Dynamically Adapting Tuple Replication for Managing Availability in a Shared Data Space

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

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

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 separation of concerns. In particular, we present a solution that allows developers to merely specify performance and 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.

1 Introduction

The shared data space model has proven to be a useful abstraction for the development of 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 the insertion and removal of components at runtime. This dynamic configuration is possible when components encapsulate functionality that has been coded independent of any runtime environment. When extra-functional requirements have been addressed (such as those for performance), widespread component deployment becomes more difficult. In essence, we are facing the problem of separating various concerns when developing and deploying components in distributed systems.

One solution to address this separation is exploiting the underlying middleware. In particular, we believe that the middleware should provide the mechanisms for specifying and enforcing extra-functional concerns. For example, if replication is required, the middleware should ideally offer mechanisms that would allow the application developer to select from different replication policies that can be subsequently enforced at runtime. If necessary, new policies can be developed and deployed as well, independent of the basic functionality implemented by legacy components.

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 have been able to dynamically adapt data distribution to the needs of an application. We have thus effectively created a closed feedback-control system, now often popularly coined as a self-managing or autonomic system.

So far, 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 components may unpredictably fail, particular care has to be taken for shared data items to remain available to other components. Similar issues arise when an application is deployed on mobile nodes. In such an environment, a node's connectivity may be highly unpredictable and a set of data items may unexpectedly disappear when a node disconnects.

A well-know 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 connected (or has failed). 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 generally requires that updates are carried out on 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 replication 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 continous 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 performance 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 mechanisms that drive GSpace decisions. 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.

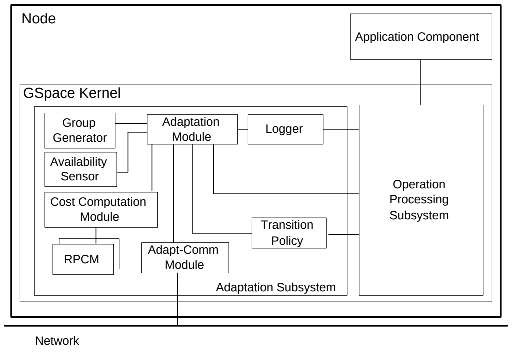


Fig. 1. Internal stucture of a GSpace kernel deployed on a node.

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 Data Space Basic Concepts

The data space concept was introduced in the coordination language Linda [3]. In Linda, applications communicate by inserting and retrieving data through a data space. The unit 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. The data space is similar to a *multiset* of data, where multiple instances of the same data item can co-exist.

2.2 Architectural Design

A typical setup of GSpace 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. This allows the system to associate different replication policies with different tuple types.

Figure 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; 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 detail elsewhere [9].

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. 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 concerned which replication policy should be applied to a tuple type. The slaves AM follow the decisions of the master. 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 we added for dealing with availability are the following:

Availability Sensor: This module is responsible for measuring the availability of the node in which it is deployed. This is done by periodically writing timestamps in a file. When a failure occurs, this time-stamp is used to compute the duration that the node was not available.

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 3 nodes in term of availability. The group that the Group Generator creates is then passed to the 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.

2.3 Autonomic Behavior in GSpace

This section describes the mechanism that allows GSpace to dynamically evaluate and select the replication policy that fits best the needs of the application.

In a distributed system such as GSpace, tuples are often stored and accessed remotely. Since nodes may fail or get disconnected, part of the shared data space could 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 for storing tuple replicas and bandwidth for exchanging information needed for keeping the replicas in a consistent state. Also, keeping replicas consistent comes at the price of global synchronization when updates occur.

Instead of proposing a one-size-fits-all solution, our approach sets flexibility as its primary goal. We included in GSpace a suite of replication policies each with its own tradeoff between provided availability, resource consumption, and performance. In this paper, we ignore performance issues, allowing the application developer to specify only the availability requirements for the tuple types used by the application. The problem is now shifted to finding the replication policy that (a) minimizes resource consumption while (b) fulfilling the availability requirements. These conditons 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 availablity in real-time. By combining these data, our mechanism can automatically detect when to switch to another replication 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 explain in turn.

