Customizable Data Distribution for Shared Data Spaces

Giovanni Russello, Michel Chaudron
Dept. of Mathematics and Computing Science
Eindhoven University of Technology

Maarten van Steen
Faculty of Science, Dept. of Computer Science
Vrije Universiteit Amsterdam

Keywords: Coordination, Component-based System, Separation of Concerns, Distributed Shared Data Space, Customization.

Abstract

To support component-based software engineering, simple and efficient mechanisms for dynamic composition and decomposition of components are needed. Shared data spaces are a simple composition mechanism, yet their efficient distributed implementation faces several complicating factors. One of these factors is that the communication needs of components may differ per data type, per application, and may even change over time. While existing data-space implementations treat all data equally, we propose a distributed data-space architecture that provides the means for differentiating distribution policies according to the type of data. Using this approach we are able to cater for the specific needs of the data. We maintain the transparency of the shared data space paradigm to the application programmer, but extend its capabilities for optimizing its efficiency.

1 Introduction

There are several forces that drive industry toward component-based development of software. From a development perspective, it is desirable to reuse existing software and to easily compose (existing) pieces of software in order to increase productivity and decrease time-to-market. From the application perspective, there is an increasing demand for flexibility in terms of adapting or extending the functionality of systems. Hence, a key technology in the successful deployment of software components is one that supports dynamic (de)composition of software components.

Generative communication, where applications communicate via a shared data space, is a promising composition technology because it provides referential and temporal decoupling between components [3]. This decoupling makes it possible to handle changes in the composition of systems in a way that is transparent to other components. The shared data-space model was introduced by the coordination language Linda [7].

The communication needs of the data that passes through a shared data space may differ per data-type, per application and may even change over time. In order to obtain an efficient distributed shared data space, the coordination infrastructure should cater for these different communication needs.

In this paper we present an architecture for a shared data space system that is able to tailor data distribution to data characteristics. The contributions of this paper are twofold. Firstly, we show that data distribution can be handled transparently to the application components. In this way, we achieve separation of concerns between functional requirements and non-functional requirements. Secondly, we demonstrate how the mechanism for enforcing distribution policies is carried out at runtime. This illustrates that the non-functional properties of the system can be changed during system deployment without the need for shutting the system down. As a consequence, writing application code is easier because programmers have only to concentrate on the basic functionality of the components. Furthermore, component reuse is enhanced, since the same component may be deployed in environments where different data distribution patterns are required.

Our approach to differentiating distribution policies differs from existing ones in that we can support any policy implementation as provided by a distribution designer. Moreover, implementations of distribution policies may be dynamically loaded into the system without the need for a shutdown and subsequent restart. Differentiating policies is not new, but has mainly been a subject of interest in distributed-shared memory systems (see, e.g., the work on Orca [2], Munin [5], TreadMarks [1], and more recently, InterWeave [6]). However, only recently, some work has been done in relation to shared data spaces [4], but is still directed towards computation-intensive applications and without the flexibility of runtime adaptations.

The paper is structured as follows. In Section 2 we present a case study that motivates the need for dedicated distribution patterns. Subsequently, in Section 3 we present our architecture. In Section 4 the case study is revisited using our model. We end with concluding remarks about future work.

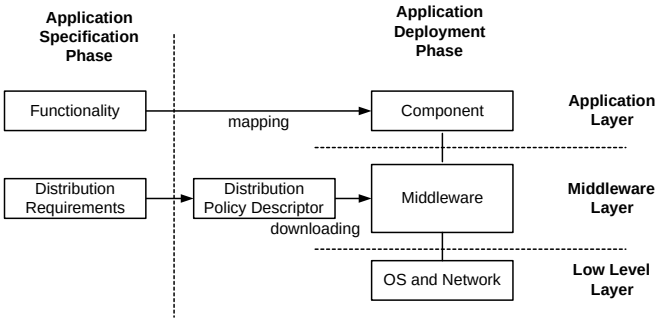

Figure 1. Our separation of concerns for component based systems.

