Separating Distribution Policies in a Shared Data Space System

Giovanni Russello (**contact**)
Michel Chaudron
Eindhoven University of Technology
{g.russello,m.r.v.chaudron}@tue.nl

Maarten van Steen
Vrije Universiteit Amsterdam
steen@cs.vu.nl

June 6, 2002

Abstract

To more fully exploit its benefits, the principle of separation of concerns should be built into the methods of designing systems as well as into the techniques that we use for implementation. We argue that for distributed applications, separating concerns should be supported by the middleware on which the applications are executed. In particular, we argue that the middleware should provide the proper mechanisms to enforce application-specific policies for different concerns. Interweaving of policies should take place at run time such that it becomes possible to change them without the need for recompiling the application source code.

To illustrate that this approach is principally feasible, we present a design for a middleware based on the shared data space paradigm. Here distribution is taken as an example of a concern that can be treated separately from other concerns and transparently to applications. We present a case study that shows how we can vary the distribution policy for an application. The case study shows a trade-off between the use of bandwidth versus the speed of message delivery.

1 Introduction

The shared data space model was introduced as a paradigm for coordinating parallel and distributed applications. It falls in the category of software that is nowadays called *middleware*.

Coordination through shared data spaces, also called generative communication, forms an attractive model for developing distributed and component-oriented systems as it supports referential and temporal decoupling of processes [8].

Unfortunately, providing efficient implementations of the shared data space model shows to be difficult. This is especially so when dealing with large-scale systems in which components can be numerous and placed far apart (see also [16]). Matters are further complicated when, for example, timing constraints need to be taken into account. Attaining efficiency is difficult mainly due to the combination of distribution and the associative nature of the generative communication model. Generative communication essentially introduces a search through a data space in order to match data to some given template. In a distributed system, such a search will generally involve searching through data that is spread across multiple machines.

In this paper, we argue that efficiency can be attained by taking into account the patterns by which applications produce and consume data. Instead of seeking general solutions, we argue that the middleware should provide the mechanisms by which various application-specific solutions can easily be supported. This idea is not new and has been extensively investigated for parallel computation. The novelty of our approach lies in the way that we separate the implementation of distribution policies from the functionality of the application.

In particular, the main contribution of the research described in this paper is that we provide a means by which we can separate the distribution of data in a coordination-based system from the way that coordination is used by applications. We provide a simple mechanism that allows us to differentiate and enforce various distribution policies in an application-specific manner.

The structure of this paper is as follows. In Section 2 we introduce our ideas on how the separation of concerns principle can be applied to the design and implementation of distributed systems. Also, we explain the role that coordination plays in the realization of this approach. In Section 3 we present our design for the data distribution mechanism. Subsequently, in Section 4, we present an experimental case study to illustrate our approach. We finish the paper by discussing related work and conclusions.

2 Background

2.1 Separation of Concerns

The general aim of our research is to provide support for the development and deployment of distributed embedded real-time applications. What often makes these applications so complex is that they need to exhibit specific behavior that is inherently strongly dependent on external, and often unstable factors. As a consequence, much effort is put into developing program segments by which the effects of these external factors become more controlled. Typical examples include program segments for failure handling and segments for data distribution to meet performance demands.

These program segments are often intertwined with segments that implement the core functionality of an application. However, it has been recognized that separating different concerns is essential for developing applications that strongly depend on their environment or that can be expected to continuously evolve in the course of time.

Separating concerns is explicitly addressed in aspect-oriented programming (AOP) where it is often addressed at the level of programming languages. The part in AOP that is relevant to our research is the technique used to interweave core functionality with aspect descriptions. Pre-compiler techniques are often used to produce a single piece of target code [12]. If one aspect needs to be redefined, it is necessary to perform the interweaving process again, that is, to recompile the program. However, the currently ongoing research in AOP is pointing to dynamic weaving environments where aspect can be changed without recompiling all the code. We argue that for distributed component-based applications, separating concerns should also be supported by the middleware on which the applications are executed. In particular, we argue that the middleware should provide the proper mechanisms to enforce application-specific policies for different concerns. Interweaving of policies should take place at run time such that it becomes possible to change policies without the need for recompiling the application source code.

Figure 1 depicts the approach that we propose for system design and implementation. Each application is specified in terms of its functionality along with a number of aspects. Functional specifications are transformed into *components* that are subsequently mapped onto the various nodes that make up the distributed system. Aspects are transformed into *policy descriptors* that are downloaded into the middleware where they are interpreted at run time by various *aspect managers*.

Besides having application-specific policy descriptors, we also support the specification of global, systemwide aspects. These aspects are transformed into application-independent policy descriptors.

The middleware itself is accordingly divided into two layers. The highest layer is responsible for interpreting an application's policy descriptors and thus for enforcing application-specific policies. The lowest layer is responsible for dealing with systemwide, application-independent descriptors by which systemwide policies can be enforced. In this way each application can define its own policies for aspects such as for distribution of data or for dealing with time constraints without interfering with policies of other applications. It is also possible to define and enforce policies that apply to all applications.

Figure 1 presents the case in which two applications, A and B, are specified and deployed. We use light grey blocks for application A and dashed line blocks for