- monitoring phase
- evaluation phase
- adaptation phase

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 information on node availability, the GSpace kernel is instrumented with a sensor that monitors the availability of the node where it is running. Before diving into implementation details, we introduce the basic math behind the measurements that our sytem performs. The formula for calculating the availability of a single node is as follows:

$$Availability = \frac{\textit{Mean Time To Failure}}{\textit{Mean Time To Failure + Mean Time To Recover}} \tag{1}$$

It is important to understand what exactly *Mean Time To Failure* (MTTF) and *Mean Time To Recover* (MTTR) mean. With MTTF we indicate the average time that the node is continuosly operating, i.e. the average time between the end of one failure and the beginning of the next. With MTTR we address the average time necessary for the node to recover from an experienced failure.

Figure 2 sketches the time line of a node that experiences some failures. When a failure i occurs we indicate with sf_i and ef_i respectively the time when the failure i

starts and ends. We assume that the starting time of the node (the very first time that the node is activated) is equivalent to ef_0 (end of failure 0). Figure 2 also provides a graphical representation of MTTF and MTTR that helps to understand how to compute those values. For instance, the availability value after the *n*-th failure is obtained by the following fromula:

$$Availability = \frac{\sum_{i=1}^{n}(sf_i - ef_{i-1})}{\sum_{i=1}^{n}(sf_i - ef_{i-1}) + \sum_{i=1}^{n}(ef_i - sf_i)} \qquad (2)$$

For computing (2) we need to collect the 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, the sensor is actived and writes timestamps into the same file. Actually, a timestamp is just the time at which the sensor is active. After a node experiences a failure, at re-booting time the sensor detects that the system was down (since the timestamp file is stored persistently). The starting time of a failure then is 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 executed timestamp.

For collecting information about the application behavior, we employ the same method as described in our previous work [11]. Each data-space operation that application components execute is logged and stored per tuple type. Figure 3 shows the message sequence chart during the operation logging. The data that is logged 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 GSpace kernel (i.e., node) 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

When the number of executed operations on a node reaches a given threshold the system starts the next phase.

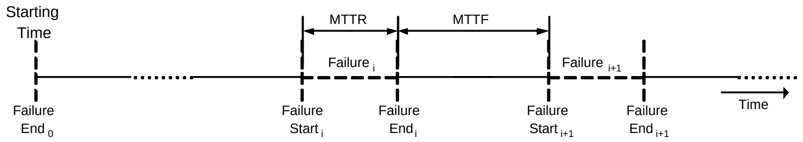


Fig. 2. The time line of a node that experiences some failures.

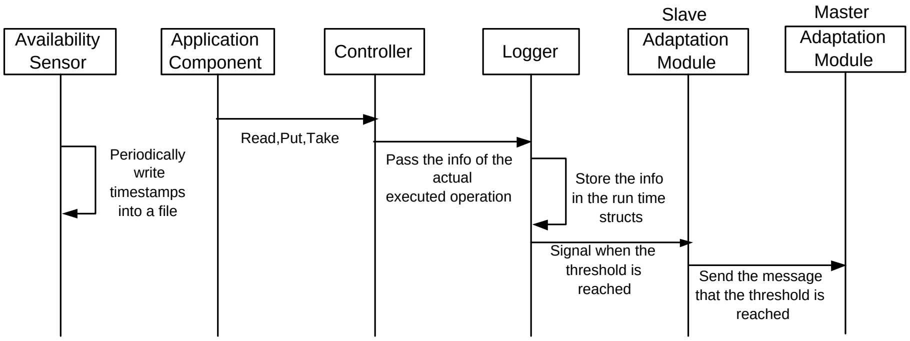

Availability Sensor
Application Component
Controller
Logger
Slave Adaptation Module
Master Adaptation Module
Read,Put,Take
Periodically write timestamps into a file
Pass the info of the actual executed operation
Store the info in the run time structs
Signal when the threshold is reached
Send the message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

Fig. 3. The MSC of operation logging.

