Design and implementation of a real-time distributed shared data space

Spoor, R.D.G.M.

Award date:
2004

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

Design and Implementation of a Real-Time Distributed Shared Data Space

by
R.D.G.M. Spoor

Supervisors at TU/e : dr. M.R.V. Chaudron
G. Russello MSc.

Eindhoven, November 2004

Abstract

Existing real-time middlewares require that real-time constraints are specified by the application components. Software engineers have to interweave component functionality with timing constraints. This diminishes component reusability; changing timing constraints requires changing application components.

We have worked on a real-time middleware, based on the shared data space model, that separates both data distribution and timing constraints from component functionality. In this thesis, we will give the design of a proof of concept, which we call RGSpace, and give results of a real-time application using RGSpace.

Acknowledgments

First of all, I would like to thank my two supervisors: Michel Chaudron for introducing me to this project and guiding me throughout the project, and Giovanni Russello for guiding me and giving me technical advice when I needed it. I couldn't have finished this project without their help.

Iñaki Lazarobaster Badiola, for his friendship and for providing me the network communication I needed.

Reinder Bril for sharing his knowledge about real-time, he was really a great help.

Richard Verhoeven for helping me set up the test environment, and his technical expertise.

I would also like to thank the rest of the SAN group for their friendship the past fifteen months.I feel really lucky to have been part of this group.

Last but certainly not least, I would like to thank my family and girlfriend for their mental support throughout these fifteen months, I love you all!

Contents

Chapter 1

Introduction

In software engineering, it has become highly desirable that software can be reused so productivity can be increased and time-to-market decreased. However, existing pieces of software must be easily composed; otherwise the effort it would take to compose those pieces would nullify the effort saved by using existing software. Therefore, easy dynamic (de)composition of software components is essential for software components to be successfully deployed. This dynamic (de)composition is realized by means of a composition technology. Usually, this composition technology is a part of an extra software layer called *middleware*.

Code reusability is particularly relevant in engineering real-time systems. Up to now, real-time systems were built from scratch since there has been a lack of tools and support for valid software engineering techniques enhancing dynamic (de)composition of software.

Matters have changed after the advent of component-based techniques supporting real-time properties. However, without middlewares providing real-time guarantees it could not be possible to build real-time component-based systems. There are quite a number of real-time middlewares that support dynamic (de)composition of software components. Yet, those middlewares require that real-time constraints such as periods and deadlines are specified by the application components. This means that real-time software engineers have to specify timing constraints in the application code. As a result, the basic functionality of an application will be interweaved with extra-functional properties. If a component is deployed in an environment with different timing constraints, then the component code needs to be changed. Clearly this makes components less reusable if an application needs to be deployed in different environments.

To facilitate reuse it is important that timing constraints are separated from component functionality. In the approach adopted in this thesis, com-

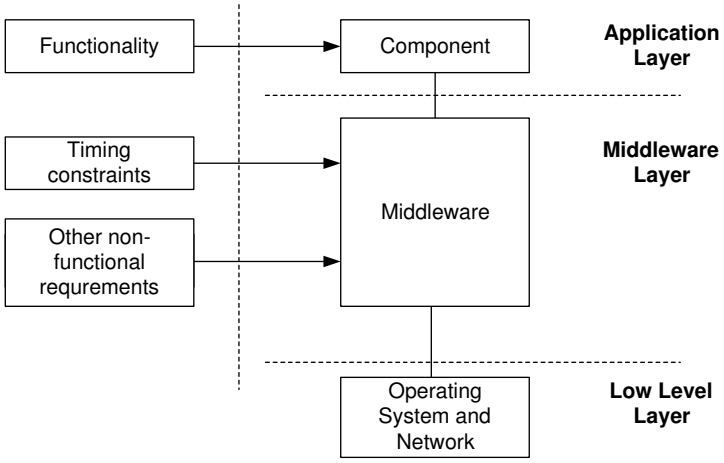


Figure 1.1: Separation of concerns

ponents specify only their functional requirements, while timing constraints and other non-functional requirements are directly passed to the middleware. Figure 1.1 shows this approach. This separation of concerns enables non-functional requirements to be changed without changing any component code.