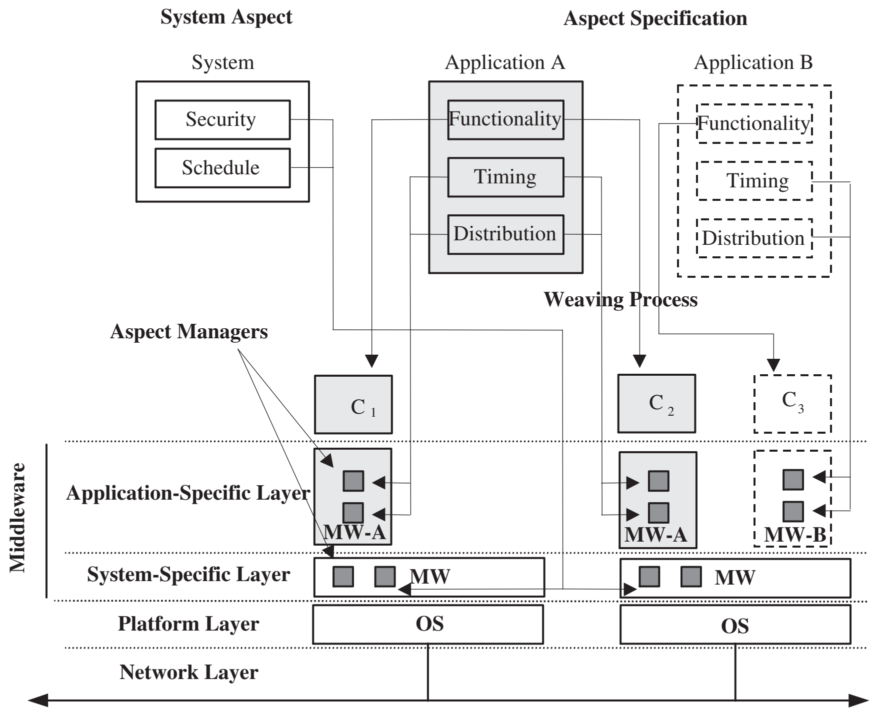

Figure 1: Our aspect-oriented approach for embedded distributed real-time applications.

application B. For both applications, functional specifications are transformed to a set of components. For A this leads to components C1 and C2, while component C3 belongs to B. Aspect descriptions for either application are downloaded as policy descriptors into the application-specific layer of the middleware. Each application can thus define its own policies independent from other applications and from the systemwide policies. The figure also shows the integration with systemwide policy descriptors, such as those used for global scheduling or security. These descriptors are downloaded into the middleware at the level of the system layer.

2.2 Separation by Coordination

Instead of seeking solutions for aspect-oriented programming in general, we adopt coordination as the basis for distribution of data, processes, and control. Fundamental to coordination-based systems is the separation between a computation model and a coordination model. The computation model captures application

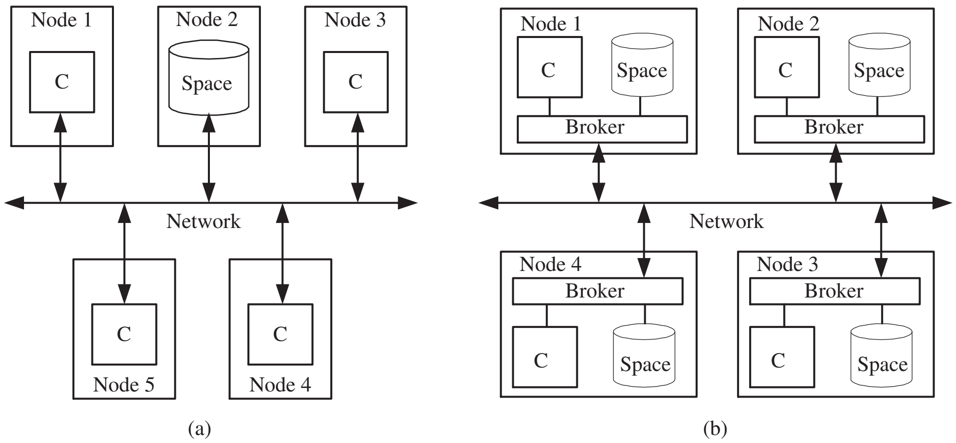

Figure 2: (a) Configuration with distributed applications and centralized tuple space, and (b) one with a distributed tuple space.

functionality; the coordination model links the various application *activities* by providing mechanisms for communication and synchronization. One of the first examples of a coordination model came in the form of generative communication as implemented in Linda [7]. There are currently many different kinds of coordination models [14].

An important advantage of coordination-based systems is that processes are referentially and temporary uncoupled [2]. This uncoupling is a prerequisite for any model that aims to support separation of these concerns. Coordination has also been successfully adopted for real-time applications, as is witnessed by various commercial distributed publish/subscribe systems (e.g., SPLICE [1]).

In the coordination-based systems that we consider, communication between components is facilitated by means of a *tuple space*. The tuple space contains tuples that represent data items. Components that are connected to the tuple space can *produce* and *consume* tuples. Consuming a tuple requires that the consuming component provides a *template* that is used to match against tuples in the space. Consuming thus generally requires an associative searching technique. As it turns out, general-purpose distributed associative searching is difficult to implement. For this reason, many systems implement a tuple space by means of a single server, as shown in Figure 2(a).

An important drawback of this centralized approach is that we may run into serious performance and scalability problems when components are placed far apart, or when (part of) the network offers only limited quality of service. The solution to these problems is to partition and replicate the tuple space across multiple servers.

This approach has been widely applied in the parallel programming community where efficient implementations of the Linda tuple space have been sought by statically distributing tuples using compile-time analysis. When dealing with real-time applications, we argue that better results can be achieved if coordination is combined with aspect-oriented techniques as described above. In particular, we believe that by separately specifying distribution aspects for tuples it is possible to derive an efficient implementation of the coordination between components. How we can come to such implementations is the main topic of this paper and is discussed in Section 3.

3 GSpace: A Distributed JavaSpace

We aim to build our coordination system incrementally. At this stage we want to focus on distribution policies. Therefore, where possible, we use existing components for parts that can be separated from the distribution aspects of our system. We are currently using JavaSpace, a tuple-space implementation using Java technology as the basis for supporting application development [6]. This enables us to focus on distribution issues, while using an existing associative matching mechanism. In this section, we explain how we have implemented a distributed version of JavaSpace, in which the combination of efficiency and distribution transparency is achieved through our approach as discussed above.

We have named our system *GSpace*. GSpace is partly based on the approach followed in the Globe distributed system [17] 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 takes place between object types; in Globe (in which objects are fairly course grained), differentiation is done between objects.

The section is structured as follows: we give a brief overview of JavaSpace. Then we present our approach to address data distribution and some policies that are supported. This section ends with a detailed description of the architectural characteristics of our current middleware system.