2 A Case Study

In this section we present a case study that illustrates that different data types in a distributed system shows different distribution patterns. Our case study concerns in-home networks of consumer devices. Today, the number of chips that is embedded in devices in the home is increasing and so is the computational power that they offer. This allows devices to cooperate autonomously with each other to make the home environment more comfortable and configurable. Below, we discuss some scenarios that may occur in the context of our case study. For each scenario we describe some distribution patterns.

Profiling Housing Environment In this scenario, devices cooperate to recognize a person in a room and adapt the room environment to the preferences for that person (assuming these are stored as a profile somewhere in the system). For instance, setting the temperature in the room to a preferred value or starting to play favorite music are few actions that can adapt the room to the person's desires.

Fundamental in this scenario is the recognition phase. To this end, each room is equipped with a digital video camera. Each camera detects when someone enters the room and starts to collect images of the subject's face. This information is used by a *Face Recognition Application* (FRA). This application compares the data collected with the images stored in its database. When a match is found, the database returns the profile associated with that subject. Then, the profile is made available to all the other devices (such as, the heating controller, the CD player, etc.). In case no match is found, the FRA may set off a burglar alarm.

This scenario illustrates two data distribution patterns. One pattern is the *push-to-one* pattern for the image information: images are sent from all camera locations to the FRA location. Another pattern is the *push-to-all* pattern: data from the FRA is sent to all devices in the house.

Ubiquitous Message System In this scenario a user can leave a voice message for other persons that live in the house. A special device, the **Message Storing Device** (MSD) is designed to record and play voice messages. However, the user can request to listen to the received messages from another device, such as a PDA or a TV. It is desirable that the PDA can access the messages even if it is not connected to the home network. Thus, all messages are migrated and stored locally on the PDA.

By default, the messages are stored in the MSD. In case the messages are moved to another device and the device is still connected to the home network, new messages are sent to that device. When the device is not anymore reachable, then the default device, that is the MSD, is again used for storing messages.

The distribution pattern that data shows in this scenario is what we call *migrate-on-demand*: data is migrated to the PDA, if the PDA has declared an interest in it.

Service Discovering In this scenario, a rendering device searches in the home network for devices that provide services. For instance, a user wants to watch a movie on her PDA. The PDA searches the network for a device such as DVD player and requests the list of movies (*list of content*) that it can stream. The list of content is stored locally to the provider device. The PDA could cache locally the DVD player's list of content for fast access and displays the information to the user.

However, the DVD player may update its list of content, for instance, when new movies are added. This update requires that all the cached copies are invalidated. If the user wants to see the list, the PDA has to request and cache the new list of content and display the updated information to the user.

In this scenario the data that represents the list of content shows a *cache-on-demand* pattern, that is, the data is cached on the location of the requester only when it is requested. Moreover, an invalidation mechanism is necessary to prevent outdated data being returned after an update.

3 The Architecture of a Distributed Shared Data Space System

This section describes the architecture of our distributed shared data space system. The system was designed to support separation of component functionality from non-functional issues concerning data distribution.

3.1 Conceptual View

Figure 1 depicts the approach that we propose for system design and implementation. Each application is specified as

a combination of a definition of its functionality and a definition of its data distribution requirements. Functional specifications are mapped onto *components* which may be distributed onto various nodes of a distributed system. Distribution requirements are converted into a *distribution policy descriptor* that is downloaded into the middleware where it is interpreted at runtime.

The middleware provides the application components a set of operations for communicating with the shared data space. Moreover, the middleware is responsible for interpreting an application's distribution policy descriptor and for enforcing application-specific policies.

In the next section we describe the actual implementation of our model. Currently, we focus on data distribution. We aim to deal with timeliness concerns in similar ways.

3.2 Implementation View

We have named our system **GSpace**. Conceptually, applications view GSpace as a single data space. However, GSpace is implemented by several *GSpace kernels* distributed across multiple nodes (see Figure 2). A GSpace kernel provides a local storage service plus some means to communicate with other GSpace kernels. Application components may be placed on the same node where a GSpace kernel is running, as is the case with node 1 in Figure 2. Alternatively, if an application components resides on a node with limited computational resources, then GSpace may be located on another node, as in node 3. In this latter case, a proxy is used to bind a remote GSpace kernel to components. GSpace kernels have their own threads and exist independently from application component resources. This is an important design property because it enables temporal decoupling. In other words, components do not need to be active at the same time to exchange data.