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

Figure 4 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 incurred by each replication 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 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 replicationion policy that leads to the lowest costs represents the best policy for a given application.

In this work, we apply the same method as described in [11] 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:

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

In this way, 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:

$$CF_p \;=\; w_1 * bu(p) + w_2 * mu(p) + w_3 * da(p) \qquad (3)$$

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

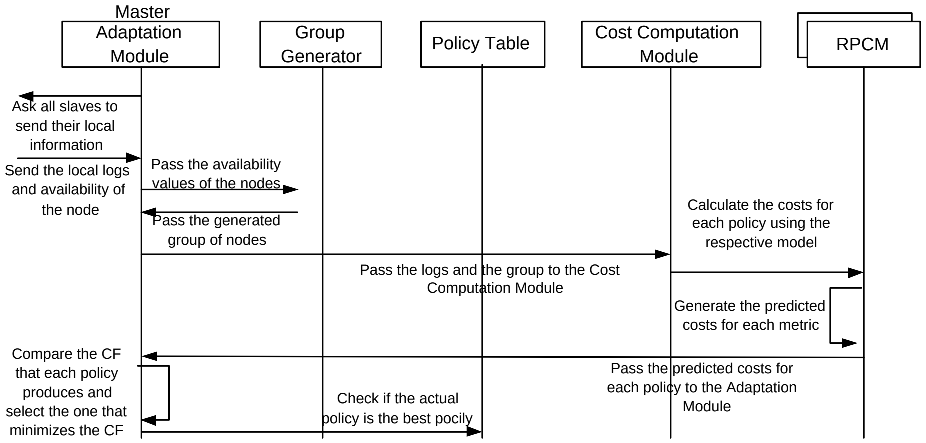

Master Adaptation Module
Group Generator
Policy Table
Cost Computation Module
RPCM
Ask all slaves to send their local information
Send the local logs and availability of the node
Pass the availability values of the nodes
Pass the generated group of nodes
Calculate the costs for each policy using the respective model
Pass the logs and the group to the Cost Computation Module
Generate the predicted costs for each metric
Compare the CF that each policy produces and select the one that minimizes the CF
Pass the predicted costs for each policy to the Adaptation Module
Check if the actual policy is the best pocily

Fig. 4. The MSC of the evaluation phase.

still the best one. If this is the case, no furhter actions are undertaken. Otherwise, the AM starts the phase described next.

Adaptation Phase During this phase the system switches replication policy and adapts the data space content accordingly. In Figure 5 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. Afterwards, the each kernel takes care of updating its own data structure and redistributes the tuples still in the space according to the new replication policy. When this transition period is finished the master AM resumes the operations in all nodes.

3 Implementation and Experiments

This section describes the experiments that we performed using a simulator of GSpace. The experiments model a distributed system with 10 nodes connected via a LAN.

Our previous experiments were concerned with distributed systems in which application components can dynamically join and leave a system during execution (but in which the nodes were always available). In [10] we showed that there is no single distribution policy that is best for this dynamic type of application behavior. Furthermore, in [11] we showed that dynamically adapting the distribution policy outperforms the use of 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. This occurs, for instance, in ad-hoc networks where devices join and leave a network.