One typical timing constraint is the *deadline* of an operation. When we will talk about deadlines, we mean the maximum allowed time between the time an application component sends a request to the middleware and the time this request is handled completely[1].

There already exists a non-real-time middleware that supports a separation of concerns: *GSpace* [12][13]. This middleware is based on the distributed *shared data space* model. It introduces another non-functional requirement: *data distribution*. Instead of having a fixed data distribution policy, data can be distributed according to the needs of components. Examples are data stored in one place or replicated throughout the distributed shared data space. Currently, GSpace only separates data distribution from component functionality. However, other concerns are being investigated.

1.1 Goals

The goal of this project was to show the feasibility of separating real-time constraints from functionality. To this end, we have created a real-time middleware that supports this separation. Since GSpace already has a separa-

[1] If the request requires a result to be sent back to the component, the time this result is received is the end time.

tion of concerns, we have decided to use it as a base for our new middleware; as a result, it will know a separation of both timing and data distribution. We have also built a proof of concept. Because in hard real-time systems missing deadlines can be disastrous, predictability is more important than performance.

1.2 Outline of report

The next chapter gives a more detailed description of the shared data space model. Next, GSpace will be discussed in Chapter 3. After that we will give a case study in Chapter 4, followed by the design and implementation details of our real-time version of GSpace in Chapter 5. Chapter 6 will contain some related work, and we will conclude in Chapter 7 with an evaluation of the project and some recommendations for future work.

Appendix A lists all abbreviations used throughout this thesis, and Appendix B gives a short description of some scheduling algorithms. The class and sequence diagrams of the implementation of Chapter 5 are collected in Appendix C, and Appendix D contains technical details about the test environment.

Chapter 2

Shared data spaces

The shared data space model provides dynamic (de)composition through the blackboard principle: components can insert data into the data space or retrieve data from the shared data space. Since all messages are stored in the data space, all communication between components goes through the data space. This means that there is a *referential and temporal decoupling* between components [6]. Referential decoupling means that components exchange data without the needs of knowing each other. Temporal decoupling means that those components do not even have to be online at the same time. This way, components can be connected to or disconnected from the data space at any time, making them easier to combine or replace.

2.1 Linda

The shared data space model was introduced by the coordination language Linda [7][8][9]. Storage in Linda takes place in a so-called *tuple space*. In this tuple space, data is stored as persistent objects, called *tuples*.

Linda provides three basic operations: `out`, `in` and `rd`, and two variant forms, `inp` and `rdp`. `out` creates tuples and inserts them into the tuple space. The `in` and `rd` operations respectively take (destructive) and read (non-destructive) a tuple from the tuple space, using a template for matching. This template is a tuple by itself, and the tuple returned must exactly match every value of the template. Templates, however, can contain wildcards, which match any value, expanding the range of possible matches. Whereas putting a tuple inside the tuple space is non-blocking (i.e. the process that puts the tuple returns immediately from the call to `eval` or `out`), reading and taking from the tuple space is blocking: the call returns only when a matching tuple is found. `inp` and `rdp` are predicate versions of `in` and `rd`: they too try to return a matching tuple. However, if there is no such tuple they do not block but return a value indicating failure.

2.2 JavaSpaces

Java has its own specification for shared data spaces, which are called JavaSpaces [16]. Although it is strongly influenced by the Linda model, there are basic differences:

- Unlike Linda, JavaSpaces have rich typing on the tuples themselves, not just their values. As a result, looking up tuples takes into account the type: only tuples of the same type – or a subtype – are returned.
- Since tuples in JavaSpaces are Java objects, they may have methods associated with them.
- Tuple fields in JavaSpaces may contain Java objects as well, allowing more complex data structures inside the shared data space.

There is also a different set of terms in JavaSpaces. Tuples are called *entries*, and `out`, `in`, `rd`, `inp` and `rdp` are renamed to `write`, `take`, `read`, `takeIfExists` and `readIfExists` respectively.

Unlike in Linda, `take` and `read` operations are not fully blocking: if there is no matching entry inside the JavaSpace, the operation could only wait for a user specified time, before returning the value `null`. Also, `write` operations could write entries under a *lease*. When the lease expires, the entry is removed.