Data in a space are represented as *tuples*. A tuple is an ordered collection of typed fields and each field has a value associated with it. Components can insert tuples in a space and retrieve tuples in an associative manner by matching with a *query*.

There are many design decisions that can be made (see also [9]). We focus on two issues related to the partitioning and distribution of the data space: (1) how do we partition the data, i.e., which tuple is stored where, and (2) when, how, and where to are tuples moved.

In our system we assume that the tuples in the data space are *typed*. We support partitioning of data based on these types as well as on their content. In our Java implementation, the tuples are typed by their *class*. To obtain an efficient distributed shared data space, we treat tuples of the same class and same content according to the same distribution policy. In other words, tuples of different types may be distributed according to very different strategies. This is

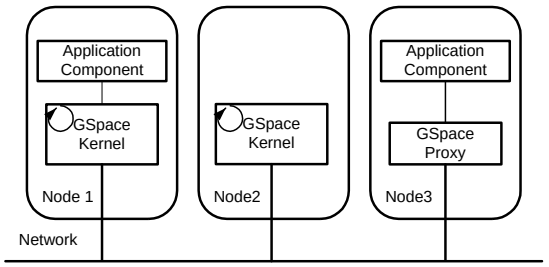

Figure 2. GSpace kernel and application components placement across several nodes.

an important distinction with the traditional way of distribution in shared data spaces, which is often realized through distributed hashing techniques [12]. It is furthermore important to note that this distribution of GSpace tuples across multiple nodes is completely transparent to the application components.

The internal architecture of GSpace is shown in Figure 3. Internally, each GSpace kernel is organized as follows.

- The **System Boot** module is responsible for initiating all the other modules of GSpace kernel. Subsequently, it advertises its presence to other GSpace nodes in order to establish communication channels and join a GSpace group.
- The **Controller** provides the following operations to components.
  - `put(t)` inserts tuple `t` in the space.
  - `read(q)` if there is a tuple in the data space that matches the query `q`, then a copy of this tuple is returned to the calling component. The `read` operation blocks the caller component until a tuple matching the query is available in the system.
  - `take(q)` is similar to the `read` operation with the difference that the matching tuple is removed from the space.
- The **Dynamic Invocation Handler** (DIH) determines which distribution policy to apply based on the type and content of the tuple (or query). It operates as follows. If a component invokes a GSpace operation, then the DIH looks up which policy to use for this operations. This information is obtained from a **Distribution Policy Descriptor**. A distribution policy descriptor is a file containing *(template, distribution policy)*–pairs. Tuples and queries are matched against templates to determine which distribution policy to apply.

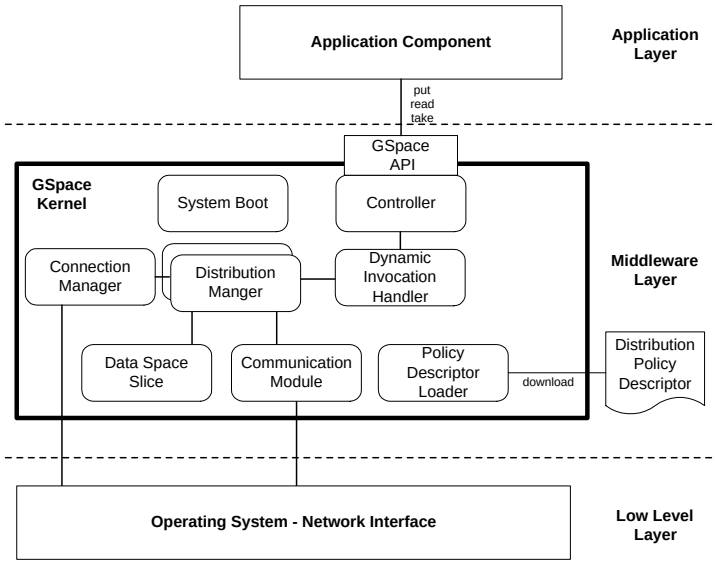

Figure 3. Internal structure of a GSpace kernel on one node.