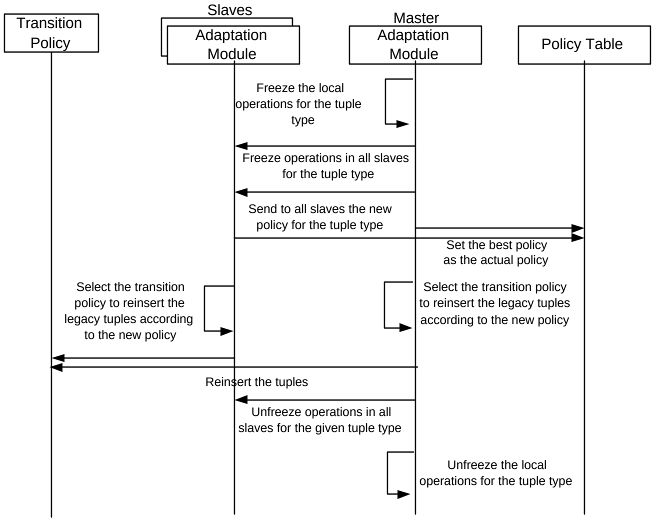

Transition Policy
Slaves
Adaptation Module
Master
Adaptation Module
Policy Table
Freeze the local operations for the tuple type
Freeze operations in all slaves for the tuple type
Send to all slaves the new policy for the tuple type
Set the best policy as the actual policy
Select the transition policy 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.

We show the impact of changing infrastructure on sustaining a level of availablity: without adaptation, no single static policy is able to sustain a given level of availability. Moreover, we show that the dynamic adaptation of the policy provides a better level of availability in the case of changing infrastructure. Furthermore, we show that the adaptation mechanism can handle situations where both the infrastructure as well as the application behavior change dynamically.

The goal of the experiments is to show that our system can adapt the policy it uses to changes in the availability characteristics of the nodes in the network. As a result, it can maintain a level of availability of tuples while the availability of nodes in the underlying network varies.

The results of this simulation are now being incorporated in an our distributed implementation of GSpace. Previous experience with the simulation [11] shows that the accuracy of the simulation is in the order of 5 percent. Hence the simulation provides fairly accurate predications about actual system behavior.

In the following, we first describe the set-up of the experiments. Thereafter, we describe the replication policies used for the experiments. We conclude discussing two interesting cases.

3.1 Set-up of the Experiments

The experiments are based on the simulation of the deployment of GSpace in a network of 10 nodes connected via a LAN. We control the simulation experiment through the following parameters:

- *Application behavior*: the operations that the application components execute using GSpace.

The simulation contains a library of different application usage patterns. A pattern consists of a series of read, put and take operations. A *run* of an experiment consists of the concatenation of a number of patterns. The patterns in a run may be of the same type, or they may be of different types. The approach we follow for the synthesis of application behavior is described in [11].

- *Node availability behavior*: the availability characteristics of the physical nodes where GSpace is deployed; including its change over time.
  The availability behavior of nodes during execution can be set to one of the following:
  - constant
  - increasing from a given value to a max value by increments of a given δ
  - decreasing from a given value to a min value by increments of a given δ
  - alternating between a min and max value by increments of a given δ

During the simulation, data about performance parameters is collected and passed to the Adaptation Manager. Using this data the Adaptation Manager evaluates the cost function, and determines which replication policy to use in the next phase.

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**. This policy puts a copy of every tuple on every node in the system (as soon as a tuple is inserted)
- **Fixed Replication**. This policy replicates tuples to a fixed number of nodes (as soon as the tuple is inserted). When awareness of node availability is enabled, the Group Generator provides the nodes where tuples should be replicated.
- **Dynamic Consumer Replication**. 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 that hosts consumer components can not provide the required availability the policy includes in the group nodes provided by the Group Generator.
- **Dynamic Producer Replication**. This policy replicates tuples to all nodes that host an application component that is a produced of this type of tuple. Nodes provided by the Group Generator might be included in the group of producer nodes whenever this group can not sustain the required availability.

For maintaining consistency among the nodes where replicas are stored, the replication policies collaborate using a Group Communication Protocol. The nodes on which tuples are replicated are joined in a group where the operations are executed atomically. Moreover, the Group Communication Protocol takes care of consistency issues that could arise from the failure of some of the nodes in the group.