JavaSpaces also have a *notify* operation, which notifies when entries matching the given template are written to the JavaSpace.

JavaSpaces provide a simple *transaction* system that allows composition of space operations into a single atomic operation. If not all operations can be completed, the effects of all other operations will be discarded. Transactions in JavaSpaces closely resemble transactions in database systems.

Chapter 3

GSpace

Although shared data spaces (see Chapter 2) are a simple composition mechanism, their efficient distributed implementations face several complicating factors. One such factor is the fact that "the communication needs of components may differ per data type, per application, and may even change over time."[12]. Since existing data space implementations treat all data equally, research described in [12] and [13] has addressed this issue. As proof of concept, a prototype of a distributed shared data space was built, named **GSpace**.

3.1 GSpace architecture

This section describes the architecture of GSpace. GSpace's design allows application designers to separate functionality from extra-functional concerns, such as data distribution. In GSpace, data distribution requirements are declared separately from the application code. As a result, writing application component code is easier because developers can focus on the functional requirements of the components only. Extra-functional requirements can be treated separately and added in a later stage of the design.

3.1.1 Conceptual view

Figure 3.1 shows the separation of concerns in GSpace: an application's functional specifications are mapped onto *components* which may be distributed onto various nodes. Distribution requirements are converted into a *distribution policy descriptor* that is downloaded into the middleware where it is interpreted at runtime.

3.1.2 Implementation view

Although applications conceptually view GSpace as a single data space, GSpace is composed of several *GSpace kernels*, distributed across multiple

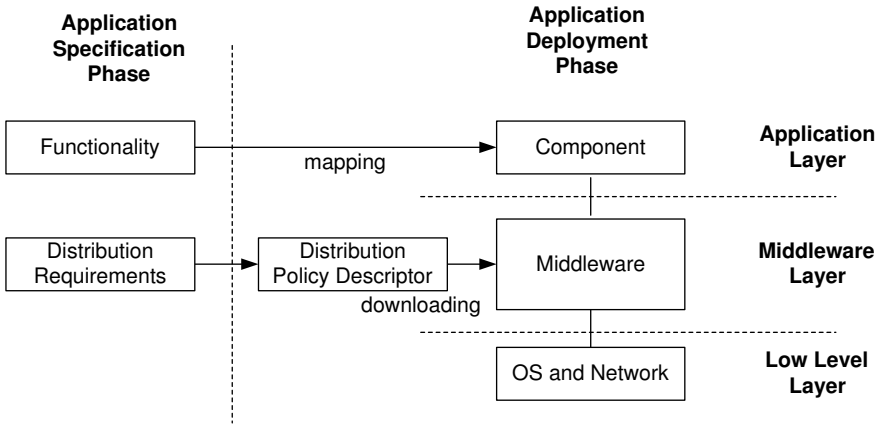


Figure 3.1: Separation of concerns in GSpace

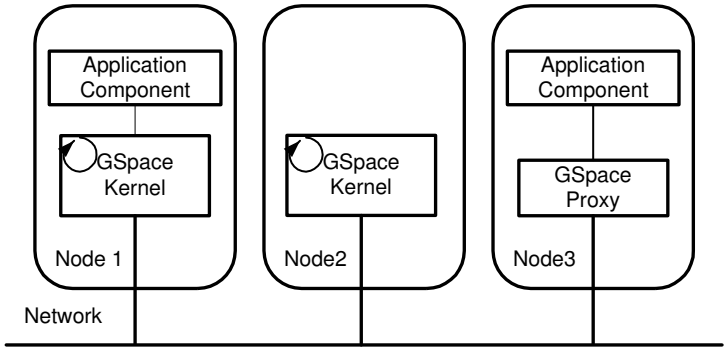


Figure 3.2: GSpace kernel and application components placement across several nodes

nodes (see Figure 3.2). A kernel provides local storage and some means to communicate with other kernels. Applications may be placed either on the same node as a kernel, like with node 1 in Figure 3.2. If the node has limited resources however, the application can use a proxy to communicate with a remote kernel (node 3 in Figure 3.2).