3.1 JavaSpace Summary

JavaSpace provides a relatively simple application programming interface (API) for the storage and retrieval of tuples. The core JavaSpace API consists of two in-

Method	Description
write	Writes a serialized copy of the given entry object into a JavaSpace, where it is kept until a specified lease expires.
read	Look up an entry object by means of a template, and block the caller until a match has been made. A copy of the entry object is returned to the caller.
readIfExists	Look up an entry object, but block the caller only if a matching object is found, which is then copied and returned to the caller.
take	Look up an entry object, and block the caller until a match is made. Return the matching object, and remove it from the JavaSpace.
takeIfExists	Look up an entry object, but block the caller only if a matching object is found, which is then removed from the JavaSpace and returned to the caller.
notify	Register that the caller wants to receive a notification when an entry object matching the given template is written to the JavaSpace.

Figure 3: The most important methods available through the JavaSpace interface.

terfaces: JavaSpace and Entry. The JavaSpace interface provides the methods for creating a JavaSpace and for inserting, reading and removing tuples from the space (see Figure 3). In JavaSpace terminology, a tuple is called an *entry*. Looking up an entry is done by defining a *template* and supplying this to the read or take methods. The core JavaSpace API has no reference to distribution aspects whatsoever; in this sense, the interface is distribution transparent.

3.2 Implementation Support for Differentiating Distribution Policies

In Section 2.2 we discussed the drawbacks of a centralized implementation of a shared data space. The alternative is to distribute the implementation across multiple nodes, as Figure 2(b) shows. In this configuration, the *broker* mediates the communication between components and data space in the same node, and between data spaces among different nodes.

There are many design decisions that can be made (see also [16]). We focus on the following issues related to the partitioning and distribution of the data space.

- How do we partition the data, that is, which entry is stored on which node?
- When, how, and where to are entries moved?

In our system we assume that the entries in the data space are *typed*. We support partitioning of data based on these types. In our Java implementation, the entries are typed by their *class*. In other words, we assume that in order to obtain an efficient distributed implementation of a JavaSpace, we can treat entries of the same class according to the same distribution policy. Using Java's reflection capabilities, by which we can inspect an object's class at run time, it then becomes possible to do a run-time lookup of a distribution policy for a given entry.

We currently support the specification of only simple distribution policies. For example, a developer can specify, per entry class, at which servers entries of that class are to be stored. To that end, each component specifies whether it is a producer or a consumer (or both) of entries of a given entry class. This approach is similar to the publish/subscribe style of communication. A component that produces data is a publisher and a component that consumes data is a subscriber.

We consider the following distribution policies:

- *Pull policy*: Data is stored at the node running the producer. Data is sent to the node where a consumer resides when that consumer performs a read command (for this type of data).

- *Push policy*: Immediately after its production, data is forwarded to the nodes where the consumer(s) for this type of data reside.

Also, we allow to specify whether the transfer of entries between producers and consumers is reliable or unreliable. A reliable transfer means that each entry object produced is guaranteed to be delivered at the consumer side. In contrast, with unreliable transfer entry objects may not arrive at the consumer side. Clearly, it strongly depends on the nature and use of the data whether such a policy is realistic or sensible.

These policies are fairly basic and more elaborate distribution policies need to be supported as well. This is subject of future research. However, we use these simple policies to show that our approach is feasible.

3.3 GSpace Middleware

Next, we concentrate on the design and implementation of GSpace. The basic architecture of GSpace is shown in Figure 4. It is organized as a middleware layer between application components and JavaSpace servers. Each application component has its own GSpace broker. The GSpace broker mediates between application components and JavaSpace. In theory each GSpace broker can have its own JavaSpace. However, to reduce resource use, we instantiate a single JavaSpace server for each node. This JavaSpace server can be used by all GSpace brokers running on the same node.

As we explained, the distribution itself is performed in a way similar to publish/subscribe systems, where components tell whether they act as publishers or

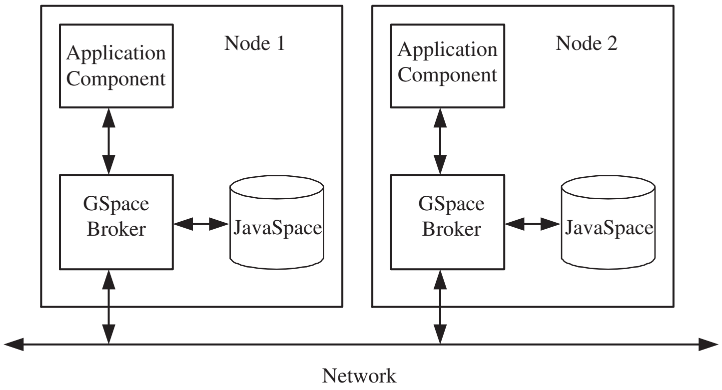

Figure 4: System architecture used in GSpace.

Method	Description
put	Insert a new entry object and the system decides where to store it. If an old entry of the same type is already present in the destination space it will be overwritten by the new one.
get	Look up an entry object by means of a template, and block the caller until a match has been made. The entry is removed from the space in which it was stored.
producer	Declare that a component is connecting to the system as producer of a specified entry type.
consumer	Declare that a component is connecting to the system as consumer of a specified entry type.

Figure 5: The interface GSpace provides to applications.

subscribers of specific entry classes. The GSpace broker offers the interface in Figure 5 to applications. The semantics of the operations *put* and *get* are insertion and retrieval of a entry from the data space.

The way in which these operations are implemented is that an entry is removed from the data space when it is retrieved and insertion overwrites any existing entry with the same data type. However, the semantics leaves room to vary the implementation of these operations. For example, if it is important to receive multiple issues of the same entry type, then the *put* primitive should be implemented as a simple *write* operation. At present, our system allows to specify these semantics separately as well. However, we still need to explore the usefulness of its applicability.