The availability that a given replication policy can sustain is determined by the availability of the group of nodes that is used for replicating tuples to. In particular, a group of nodes is considered available if at least one node of the group is available.

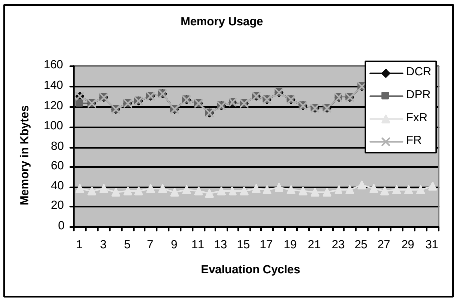


Fig. 6. Memory Usage measured for the different replication policies.

Then, the group availability, *GA*, equals 1 minus the probability that all nodes within the group fail:

$$GA = 1 - P_{all\_nodes\_down} \tag{4}$$

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:

$$P_{all\_nodes\_down} = \prod_{i=1}^{n} f_i \tag{5}$$

3.3 Adding Awareness of Node Availability to Policies

In this section we introduce replication policies that base their decisions on the availability of nodes. The experiments in this section show that by constantly monitoring the underlaying infrastructure, the GSpace system improves sustainability of the required availability requirements despite the unpredictable behavior of the nodes.

For the experiments in this section, we assume the application behavior is fixed according to the following pattern. All the application components act both as consumers and producers.

For this application behavior, both Dynamic-Consumer and Dynamic-Producer policies replicate the tuples in all nodes. This means that the memory usage is the same as that for the Full Replication policy, as Figure 6 shows. Instead, the memory footprint of the Fixed Replication policy is smaller than that of the other policies since this policy replicates tuples on a smaller number of nodes.

First we consider the case when the availability monitoring is disabled. The required availability for the tuple type used in the experiments is 70%. The Fixed Replication policy is defined to use the three nodes that provide the highest availability at the moment the system is started. However, the availability behavior of these nodes is programmed to decrease from 90 to 10 in steps of 5 (percent).

Using these three nodes the Fixed Replication policy initially satisfies the availability threshold. However, during execution, the nodes that are used by the Fixed Replication policy experience an increasing number of failures. Hence, the availability of the

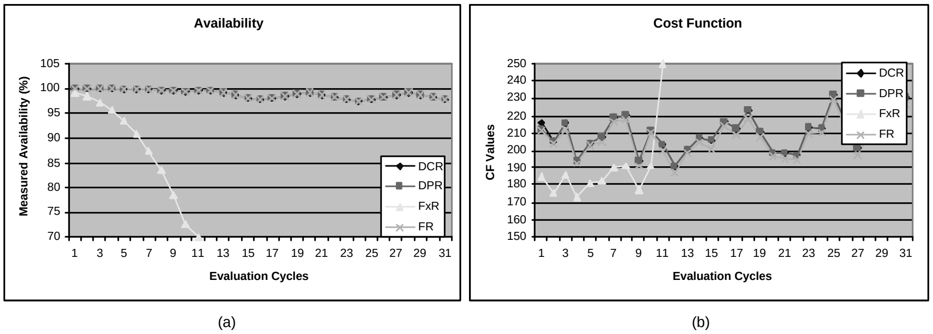


Fig. 7. Availability and Cost Funciton values for the replication policies when availability awarness is disabled.

nodes decreases and as a result, the availability that the Fixed Replication policy provides decreases. Figure 7(a) clearly shows this descreasing behavior. The other replication policies provide a fairly stable availability with minor fluctuations. This is due to the fact that the changing availabiliy of three nodes out of 10 has a minor impact on their overall availability.

The previous graphs were concerned with availability. Next, we look at the effect of the replication policies on the cost function.

From Figure 8(b), we can conclude that as long as the availability requirements are met, Fixed Replication is the best policy since it uses the least memory. However, around the 10th evaluation cycle this policy can no longer sustain the required level of availability. As a result, the cost function value increases dramatically.