Distribution Manager	Description
Store locally	A tuple is always stored in the local slice. Consuming operations (i.e. read or take) are performed on the local slice. If the tuple is not found locally then the request is forwarded to other nodes.
Push-to-all	A `put` operation forwards a tuple to all known nodes. `Read` and `take` are performed on the local slice.
Push-to-one	A `put` operation forwards a tuple to one specific slice. `Read` and `take` are performed on this specific slice.
Push-where-needed	A network of GSpace nodes can be partitioned into sets S_i. Each set is then associated with a template. For instance, if set S_i is associated with template t_i a tuple matching t_i is inserted in the local slices of all nodes in S_i. Consuming operations are forwarded to the nodes in the set. Particular care should be provided for a `take` operation because a tuple must be removed atomically from all nodes in the set.
Migrate-on-demand	Tuples of type t are stored in one location, called $home-location_t$. When a `read` or `take` operation is performed from another location, all tuples of type t are migrated to the requester location, that becomes the $home-location_t$. In case that the $home-location_t$ is not reachable then a default location is used.
Cache-on-demand	Tuples are stored locally. When a remote location performs a `read` operation, a copy of the tuple is cached on the local slice of the requester location. When a cached tuple is removed through a `take` operation then an invalidation message is sent to invalidate all the cached copies of that tuple.

Figure 4. A list of several Distribution Managers available in GSpace.

Once the applicable policy has been determined, the DIH ensures that a distribution manager that implements this policy is instantiated [1]. This means that it is possible to instantiate a class and invoke its methods at runtime without knowing the class at the time the code was written. The benefits of this methodology are twofold. First of all, it is not necessary to load at bootstrap time all the available distribution managers, but only the ones requested by a distribution policy descriptor. Moreover, new implementations of distribution managers can be added and used at runtime, without recompiling the source code of the GSpace kernel.

- The **Distribution Managers** are responsible for enforcing distribution policies. For each distribution policy that the system supports there is a separate distribution manager. Depending on the policy that the manager implements, it may dictate that tuples be sent to or requested from GSpace kernels on other nodes. Communication between distribution manager is realized by the communication modules.
- The **Data Space Slice** provides a local storage for tuples together with the associative method for retrieving them.
- The **Communication Module** provides facilities for sending tuples to and receiving tuples from other GSpace kernels. This module provides support for different forms of communication (such as multicasting or point-to-point communication), but also different qualities of service (such as reliable or unreliable communication) to be used for defining different policies.
- The **Connection Manager** is responsible for keeping track of information about network locations of other GSpace nodes that can be used by distribution managers. We currently foresee the use of GSpace on local-area networks and assume the availability of an efficient mechanism for multicasting communication, making it much easier to discover the locations of other GSpace nodes.
- At bootstrap time, the **Policy Descriptor Loader** downloads distribution policy descriptor file. The file is made available to all nodes where GSpace kernels are instantiated. The descriptor may be changed at runtime. Thus, this module monitors for updates of the local descriptor file and reloads it if necessary.

GSpace is partly based on the approach followed in the Globe distributed system [11] in which each distributed object encapsulates state, operations on that state, but also the implementation of a policy that dictates how state and operations are distributed across multiple machines. GSpace and Globe have in common that distribution policies can be differentiated between objects. In GSpace, differentiation may take place based on object *types* or on content; in Globe differentiation is done between objects.

We have implemented GSpace on a local-area network and designed several distribution managers, listed in Table 4. For brevity, we mention only a few distribution managers that we developed. However, as we mentioned, part of the novelty of GSpace lies in the fact that we allow users to provide *arbitrary* policies and download them into the middleware while the system is in operation.

4 The Case Study revisited using GSpace

In this section we show how GSpace can be used as communication middleware for the in-home networking case study. We assume that each home device has some power of computation and some means to communicate and that devices may be connected to the in-house LAN using either wired or wireless connections. Application components running on devices may provide GUIs to users. GSpace kernels are instantiated on several devices, forming the shared data space of the house, as shown in Figure 5.