The GSpace interface is independent from the implementation of the data space used to store messages that components exchange. Hence GSpace is not obliged

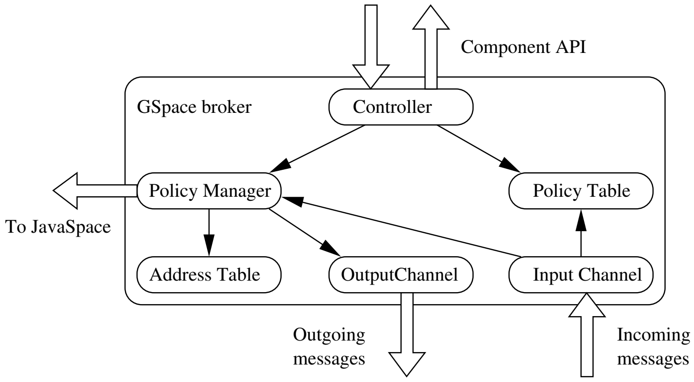

Figure 6: Internal structure in a GSpace node.

to use JavaSpace. We foresee to implement our own version of a data space to be used with GSpace. Moreover GSpace can be used as a mechanism for achieving interoperability between different data space implementations.

It is important to note that the distribution of GSpace entries across multiple nodes is completely transparent to the application components.

Internally, each GSpace broker is organized as shown in Figure 6. There are four internal modules and two different tables.

Controller: This module provides the GSpace API to applications. When an application invokes a *put* or *get* method, the controller determines which distribution policy to apply based on the type of the entry that is passed. This information is obtained from the **policy table**, which contains *(entry type, distribution policy)*–pairs. After it has determined which policy to apply, it invokes the proper **policy manager** to enforce that policy.

Policy manager: For each distribution policy, there is a separate policy manager responsible for enforcing that policy. A policy manager provides three groups of methods:

- A method for storing entries in GSpace
- Methods for looking up and retrieving entries from GSpace
- A method for processing an entry when it is received from a remote host.

When an entry needs to be stored at or retrieved from a remote node, the policy manager must determine the host address of that node. For this reason, an *address table* is used to store the locations of producers and consumers. This table is updated using a standard publish/subscribe approach. When a component joins the system it can declare itself to be a producer or consumer for a specific entry type. This information is then broadcast and subsequently stored in the address table of each GSpace node. When a component leaves the system, the controller initiates an update of the address tables.

Output channel: The output channel implements an interface for sending entries to other GSpace nodes. 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.

Input channel: This module handles incoming entries from other GSpace nodes. When an entry arrives, the input channel extracts the entry's type and looks up its associated distribution policy in the policy table. Using this information, the channel then selects the appropriate policy manager to which it then passes the entry for further processing.

The policies that the application developer selects are stored in a file that is made available to each GSpace node before starting the application. The file is read by all nodes, allowing each node to initialize its policy table by mapping each user-defined type to the appropriate policy manager. We plan to replace this way of specifying and processing distribution policies by a more elaborate means. However, it should be clear that the principles behind our separation of concerns approach are relatively simple and easy to realize.

4 Experiments and Results

As described before, the question we research is whether it is possible to attain efficiency by tuning distribution policies in an application transparency way. To this end, we built a middleware system that supports several distribution models, separating functional from non-functional requirements. In theory, applying a distribution policy that fits with the data characteristics should lead to better performance if compared to a model in which it is possible to apply only a fixed policy. On the other hand, overhead is introduced related to finding and applying the right policy for each message, when compared to fixed-policy middleware.

In this section we present a case study that we implemented using GSpace and we compare the results with an implementation of the same case on top of a

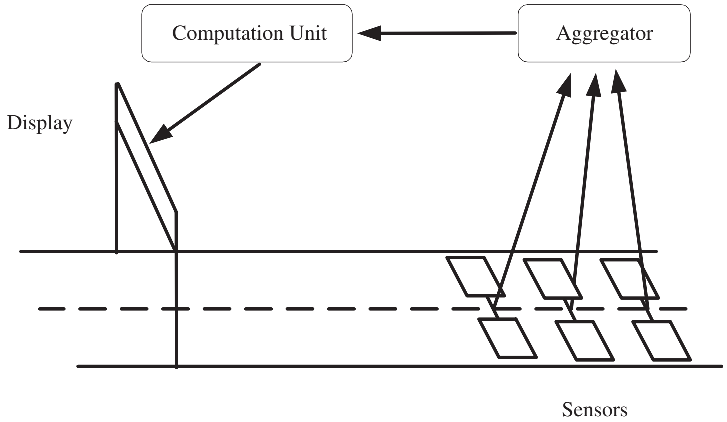

Figure 7: Traffic Management Point

traditional, that is, centralized JavaSpace. We also compare the performance of different distribution policies. Performance is expressed in terms of turnaround time and (application) bandwidth.

4.1 Traffic Management Point (TMP)

The case study that we propose is a model of a Traffic Management Point (*TMP*) used in the Dutch motorways. To reduce traffic jams or to avoid accidents along the motorway there are displays that inform drivers about the current traffic situation. To this end, information sensors are placed inside the road surface. Figure 7 gives more details on the TMP structure. A TMP consists of the following units:

Sensor: These units measures the road status (for instance traffic speed, fog presence, current temperature etc.).

Aggregator: This unit computes averages of consecutive samples of sensor data.

Computation Unit (CU): Based on average values, this unit computes a speed advice.

Display: Displays the speed advice.

For our experiments, we simulated a simple model of a TMP on an (100 Mb/s) Ethernet LAN consisting of three PCs (Pentium III-400). For each unit present in the TMP structure we implemented a component.

