault-tolerant shared data spaces

PLinda [9] is a variant of Linda that addresses fault-tolerant applications. In PLinda both data and processes are resilient to failures. In particular, by using a transaction mechanism extended with a process checkpoint scheme, PLinda ensures that a computation is carried out despite node failures. Compared to our approach, PLinda offers more functionality since it is resilient against process failures. On the other hand, in PLinda application developers have to explicitly declare which part of their application code should be executed in a fault-tolerant fashion. Application code is thus interwoven with extra-functional concerns not relevant to the application functionality. Our goal has been to separate these concerns.

Another fault-tolerant implementation of Linda is FT-Linda [2]. Like PLinda, FT-Linda supports a transaction mechanism that allows the recovery of data and processes after a failure. However, FT-Linda requires the application developers to put extra effort in making their application resilient to failures. For instance, the application developer has to program the application to remove intermediate results after a failure. By separating fault-tolerant issues from application functionality, we claim that it is easier to develop distributed applications that make use of shared data spaces, as in our case.

Although it was designed for taking advantage of idle time of workstations for running parallel applications, Piranha [11] could be used for addressing fault-tolerant applications as well. In Piranha, worker processes execute tasks on idle workstations. As soon as a workstation becomes busy, a worker process has to stop its current computation. The task has to be carried out by another Piranha worker on another idle workstation. Therefore, a retreat

has the same effect as a failure. The Piranha model assumes that the execution of the task is carried out atomically despite the retreat. The Piranha system requires the application developer to program the application to clean up intermediate results when a task has to retreat. Again, we see that application code is interwoven with fault-tolerant concerns.

An alternative approach for building fault-tolerant shared data spaces is proposed in [18]. In this work, code mobility is exploited as a mechanism for fault tolerance. By using code mobility, the system can guarantee operational semantics in which either all operations are executed or none. The approach uses a runtime system that contains a checkpointing mechanism. In this way, the application developer does not need to interweave code for fault tolerance in the application since the runtime system will deal with this. To address the removal of legacy data left by a mobile agent that is no longer alive, so-called *agent wills* are used. The agent will is a small piece of code embedded in the runtime system that describes what to do with data after the agent ceases activity, and is automatically executed when the associated agent dies.

An evaluation of fault-tolerance methods for large-scale distributed shared data spaces is described in [24]. The authors divide the methods for providing fault tolerance in two groups: runtime level and application level. The first group contains methods that provide fault tolerance only for the shared data space content. These methods make sure that the space content is not lost even if one or more nodes fail. The second group contains methods that extend fault-tolerant guarantees to the application level. These methods can ensure that a sensitive tuple (such as one acting as a lock) is not lost in case the application component that retrieved it fails. Clearly GSpace falls in the first group.

The authors continue their evaluation by describing the mechanisms that are typically used for fault tolerance. These include transactions, mobile code, checkpointing, and replication. The set of availability policies that can be built for GSpace utilize checkpointing and replication as basic mechanisms for providing fault tolerance.

Interesting, the authors conclude their evaluation pointing out that for supporting fault tolerance in Linda-like systems it is desirable to separate the set of primitives into two levels: functional and non-functional. Of the two, the latter should be used for specifying actions related to fault-tolerant concern. This is exactly in line with our way of dealing with fault tolerance, although we go further by completely separating this concern from the application functionality. Another interesting conclusion of the authors is that supporting runtime adaptation of fault-tolerant mechanisms can improve the optimization of the resource usage that fault-tolerant mechanisms require. This is exactly what the experimental findings of our research show.

4.2. Other distributed shared data spaces

More recently, we can see a regained interest in deploying shared data spaces for distributed systems, again showing that the decoupling of processes in time and space remains important. Moreover, the commercial deployment of the model has been quite successful, exemplified by systems such as JavaSpaces [6], TSpace [26] and GigaSpace [8].

The main problem with distributed shared data spaces is to achieve scalability. The approaches for parallel applications worked because access patterns are regular (and often known in advance). For general applications, we need runtime solutions by which inserted tuples can be efficiently found. One specific approach is to store tuples locally and to let templates (representing queries) to search for matches. This approach has essentially been adopted in PeerSpaces [3], although it is possible to move tuples to establish load balancing. Also, PeerSpaces allows tuples to be replicated to increase performance and availability. In the last case, only nondestructive read operations are allowed, and tuples are removed after a specified expiration time. Searching is done through (limited) flooding of templates, similar to what is done in peer-to-peer systems such as Gnutella.