Now, let us describe how GSpace is used to tailor distribution patterns of data for some of the scenarios introduced in Section 2.

Figure 6 shows a Message Sequence Chart that illustrates a migrate-on-demand pattern of the Ubiquitous Message System scenario. The application component `MessageRecorder` running on the MSD stores tuples representing messages in the local GSpace kernel. The `MessagePlayer` running on the PDA queries the GSpace kernel on the PDA for Michel's messages. The kernel forwards the request to the kernel on the MSD. The latter replies by moving all the tuples matching the query to the PDA kernel. When the `MessagePlayer` on the PDA queries the space the next time, the kernel on the PDA can return a tuple directly.

Figure 7 depicts a Message Sequence Chart for a migrate-on-demand pattern for the Service Discovering scenario. In this case, the component `ListManager` on the DVD Player puts the list of content into the shared data space. This list is stored in the local slice. When a rendering device, such as a TV set, queries for the list of content, its local GSpace kernel forwards the request to the kernel running on the DVD Player. Then, a copy of the tuple representing the list of content is sent to the requester's kernel, which stores it on the local space before returning it to the application component. When the list of content is updated, the old tuple is removed and invalidation messages are sent to the node where it was cached. The new tuple is put in the

[1]The DIH uses the Java Dynamic Proxy Class mechanism to dynamically instantiate distribution managers and invoke their methods. A dynamic proxy class is a special class created at runtime by the Java virtual machine.

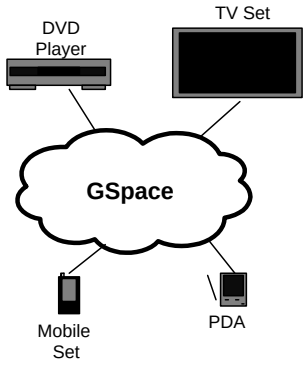

Figure 5. A GSpace composes devices in an in-home environment.

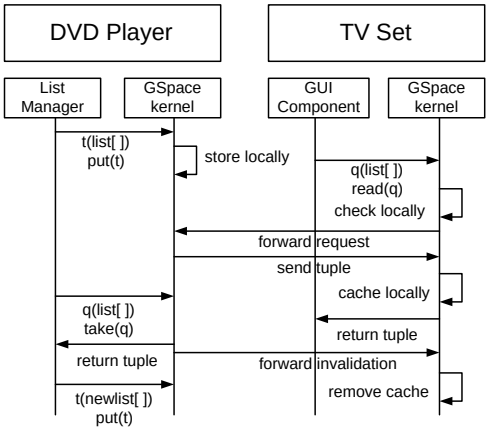

Figure 7. Message SequenceChart for the Service Discovery Scenario.

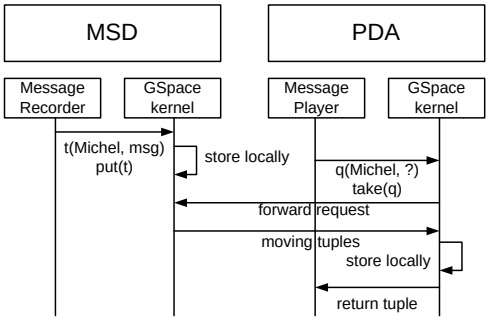

Figure 6. Message Sequence Chart for the Ubiquitous Message System Scenario.

space. The next time the application component requests that tuple, the kernel has to forward a message to receive the updated version.

These example scenario's illustrate that using our approach it is possible to implement complex distribution policies, yet without interweaving distribution requirements with the functionality of application components.

It may be an objective of an application to store data persistently in GSpace. Then, depending on the capabilities of the devices, a distribution policy should be used that stores data on devices that are unlikely to leave the network. Also, the policy may impose some degree of replication of data across multiple devices.