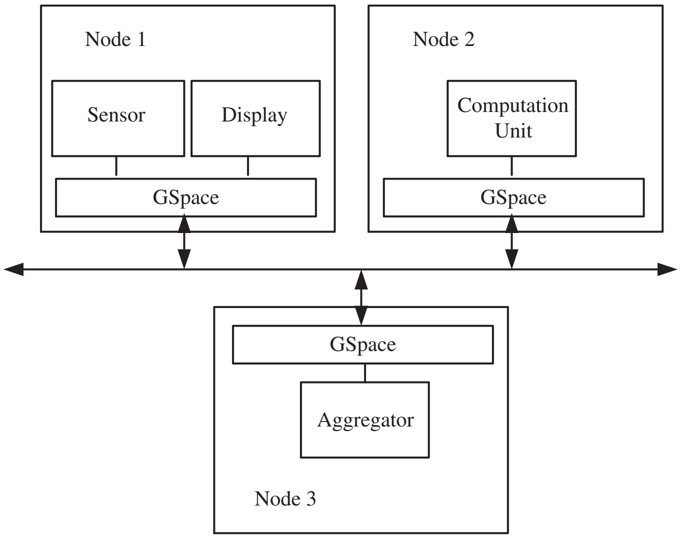

Figure 8: TMP component deployment using GSpace

4.2 Examples of Distribution Policies

In this section we present an implementation of TMP that uses GSpace and we introduce two examples of distribution policies. We compare the performance of the policies using metrics for bandwidth use and turnaround time.

Figure 8 shows the deployment of components in the distributed GSpace implementation. Brokers and data spaces are instantiated on each node and share the same resources with the application components.

The TMP case study presents classical real-time behavior. There are sensors to collect information about the environment. An application collects sensor data, performs some computations and produces data that other units (such as actuators or displays) can use to modify the environment. Each unit has its own pattern of input/output behavior.

In the experiment, we used two settings for distributing application messages. In the first setting all the data is pushed from the producers of data to consumers of data; e.g. from sensors to the aggregator. This policy leads to the behavior depicted in Figure 9(a).

In the second setting, GSpace broker sends the data produced by the sensor to the aggregator node only after the aggregator has performed a *get* operation for that data. The other types of messages are pushed. This policy is illustrated in Figure 9(b).

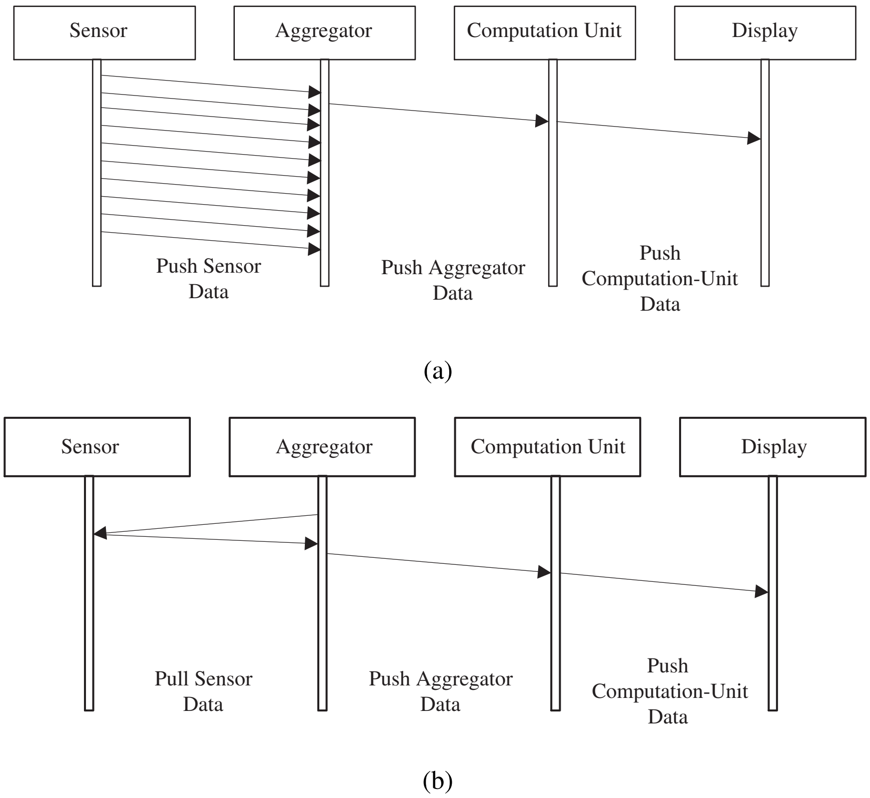

Figure 9: The message flow in (a) the push setting and (b) the pull setting.

4.2.1 Quality of Service

We investigate how changing the quality of service (*QoS)*), that the system offers, can affect performances according to the metrics that we used in our experiments. In the case study presented the QoS can be expressed as the frequency at which the data on the display is refreshed.

We run a series of experiments where we fix the sensor speed at 20 messages per second and successively increase the speed by which the aggregator consumes (and correspondingly produces) messages. We start with an aggregator speed of one message per second and increase its speed up to 40 messages per second (which is twice the production speed of the sensor). The other components consume messages at the same speed as the aggregator. The experiment is terminated after the sensor has produced 1000 messages.

In the next sections we present the effects of this experiment on the turnaround

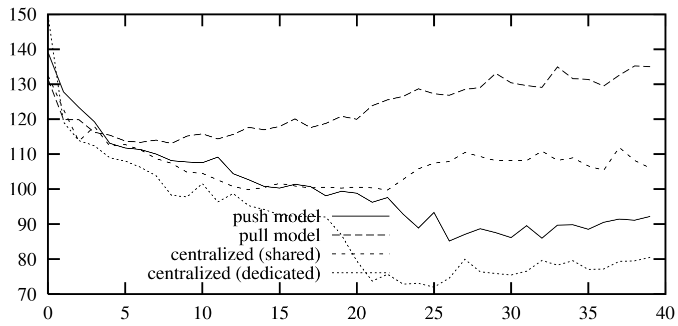

Figure 10: Total turnaround time (in ms) measured in relation to the number of sensor values (per second) consumed by the Aggregator.

time and on bandwidth.

4.2.2 Turnaround Time Measurement

We define the turnaround time as the difference between the time of production of a message by a sensor and the time this message is consumed by the display.