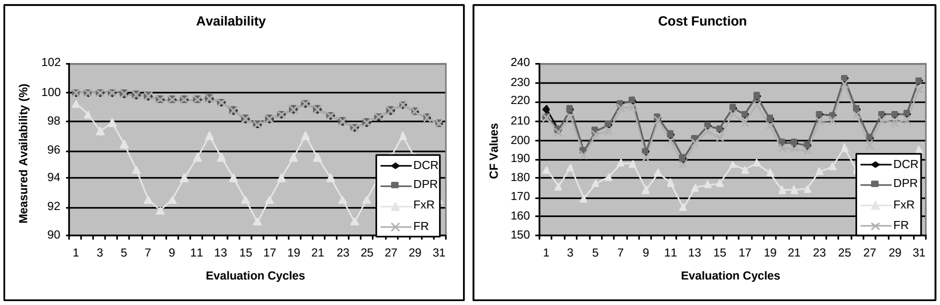
Next, we re-execute the same sequence of operations but this time with the availability monitoring enabled. The Fixed Replication policy still makes only a fixed number of copies, but now it selects the three nodes that have the highest availability at the time of evaluation[1].

The memory usage graph is the same as the one shown in Figure 6 since the application behavior is the same. However, now the system is able to select nodes based on the measured availability of the nodes. At each evaluation, the system selects the three nodes that have highest availability. Now, Figure 8(a) shows that Fixed Replication is able to provide the required availability. Moreover, since the memory footprint is lower than that of the other policies, Fixed Replication is always the best policy. This is shown in the cost funtion graph on Figure 8(b).

3.4 Combinging Dynamic Application Behavior and Dynamic Node behaviour

In this section we analyze the case where both the application components change their behavior and also the availability of nodes changes during execution. The results will

[1] These nodes are provided by the Group Generator module



(a) (b)

Fig. 8. Avaialbility and Cost Funciton values for the replication policies when availability awarness is enabled.

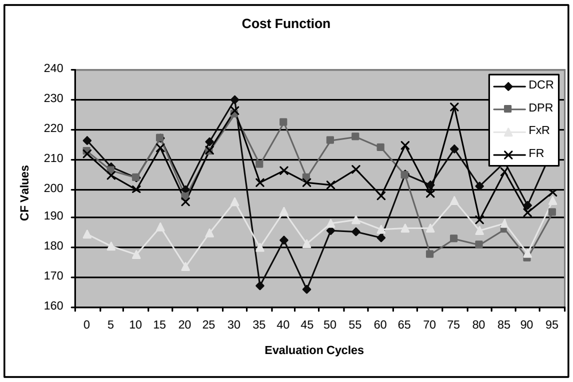


Fig. 9. Cost Function values when the application behavior changes.

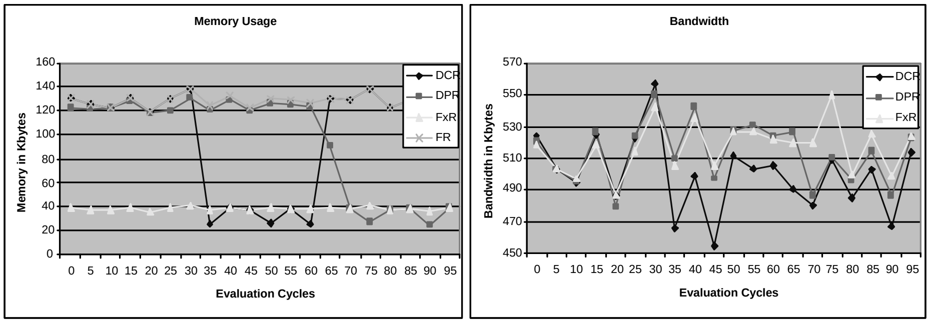
show that our mechanism not only is able to select the replication policy that satisfies the availability requirements but also it selects the policy that best suits the components' behavior.