5 Discussion

The development of GSpace is still in a preliminary phase. At present, policies are described implicitly by means of the implementation of a corresponding distribution manager. An application developer simply identifies for each template which distribution manager is to be used. This manual specification should preferably be supported by means of a policy description language that could also cater for conflict detection. For example, templates can generally be organized as a partially ordered set in which $t_1 < t_2$ if template t_1 is more specific than t_2. Currently, we demand that each template can be an immediate specialization of at most one other template, effectively enforcing a tree structure. This prevents conflicts in distribution policies.

Although we have not substantiated out claim that differentiating distribution policies will lead to a better performing shared data space, preliminary experiments actually do indicate the validity of our claim. In particular, if we construct applications with different put/get/take ratios, it turns out that all the policies described in Figure 4 show to be relevant. However, the research described in this paper was partly motivated by the outcomes of research concerning differentiation of replication and distribution strategies for Web documents, as described in [8]. There, it was shown that if a Web document could be assigned its own distribution strategy, global performance optimization could be achieved. We expect similar results for GSpace.

6 Concluding Remarks and Future Work

Component-based systems are well served with the temporal and referential decoupling provided by the shared data space model. Hence the shared data space provides a suitable mechanism for (de)composing components.

In this paper, we have presented a model for a data space-based coordination system that supports dynamic component-based applications. This system separates functional and distribution concerns. We show how distribution policies can be customized to the communication needs of particular data types. This customization enables the efficient distributed implementation of a shared data space, while maintaining its simple programming model. Furthermore, a special feature of the GSpace architecture is that, in addition to catering for changes in the functionality of the system, it allows the run-time modification of non-functional properties. The effectiveness of this approach is confirmed by a number of experiments which we have performed using a Java prototype in [10].

Moreover, the separation of functional and distribution concerns has the following advantages:

- The potential for component reuse is enhanced, since components need not be obfuscated with (hard-wired) distribution policies. As a result, they may be deployed in environments where different data distribution patterns are required.
- The job of the component designer is simplified. He does not need to incorporate distribution requirements into his component.
- Distribution policies become a unit of reuse; i.e. they can be reused by many applications.
- Typically the design of distribution aspects of an application is error-prone. Separating distribution policies as unit of reuse increases the quality because it can be designed by distribution-experts and be proven by field testing.

As a next step, we plan to investigate separation of other non-functional concerns such as global scheduling of timing constraints, error handling and security. Furthermore, we are working on reflexive mechanisms for the automatic runtime selection of policies. The latter touches upon one of the challenging issues for GSpace: the automatic adaptation of policies when application behavior changes. Ideally, the GSpace system obtains run-time information on the way that classes of tuples are used and then automatically and transparently adapts the policy for each class of tuples separately to optimize their distribution. Preliminary experiments indicate that such an adaptation is indeed feasible.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] H. Bal and M. Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Eighth OOPSLA*, pages 162–177, Washington, DC, Sept. 1993. ACM.

[3] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile-Agent Cooordination Models for Internet Applications. *IEEE Computer*, 33(2):82–89, Feb. 2000.

[4] J. Carreira. *Researching the Tuple Space Paradigm in Parallel Programming*. PhD thesis, University of Coimbra, 1998.

[5] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Trans. Comp. Syst.*, 13(3):205–244, Aug. 1995.

[6] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. InterWeave: A Middleware System for Distributed Shared State. In *Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*. ACM, May 2000.

[7] D. Gelernter. Generative Communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.

[8] G. Pierre, M. van Steen, and A. Tanenbaum. Dynamically Selecting Optimal Distribution Strategies for Web Documents. *IEEE Trans. Comp.*, 51(6):637–651, June 2002.

[9] A. Rowstron. Run-time Systems for Coordination. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies and Applications*, pages 78–96. Springer-Verlag, Berlin, 2001.

[10] G. Russello, M. Chaudron, and M. van Steen. Separting Distribution Policies in a Shared Data Space System. Technical Report IR-497, Vrije Universiteit, Department of Mathematics and Computer Science, June 2002.

[11] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, Jan. 1999.

[12] G. Wilson, editor. *Linda-Like Systems and Their Implementation*. Edinburgh Parallel Computing Centre, University of Edinburgh, Edinburgh, June 1991.