The graph in Figure 10 shows the result of the measurements. The horizontal axis represents the aggregator speed given as the number of messages consumed per second. The vertical axis gives the turnaround time in milliseconds.

The results show that for the pull setting the turnaround time is higher than in the push setting (apart from small aggregator speed value). This is due to two aspects of the pull setting. First, sensor messages spend some time stored at the node of the sensor before they are pulled to the aggregator node. Second, the JavaSpace instance running on Node 1 (see Figure 8), that GSpace uses to store and retrieve application data, is overloaded. In fact, this JavaSpace has to process requests performed by all the application components[1]: the sensor data are stored locally in it and the aggregator requests for the sensor data are redirected to this JavaSpace server (according to the pull setting). The computation unit data are pushed to this server, and the get operations executed by the display are performed on it. Since this JavaSpace server shares the resources (CPU time, memory access) with other processes (sensor and display components), its response time is very low. Both these side-effects negatively affect the turnaround time.

[1]Note that, in this setting not all the requests are sent to the JavaSpace running on Node 1. The JavaSpace on Node 2 receives also requests: The aggregator messages are stored in this space and also the computation unit requests for this data type are directed to this space. Instead, the JavaSpace instantiated on Node 3 does not receive any requests.

In Figure 10, the values produced in a centralized JavaSpace configuration are also presented; we will discuss this configuration in Section 4.3.1.

4.2.3 Bandwidth Measurements

Let us now consider the bandwidth that the application uses as a metric for the performance. The *application bandwidth* is defined as the product of the size of the message sent and the number of messages that application components exchange using the network.

We use the following notation: *SensorTemplate* is the template that the aggregator uses in order to obtain a sensor message. *Sensor*, *Aggr* and *Compute* denote messages produced by the sensor, aggregator and computation unit respectively. The size of a message of type T is denoted $S(T)$. The speed of the aggregator is denoted v.

The bandwidth usage, measured for a given aggregator speed, is defined as the volume of data that the application components transfer through the network for the duration of the experiment.

The number of messages that the aggregator and computation-unit components produce depends on the aggregator speed and are denoted $NM_{pull}(v)$ for the pull setting and $NM_{push}(v)$ for the push setting.

Next we explain the formulas that we use for calculating the bandwidth for the different distribution policies.

In the push setting, the network traffic is generated by the sensor, the aggregator and the computation unit, but not by the display. The sensor messages are pushed to the aggregator node, and the number of messages that sensor produces over the experiment is constant. The aggregator and computation unit messages are pushed (to the computation unit node and to the display node, respectively) according to the aggregator consuming speed. Hence, for the push setting we have:

$$B_{push}(v) = 1000 * S(Sensor) + NM_{push}(v) * (S(Aggr) + S(Compute))$$

In the pull setting, the behavior of the sensor and the display components does not generate any network traffic. Here, only the aggregator and the computation unit generate network traffic. They do this as follows. The aggregator causes network traffic when it pulls messages from the sensor node. Also, the messages the aggregator produces are pushed to node of the computation unit. The computation unit causes network traffic because its messages are pushed to the display node. Hence, the formula to calculate the bandwidth usage, for the pull setting is:

$$B_{pull}(v) = NM_{pull}(v) * (S(SensorTemplate) + S(Sensor) + S(Aggr) + S(Compute))$$

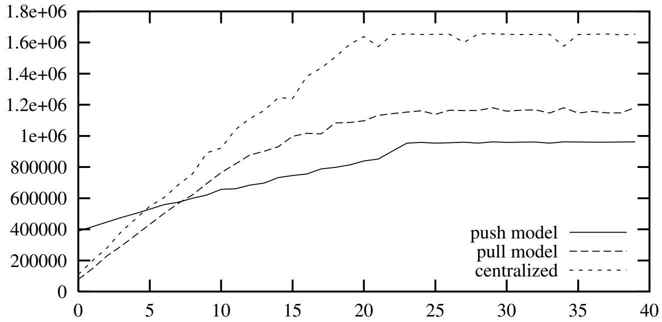

Figure 11: Total consumed bandwidth measured in relation to the number of sensor values consumed by the Aggregator.

Figure 11 shows the results of the measurement. The horizontal axis represents the number of messages that the aggregator consumes for each second. The vertical axis represents the value of application-level bandwidth. In the pull setting, we see that for relatively low aggregator speeds, the total consumed bandwidth is less than in the case of the push setting. This result is to be expected: in the pull setting the sensor does not waste any bandwidth by forwarding values to the aggregator when these values are not needed. On the other hand, when the aggregator speed increases, we see the overhead the pull setting introduces in the form of the additional requests that need to be sent to the sensor component. In effect, for larger aggregator speed values, the push setting shows to be more efficient.

Figure 11 also shows the bandwidth consumption when using a centralized JavaSpace server. This situation is discussed next.

4.3 A Centralized JavaSpace Implementation of a TMP

In this section we compare our results with an implementation that uses a single JavaSpace. Figure 12 shows the deployment of the components on the different nodes. For a fair comparison with GSpace, where all JavaSpaces share the resource with application components, we instantiate a single JavaSpace server at the node where the sensor and display applications reside. All messages are exchanged used this single, non-distributed data space. Figure 13 shows the message sequence diagram for this configuration.

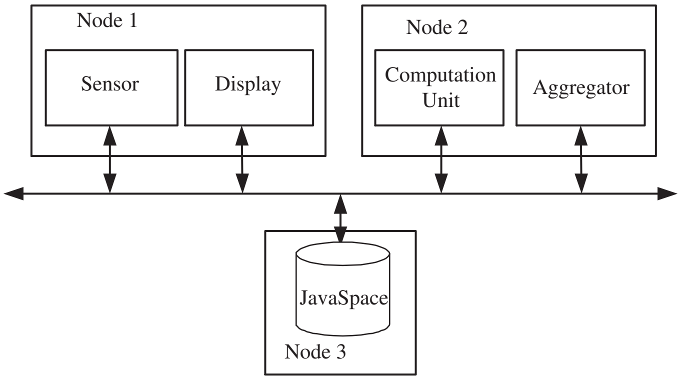

Figure 12: TMP component deployment using JavaSpace

4.3.1 Turnaround Time Comparison

As the graph in Figure 10 shows, the centralized JavaSpace configuration performs between the GSpace push and pull settings. The reason that leads to these results, is related to the performance of JavaSpace. In each configuration (GSpace push and pull settings, and centralized JavaSpace) the JavaSpace response time affects the turnaround time. When a JavaSpace has to serve too many requests and has to share computation resources with other applications, its response time increase is the main cause of the increase in turnaround time.