During these experiments the availability characteristics of nodes are measured from the system and made available to GSpace. The application component behavior is programmed to change during execution according to the following phases:

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

Moreover, the availability of nodes n_9 and n_{10} is programmed to oscillate between 10% and 90%. Therefore, the group formed by these two nodes is not always able to sustain the required level of availability, which is fixed to 70%.

Let us begin analyzing the cost function values on Figure 9. During the first phase of the execution, the best policy that can guarantee the availability requirements with minimal memory usage is Fixed Replication.



(a)

(b)

Fig. 10. Measured Memory and Bandwidth Usage when the application behavior changes.

During the second phase of execution, Dynamic Consumer Replication is the best policy. This is due to two factors. Firstly, only two nodes host application components that act as consumers. Therefore, Dynamic-Consumer Replication uses a group of nodes that is at most as large as the group used by Fixed Replication. This has a major impact on the memory usage, as Figure 10(a) shows between evaluation cycles 32 and 64. In fact, when the combined availability of node n_9 and n_{10} is above the required availability, Dynamic Consumer Replication has a smaller memory usage footprint than Fixed Replication. However, sometimes those two nodes are not enough to guarantee the required availability. Thus, Dynamic Consumer Replication has to include other nodes to sustain the required availability. This is done by adding a node that is selected by the Group Generator module. The second factor is the reduced bandwidth usage that Dynamic Consumer Replication incurs. This is shown in Figure 10(b).

The last phase of execution witnesses another change. Application components switch behavior. In particular, after evaluation cycle 64, the application components on nodes n_9 and n_{10} start acting as producers. All the other components start to act as consumers.

After a transition phase between cycles 64 and 70, where the components' behaviour stabilizes, the Dynamic Produred Replication becomes the best policy, as Figure 9 shows. This is mainly due to the same factors that we discussed for Dynamic Consumer Replication. This is confirmed also by the graphs in Figure 10, after the stabilizzation phase.

To conclude, we want to show that in all cases the availability sustained by the policies used in the different phases is always greater than the required value (Figure 11). This is an improvement over the behaviour that is oblivious to changes in availability of nodes, yet the adaptation happens transparently to the application.

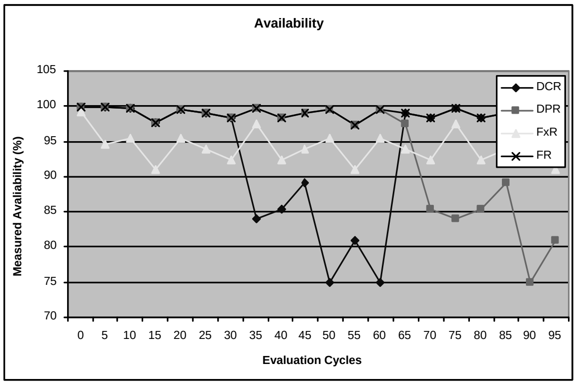


Fig. 11. Availability values when the application behavior changes.

4 Related Work

In this section we describe other approaches for realizing shared data space resilent to failures.

PLinda [4] is a variant of Linda that addresses fault-tolerant applications. In PLinda both data and processes are resilent 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 resilent against process failures. On the other hand, in PLinda application developers have to explicity declare which part of their application code should be executed in a fault-tolerant fashion. As a consequence, application code is interwonen with concerns that are not relevant to the main functionality of the application.

Another extension of Linda that provides support for fault tolerance is FT-Linda [1]. As for 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 resilent to failures. For instace, the application developer has to program the appliation to take care of removing intermediate results after a failure. Again, this is clearly against separating different concerns in the application design.

Although it was designed for taking adavantage of idle time of workstations for running parallel applicaitons, Piranha [5] 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 as the same effect as a failure. The Piranha model assumes that the execution of the task is carried out atomically despite the retreat. As for the FT-Linda, 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 interwonen with fault-tolerant concerns.