Although GSpace is written in Java, it is *not* a JavaSpace. While objects (called tuples, as in Linda) are typed, and matching on exact types is possible, matching on subtype is not. Furthermore, for `read` and `take` operations, GSpace follows the Linda model: they block until a matching tuple is available. `readIfExists` and `takeIfExists` are not supported. `write` operations, renamed to `put`, also behave like Linda's `out`; the lease system is not present in GSpace. Finally, GSpace also lacks the notion of transactions.

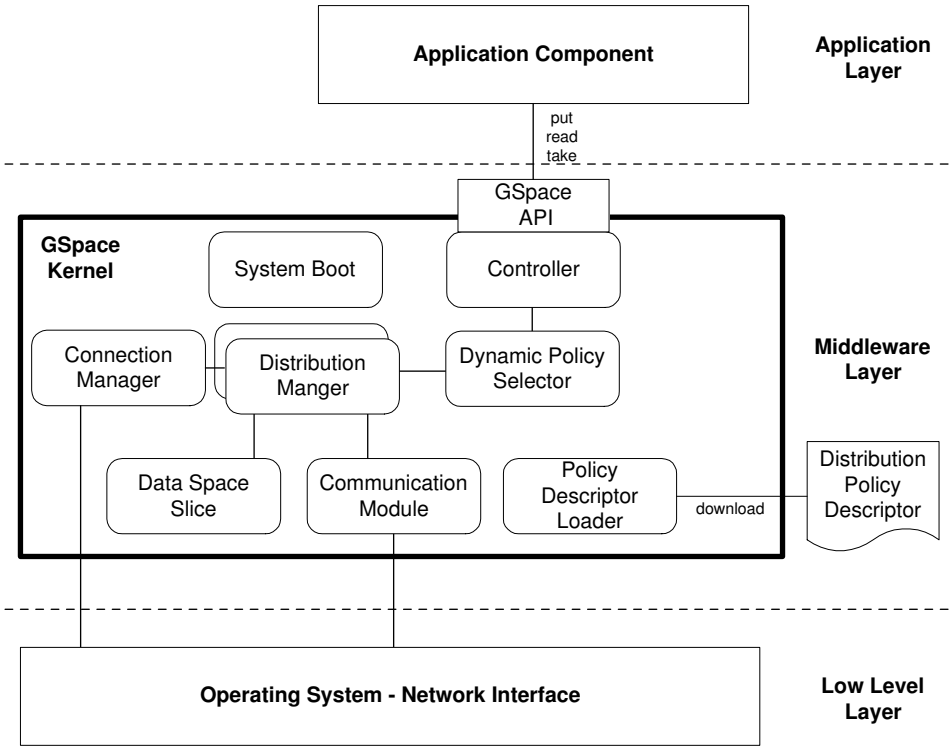


Figure 3.3: Internal structure of a GSpace kernel

Partitioning of the data (tuples) in GSpace is based on tuple types. Tuples of the same type are treated according to the same distribution policy. In other words, "tuples of different types may be distributed according to very different strategies."[12].

The internal architecture of a GSpace kernel is shown in Figure 3.3. The next listing explains each module.

- The **System Boot** module initiates all other modules of the kernel. It also advertises its presence to other GSpace kernels in order to join a GSpace group.
- The **Controller** handles application components' requests for space operations. These requests could be either local or remote.
- The **Dynamic Policy Selector** (DPS) looks up the matching distribution policy for a given tuple and instantiates an appropriate distribution manager. Distribution managers are only instantiated if needed,

and can be added at runtime.

- The **Data Space Slice** is the local storage for tuples.
- The **Distribution Managers** enforce distribution policies. For each supported distribution policy, there is a `Distribution Manager`. Tuples can be sent to or retrieved from the local slice or another kernel. Table 3.1 lists several Distribution Managers.
- The **Communication Module** is responsible for sending and retrieving tuples to and from other kernels. Communication can take on different forms (such as multicasting or point-to-point), but also different quality of services (such as reliability).
- The **Connection Manager** keeps track of the location of other kernels.
- The **Policy Descriptor Loader** downloads the distribution policy descriptor file at boot time, and fills the kernel's *policy table*: a mapping from tuple types to distribution policies. Since this file may be changed at runtime, the loader monitors the descriptor for changes and reloads if necessary. GSpace will then adapt to the new distribution requirements.