In the push setting, the accesses to the JavaSpaces are equally distributed among the nodes. In the centralized configuration, the space running on Node 1 (see Figure 12) has to serve all the requests coming from all the application components and moreover shares the computation resources with two other application (the sensor and the display). The response time in this case is worse than in the push setting and so is the turnaround time.

In the pull setting, as we explained in Section 4.2.2, the JavaSpace running on Node 1 receives most of the requests and shares the resources with other application components. Although this JavaSpace processes fewer requests than in the centralized configuration, an additional computation load is present in Node 1. In fact, in GSpace instantiates an Input Channel for each application component (see Figure 6. This module is implemented as a separated thread that processes all the incoming messages for the related application component. The presence of these threads, that perform requests on the JavaSpace, decreases the response time of this space. Consequently the turnaround time is even worse than that of the centralized configuration.

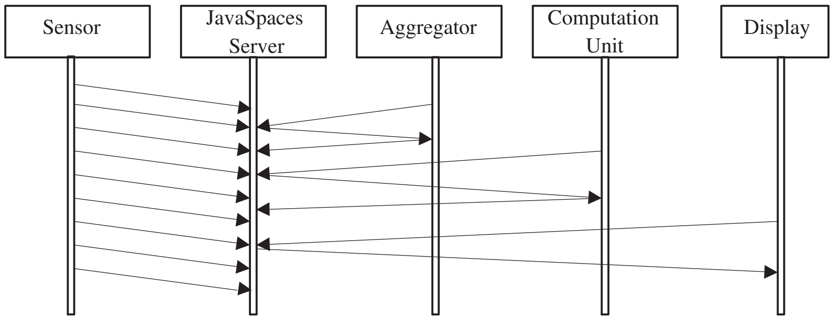

Figure 13: The message flow in the centralized JavaSpace configuration.

To validate the hypothesis that increasing the load of the node in which is running the JavaSpace leads to decreasing its performance, we modify the centralized configuration in the following way: For each of the application components running on Node 1 (sensor and display), we introduce one thread that performs requests to the JavaSpace. In this way we simulate the same computational load that Node 1 has in the pull setting. In doing so, the turnaround results are worse than that in the pull setting.

4.3.2 Bandwidth Comparison

In this section we discuss the results of the bandwidth used by the application implemented in centralized JavaSpace. We run the same experiments as in Section 4.2 and we use the same notation as defined in Section 4.2.3. *AggrTemplate* and *ComputeTemplate* are the templates used for retrieving the aggregator and computation unit data, respectively.

In the configuration shown in Figure 13, the application components exchange messages in the following way. The sensor stores its data to the local JavaSpace. Next, the aggregator collects sensor data and sends back what it produces. The computation unit gathers the aggregator data and sends its data. The display takes the computation unit data from the local JavaSpace and displays it. Hence the formula to compute the bandwidth usage, for a given aggregator speed, is the following:

$$B(v) = NM(v) * (S(SensorTemplate) + S(Sensor) + \\ 2 * S(Aggr) + S(AggrTemplate) + S(Compute))$$

All these network accesses significantly affect the application bandwidth as we vary the speed of the aggregator. Note that, as Figure 11 shows, the bandwidth use of this centralized approach is for most of the aggregator speed worse than that for either of the GSpace settings.

4.4 Experiment Conclusion

We observe that varying the distribution policies leads to different results and that no single distribution policy gives the best performance for all metrics. Hence, it is desirable that a middleware can support multiple policies.

An application developer can choose how to customize the middleware to increase the application performance with respect to a number of metrics. For instance, for the case study presented where the level of QoS is low, if the main issue is the turnaround time, then the application developer should choose the push distribution configuration, but at the cost of higher bandwidth usage. On the other hand, if the consumed bandwidth is more important, then the pull setting gives better performance. Instead, when high QoS is advisable (i.e., we want a high value for the consumer speed) the push setting is the best choice for the developer.

However, what is more important is that even for this simple application it can be seen that the performance of a system can be affected by the policies that are chosen concerning how and when data is distributed across a shared data space. We expect that differences between policies become more significant when considering other applications and other policies.

In this light, it becomes important to find a way by which, for a given metric, a "best" policy can be identified. As is also noted in [15], different data may need to be treated differently. In other words, we may need to support a range of different policies for different types of data. By monitoring the application and system behavior during run time, it becomes possible to automatically change policies if needed. How such a selection can be done is a major issue of our current research.

5 Related Work

5.1 Distributed Shared Data Spaces

From the moment shared data spaces were introduced, there has been considerable research in finding ways to attain efficient implementations on parallel and distributed machines. In many cases, *global* strategies have been investigated by which all data is distributed across the various machines following a single policy (see also [16] for an overview). In some cases, several global policies are supported and selecting the best one is left to an application developer. This approach has not only been followed for parallel applications, but also for reaching various degrees of fault tolerance (for a recent study, see [13]).

Only recently, some research has been conducted on how to differentiate strategies for the case of parallel programming [3]. The basic idea is that the programming and run time environment provides a separate interface that allows a developer to specify usage patterns for different types of tuples. Typical patterns include, for example, "write many" and "producer-consmer." This approach strongly resembles the way that researchers in distributed shared memory systems have sought to attain efficient implementations. The canonical example here is Munin [4]. Nevertheless, none of these studies provide a framework by which, in principle, arbitrary policies can be implemented as in GSpace.