An alternative approach is proposed in SwarmLinda [14,25] in which a randomized algorithm based on swarm intelligence is used to direct searches to tuples. In this case, we see that scalability may be achieved so long as the search algorithms prove to be efficient. Note, however, that SwarmLinda by itself does not explicitly address fault tolerance, although by its nature allows a shared data space as a whole to adapt to process failures. In such cases, a query will just follow a different route to a matching tuple.

This adaptive nature is also explored in the TOTA middleware [12,13] where tuples are equipped with a piece of code telling it how it should move through a distributed data space. In TOTA, tuples are not associated with individual data spaces or nodes, but instead are directly inserted into an overlay network, from which point on they can be propagated and diffused according to their own rules. Moreover, during this process, their content is allowed to change making it possible to adjust to the context in which a tuple currently resides. Again, it is the flexibility of the

approach that will allow handling of failures, although fault tolerance by itself is not explicitly dealt with in the work on TOTA described so far.

These approaches, and notably TOTA, can to a certain extent support mobility. Mobility is explicitly addressed in Lime [16,15], and has recently been extended to sensor networks [5]. In Lime, a data space is associated with a single node, but can merge with other nodes when the two are in each other's (transitive) range. In this way, queries can be matched against tuples when a data space comes in proximity of an outstanding read operation. A similar approach has also been explored in the SPREAD system [4]. Again, fault tolerance is not explicitly addressed, but can easily be incorporated as a separate concern using techniques that are independent from those used to support mobility.

5. Conclusions and future work

In this paper we made the following contributions. First, we provided a simple mechanism that allows for addressing availability concerns in shared data space systems separately from the functionality of applications. As a result, different policies can be employed for achieving different availability characteristics without affecting the functionality of the application.

Second, we demonstrated how possibly conflicting objectives (such as high availability and low resource use) can be dealt with in a fully automated fashion through the use of a cost function. Additionally, our mechanism requires minimal input from a developer to support these adaptations, with the benefit of allowing the developer to concentrate on the design and implementation of functionality.

Third, through a series of experiments we showed the superior performance of dynamically adapting availability policies to suit the availability characteristics of the infrastructure and application behavior.

Finally, we measured the overhead of the adaptation mechanism and compared it with the benefits that applications can achieve by dynamically adapting policies. The conclusion is that the gain of using adaptation overcomes the incurred overhead. However, it is crucial to identify which L_e value matches the application and environment characteristics. We provided an experimental analysis of the impact that the L_e value has on the performance of the adaptation mechanism.

As future extensions of our work we would like to port GSpace in a WAN environment. For this we foresee a partial re-engineering of the kernel discovery mechanism of GSpace. Additionally, the availability policies should be adapted to operate in a wide-area environment where the GCP cannot operate. However, we estimate that the adaptation mechanism should be able to cope with a wide-area network, as is already proven by the work done for adapting replication policies for Web pages distributed over the Internet, described in [17].

Another interesting extension would be one that deals with the deployment of GSpace in ad hoc environments. In this case, we should reconsider our model of a failure. In fact, if a node is not available it does not necessarily mean that it is not still operating. As a consequence, availability policies should also support reconciliation mechanisms when a strong-consistency model must be supported. However, in such an environment a loose-consistency model can be a better (if not the only viable) alternative. Thus, the kernel should be extended with new policies that support replication but do not enforce a strong-consistency model.

This work is an extension of earlier work where we studied separation of extra-functional concerns in shared data spaces. In [21] we showed how resource usage could be treated as a separate policy and in [23] we studied the separation of real-time and exception handling concerns. The next challenge is combining multiple concerns in one architecture. Some of these concerns are inherently coupled, yet the challenge is to find a way of combining these concerns in a single architecture that enables ease of engineering and adaptability to changes in the usage profile.

References

[1] S. Ahuja, N. Carriero, D. Gelernter, V. Krishnaswamy, Matching languages and hardware for parallel computation in the Linda machine, IEEE Transactions on Computers 37 (8) (1988) 921–929.

[2] D.E. Bakken, R.D. Schlichting, Supporting fault tolerant parallel programming in Linda, IEEE Transactions on Parallel and Distributed System (1994).

[3] N. Busi, A. Montresor, G. Zavattaro, Data-driven coordination in peer-to-peer information systems, International Journal on Cooperative Information Systems 13 (1) (2004) 63–89.

[4] P. Couderc, M. Benâtre, Ambient computing applications: An experience with the SPREAD approach, in: Proc. 36th Hawaii International Conference on System Sciences, IEEE, January 2003.