3.2 Control flow in GSpace

Figure 3.4 shows an example of the control flow of a `read` operation using a Store locally distribution policy. When an application component issues a request to GSpace through the `Controller`, the latter passes the request to the `Dynamic Policy Selector`. This looks up in the kernel's policy table which distribution policy (and therefore which `Distribution Manager`) to use. If there is no such `Distribution Manager` yet instantiated, the `Dynamic Policy Selector` creates it. Since the policy for the given tuple type in the example is Store locally (see Table 3.1) the request is passed to the `Store Locally Distribution Manager`.

This `Distribution Manager` first searches the local `Data Space Slice` for a matching tuple. In the case of Figure 3.4 case, no matching tuple could be found. Thus, the `Distribution Manager` forwards the request to other nodes through the `Communication Module`. When a matching tuple is found on a remote node, the tuple is returned to the application component, passing in the opposite order through the same modules.

Distribution Manager	Description
Store locally	A tuple is always put into the local slice. `read` and `take` operations are performed on the local slice first; if it is not present the request is forwarded to other nodes.
Push-to-all	A `put` operation forwards the tuple to all known nodes. `read` operations are performed on the local slice. `take` operations also forward requests to all known nodes.
Push-to-one	A `put` operation forwards the tuple to one specific node. `read` and `take` operations are performed on this specific node.
Cache-on-demand	Tuples are stored in the local slice. The tuple returned by a `read` operation on a remote node is also cached in the local slice of the requesting node. When a cached tuple is removed through a `take` operation, an invalidation message is sent to invalidate all cached copies of the tuple.

Table 3.1: A list of several Distribution Managers available in GSpace

3.3 Adaptation in GSpace

GSpace offers a suite of distribution policies to tailor the data distribution needs of an application. However, the selection of an appropriate distribution policy is not straightforward. Users of GSpace might select distribution policies based on their knowledge of their applications. However, it could be the case that this information is not completely correct. To complicate matters, it might be the case that the application changes its behaviour after deployment.

To circumvent this problem, a mechanism is employed in GSpace that can autonomously select the distribution policy that best suits the application's current data distribution needs. In the following, we will briefly describe this mechanism. For the interested reader more details can be found in [13].

To quantify which distribution policy best suits the application distribution needs, it must be possible to compare their performances. The following metrics have been introduced that provide a snapshot of the performance of a given distribution policy:

- *rl*: the latency for the execution of a `read` operation
- *tl*: the latency for the execution of a `take` operation

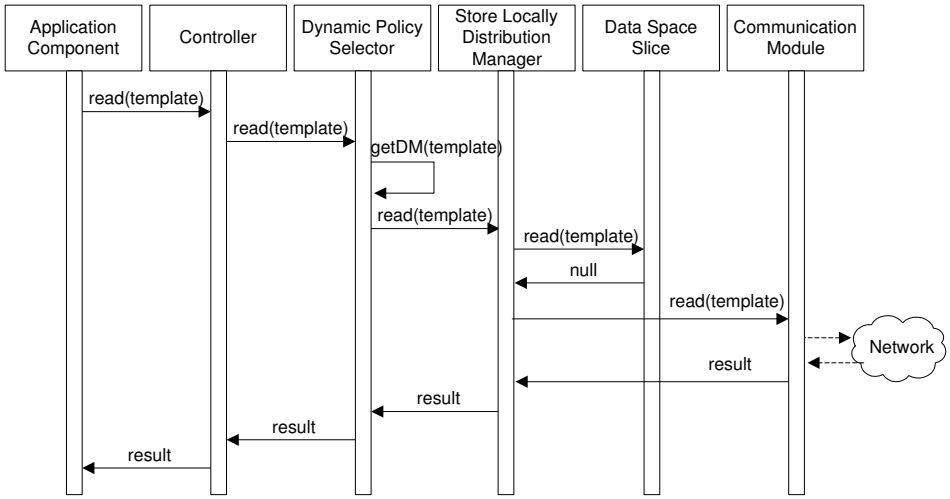


Figure 3.4: Handling a read request from an application component

- bu: the network bandwidth usage
- mu: the memory consumption for storing tuples in the local slice.