5.2 Customizable Middleware

Related to our work is research in the area of customizable middleware. An architectural framework for a customizable Object Request Broker (ORB) that supports application-specific quality of service for non-functional requirements is presented in [10]. The ORB can be dynamically reconfigured to support application-specific requirements and this mechanism is achieved at run time by the selection of alternative ORB-component instances. Each of the ORB-components implement a non-functional requirement and can guarantee a service-level that the application specifies using a descriptive language. We are applying a similar idea to coordination-based middleware.

Another approach to building customizable middleware using Aspect-Oriented Programming (AOP) is presented in [9]. Here, the basic kernel of the middleware is represented as a real-time *event channel* (*RTEC*) implemented using AspectJ [11], an aspect-oriented language built on top of Java. The basic functionality of the RTEC is to provide a mechanism for exchanging data among the components that bind it. The application developer specifies the definition of aspects in separate files together with the application code. Then, using a build tool, the source files are compiled to build executable application code and to link them to the kernel of the middleware. Since interweaving the definition of the aspects is done at compile time, there is no possibility to change aspect specifications at run time. In GSpace, interweaving is actually done at run time, allowing us to change policy descriptors while an application is being executed is.

5.3 Publish/Subscribe-Based Systems

An implementation of a distributed shared data space middleware for embedded real-time distributed application is presented in the *SPLICE* [1] architecture. In SPLICE applications are composed by application components that interact using the shared data space. The distribution of the shared data space is achieved utilizing middleware components, called *agents*. Each application component interacts with exactly one agent. Each agent is responsible for providing a local database and task for handling communication with other agents. To distribute messages among several databases, agents use a publish/subscribe paradigm. Writing a message in the data space means to publish it. Reading a message from the space is equivalent to subscribe for that message's subject. As a result of this protocol, each agent stores in its database only messages needed by the application component. However, in its first version there was no support for separation of concerns and quality of service support. In the latest version, it is possible to define the quality of service to apply to a specific message's subject.

Related publish/subscribe projects for deploying real-time applications in local area networks support the idea of differentiating quality of service on the basis of data characteristics. In the Network Data Delivery Service, *NDDS* (`www.rti.com`) it is possible to customize delivery characteristics of each publication and subscription to meet the real-time behavior of the application. For instance, it is possible to set data transmission reliability high at the cost of longer data delivery time (caused by possible retransmissions and acknowledgments), or *vice versa*.

Another publish/subscribe system that is able to differentiate the quality of delivery is Distributed Asynchronous Collection, *DAC* [5]. DAC presents a hierarchy of classes (as sets, bags, queues) with different distribution characteristics. The application developer defines each type of application message as subclasses of DAC standard classes.

However, in these examples, the QoS specification is intertwined with the application code. If a QoS specification needs to be changed, then it is necessary to modify the application code. In GSpace, the definition of non-functional requirements is separated from the application code. This allows the application developer to localize changes. Furthermore, these changes can be done at run time without the necessity to stop the running application.

6 Conclusions and Future Work

We have presented a design for distributed middleware system that supports separating aspects of an application with interweaving taking place at run time. Applications are translated into components implementing the core functionality, along with different policy descriptors that capture non-functional aspects. Policy descriptors are interpreted at run time.

We have developed a prototype implementation and investigated the approach for differentiating distribution policies for applications that coordinate their activities through a shared data space. Even for our relatively simple prototype implementation and application, we can see the feasibility and flexibility of our approach.

However, there is still much work to be done. In the first place, we need to extend the way that distribution policies are specified and translated into policy descriptors. Also, we plan to look at other distribution-related topics such as activity scheduling between different components. In addition, our prototype system needs to be replaced by a more elaborate implementation that can support more serious applications, for example, as being deployed in home-network environments.

References

[1] M. Boasson and E. de Jong. "Software Architecture for Large Embedded Systems." In *Proc. RTSS'97 Workshop on Middleware for Distributed Real-Time Software System*, San Francisco, CA, Dec. 1997.

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

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

[4] 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.

[5] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. "The Many Faces of Publish/Subscribe." Technical Report MSR-TR-2001-104, Microsoft Research Laboratories, Cambridge, UK, Jan. 2001.

[6] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns and Practice*. Addison-Wesley, Reading, MA, 1999.

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

[8] D. Gelernter and N. Carriero. "Coordination Languages and their Significance." *Commun. ACM*, 35(2):96–107, Feb. 1992.

[9] F. Hunleth, R. Cytron, and C. Gill. " Building Customizable Middleware using Aspect-Oriented Programming." In *Proc. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, Oct. 2001. ACM.

[10] B. Jørgensen, E. Truyen, F. Matthijs, and W. Joosen. "Customization of Object Request Brokers by Application Specific Policies." In *Middleware 2000*, volume 1795 of *Lect. Notes Comp. Sc.*, pp. 144–163. Springer-Verlag, Berlin, 2000.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. "An Overview of AspectJ." In *Proc. 14th European Conf. on Object-Oriented Programming*, volume 1850 of *Lect. Notes Comp. Sc.*, Sophia Antipolis, France, June 2000.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming." In *Proc. 11th European Conf. on*

Object-Oriented Programming, volume 1241 of *Lect. Notes Comp. Sc.*, Berlin, 1997. Springer-Verlag.

[13] J. Larsen and J. Spring. “GLOBE: Global Object Exchange.” Master’s thesis, University of Copenhagen, Department of Computer Science, Oct. 1999.

[14] G. Papadopoulos and F. Arbab. “Coordination Models and Languages.” In M. Zelkowitz, (ed.), *Advances in Computers*, volume 46, pp. 329–400. Academic Press, New York, NY, Sept. 1998.

[15] 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.

[16] A. Rowstron. “Run-time Systems for Coordination.” In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, (eds.), *Coordination of Internet Agents: Models, Technologies and Applications*, pp. 78–96. Springer-Verlag, Berlin, 2001.

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