[5] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, G. Picco, TinyLime: Bridging mobile and sensor networks through middleware, in: Proc. 3rd International Conference Pervasive Computing and Communications, PerCom, IEEE Computer Society Press, Los Alamitos, CA, March 2005, pp. 61–72.
[6] E. Freeman, S. Hupfer, K. Arnold, JavaSpaces Principles, Patterns, and Practice, Addison-Wesley, Reading, MA, USA, 1999.
[7] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 80–112.
[8] GigaSpaces, GigaSpaces enterprise application grid 4.1 documentation, 2005.
[9] K. Jeong, D. Shasha, PLinda 2.0: A Transactional/checkpointing approach to fault tolerant Linda, in: Proc. 13th Symp. on Reliable Distributed Systems, Dana Point, CA, 1994, pp. 96–105.
[10] M.F. Kaashoek, A.S. Tanenbaum, Efficient reliable group communication for distributed systems, Internal Report IR-295 IR-295, Department of Computer Science, Vrije Universiteit of Amsterdam, 1992.
[11] D. Kaminski, Adaptive parallelism in Piranha, Ph.D. Thesis, Yale University, Department of Computer Science, 1994.
[12] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications with the TOTA middleware, in: Proc. 2nd International Conference Pervasive Computing and Communications, PerCom, IEEE Computer Society Press, Los Alamitos, CA, March 2004, pp. 263–273.
[13] M. Mamei, F. Zambonelli, Spatial computing: The TOTA approach, in: O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, M. van Steen (Eds.), Self-Star Properties in Complex Information Systems, in: Lecture Notes on Computer Science, vol. 3460, Springer-Verlag, Berlin, 2005, pp. 307–324.
[14] R.M.R. Tolksdorf, A new approach to scalable Linda-systems based on swarms, in: Proc. Symposium on Applied Computing, ACM Press, New York, NY, March 2003, pp. 375–379.
[15] A. Murphy, G. Picco, G.-C. Roman, Lime: A middleware for physical and logical mobility, in: Proc. 21st International Conference on Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos, CA, April 2001, pp. 524–533.
[16] G.P. Picco, A.L. Murphy, G.-C. Roman, Lime: Linda meets mobility, in: D. Garlan, J. Kramer (Eds.), Proc. 21st International Conference on Software Engineering, ICSE'99, ACM Press, Los Angeles, USA, ISBN: 1-58113-074-0, May 1999, pp. 368–377.
[17] G. Pierre, M. van Steen, A. Tanenbaum, Dynamically selecting optimal distribution strategies for web documents, IEEE Transactions on Computers 51 (6) (2002) 637–651.
[18] A. Rowstron, Using mobile code to provide fault tolerance in tuple space based coordination languages, Science of Computer Programming 46 (1–2) (2003) 137–162.
[19] G. Russello, M. Chaudron, 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.
[20] G. Russello, M. Chaudron, M. van Steen, Exploiting differentiated tuple distribution in shared data spaces, in: Proc. Int'l Conference on Parallel and Distributed Computing, Euro-Par, vol. 3149, Springer-Verlag, Berlin, 2004, pp. 579–586.
[21] G. Russello, M. Chaudron, M. van Steen, Dynamic adaptation of data distribution policies in a shared data space system, in: Proc. Int'l Symp. On Distributed Objects and Applications, DOA, vol. 3291, Springer-Verlag, Berlin, 2004, pp. 1225–1242.
[22] G. Russello, M. Chaudron, M. van Steen, Dynamically adapting tuple replication for high availability in a shared data space, in: J.-M. Jacquet, G.P. Picco (Eds.), Proc. 7th Int'l Conf. on Coordination Models and Languages, Coordination 2005, vol. 3454, Springer-Verlag, Berlin, 2005, pp. 109–124.
[23] R. Spoor, Design and implementation of a real-time distributed shared data space, Master's Thesis, Eindhoven University of Technology, Department of Computing Science, 2004.
[24] R. Tolksdorf, A. Rowstron, Evaluating fault tolerance methods for large-scale Linda-like systems, in: Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, vol. 2, June 2000, pp, 793–800.
[25] R. Tolksdorf, R. Menezes, Using swarm intelligence in Linda systems, in: Proc. 4th International Workshop in Engineering of Societies in the Agents World, ESAW, in: Lecture Notes in Artificial Intelligence, vol. 3071, Springer-Verlag, Berlin, October 2003, pp. 49–65.
[26] P. Wyckoff, S. McLaughry, T. Lehman, D. Ford, T Spaces, IBM System Journal 37 (3) (1998) 454–474.