An alternative approach to transaction mechanism for building shared tuple space resilents to fault tolerance is proposd in [8]. In this work, the author proposes exploiting code mobility as a better mechanism for fault tolerance. By using code mobility, the system can guarantee an operational semantics in which either all operations are executed or none. The approach uses a run-time system that contains a checkpointing mechanism. In this way, the application developer does not need to interweave fault-tolerance code in her/his application since the run-time system will deal with this. To address the removal of legacy data left by mobile agent that is no longer alive, the author introduces the notion of *agent wills*. The agent will is a small piece of code embedded with the run-time system that describes what to do with data after the agent ceases activity. This will-code is executed by the run-time system whenever it detects that the respective agent crashed.

An evaluation of fault-tolerance methods for large scale distributed shared data spaces is described in [13].

Worthwhile to mention for the significance of their contributions, although not for fault tolerance, are the following implementations of shared data space. JavaSpaces [2] and TSpace [14] are commercial systems that have shown how the shared data space paradigm can be successfully used for building distributed applications. WCL [7] extends the basic primitives of the shared data space with some new ones. These new primitives allow the execution of operations that are impossible to achieve by the standard ones. For instance, the multiple read primitive returns copies of all tuples that match with a given template. Finally, Lime [6] addresses the issues of coordination in a distributed environment.

5 Conclusions and Future Work

In this paper we made the following contributions. First, we provide 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 demonstrate 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.

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.

Finally, we showed the superior performance of dynamically adapting the replication policy that is used. The experiments showed that our mechanism is able to dynamically adapt the replication policy to the availability characteristics of the infrastructure. Moreover, the mechanism takes in consideration the application behavior and selects the policy that suits best the application needs.

This work is an extension of earlier work where we studies seperation of extra-functional concerns in shared dataspaces. In [11] we showed how resource use could be treated as a separate policy and in [12] 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. D. E. Bakken and R. D. Schlichting. "Supporting Fault Tolerant Parallel Programming in Linda." *IEEE Trans. on Parallel and Distributed System*,1994.
2. E. Freeman, S. Hupfer, and K. Arnold. JavaSpaces principles, patterns, and practice. Addison-Wesley, Reading, MA, USA, 1999.
3. D. Gelernter. "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.
4. K. Jeong, D. Shasha. "PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda." *Proc. 13th Symp. on Reliable Distributed Systems*, 96–105, Dana Point, CA, 1994.
5. D. Kaminski. "Adaptive Parallelism in Piranha." PhD Thesis, Yale University, Department of Computer Science, 1994.
6. 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.
7. A. Rowstron. "WCL: a Co-ordination Language for Geographically Distributed Agent." In *Worl Wide Web Journal*, Vol. 1, Issue 3, pp. 167–179, 1998.
8. A. Rowstron. "Using mobile code to provide fault tolerance in tuple space based coordination languages." In *Science of Computer Programming*, Vol. 46, Number 1-2, pages 137-162, Jan. 2003.
9. 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.
10. G. Russello, M. Chaudron, M. van Steen. "Exploiting Differentiated Tuple Distribution in Shared Data Spaces." *Proc. Int'l Conference on Parallel and Distributed Computing (Euro-Par)*, 3149:579–586, Springer-Verlag, Berlin, 2004.
11. G. Russello, M. Chaudron, M. van Steen. "Dynamic Adaptation of Data Distribution Policies in a Shared Data Space System." *Proc. Int'l Symp. On Distributed Objects and Applications (DOA)*, 3291:1225–1242, Springer-Verlag, Berlin, 2004.
12. R. Spoor. "Design and Implementation of a Real-Time Distributed Shared Data Space." Master's Thesis, Eindhoven University of Thechnology, Department of Computing Science, 2004.

13. 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, pages 793-800, June 2000.
14. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. “T Spaces.” *IBM System Journal*, 37(3):454-474, 1998.