Since `put` operations are not blocking, the latency for the execution of a `put` operation is not used. For a policy p the values of each metric are combined in a cost function CF, using the following formula:

$$CF(p) = w_0 * rl_p + w_1 * tl_p + w_2 * bu_p + w_3 * mu_p \tag{3.1}$$

The w_i are user defined weights, with $0 \leq w_i \leq 1$ and $\sum w_i = 1$. The higher the importance of a metric for the user, the higher the weight. If for instance one is only interested in bandwidth usage, w_2 should be 1 and the others 0. The distribution policy with the lowest CF value is considered the best.

Now that distribution policies can be compared, it is possible to determine which policy best fits the current behaviour of an application. It is assumed that it is possible to predict the behaviour of an application from its recent past. GSpace collects logs of the last executed operations. These logs are used for simulating the recent behaviour of the application. For each distribution policy, the simulation produces the values for each metric and the respective cost function value. The policy with the lowest CF values is selected as the best policy. If the current policy differs from the selected best policy the system has to adapt. This means that all GSpace kernels enter a transitional state to update their internal structures. During this phase, the system will freeze application requests to avoid inconsistency

among the different kernels. When the transition from the old policy to the new selected policy terminates the system resumes its normal operational mode.

Because data distribution policy is differentiated per tuple type, all the above phases (evaluation, adaptation and transition) can be executed per tuple type.

Chapter 4

A case study

Control towers of airports must constantly monitor all planes that arrive or depart. Any mistake can possibly cause planes to crash into each other. Therefore, the systems of these control towers cannot afford to make mistakes, and operations must be executed in time. Any mistake or missed deadline can be disastrous, and cost human lives.

This chapter focusses on the monitoring system, albeit in a simplified form. This is an adaptation of the example found in [4]. It observes the airspace around the airport and displays the detected objects (mostly planes). It can also closely follow one such object in a so-called *tracking mode*. The observations are obtained from a radar, which "represents the observed objects in *plots*. These plots are then correlated into *tracks*, which are used to predict the position of the object at the time of the next observation. These coordinates serve to control the radar so that its next observation will not miss the tracked object"[4].

Figure 4.1 shows this system. It consists of the following components:

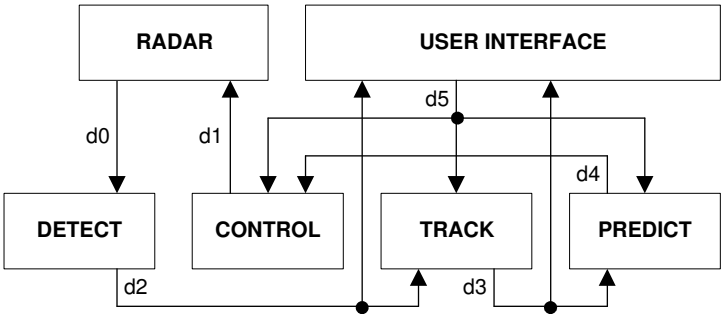


Figure 4.1: A airplane monitoring system.

- **RADAR** is the actual radar. It produces raw sensor data, and can consume control data.
- **USER INTERFACE** provides user interaction and shows data to the user.
- **DETECT** performs plot detection and extraction from raw sensor data from the radar.
- **CONTROL** controls the radar based on user commands.
- **TRACK** correlates successive plots into tracks.
- **PREDICT** attempts to predict the next position of a tracked object.

The data streams are as follows:

- **d0**: raw data from the radar.
- **d1**: control data for the radar.
- **d2**: object coordinates as seen by the radar (plots).
- **d3**: speed-vectors of tracked objects.
- **d4**: newly predicted coordinates of tracked objects.
- **d5**: user commands for managing the system.

The following is some pseudo code for some of these components:

```
RADAR:
    SensorData rawData; // data for stream d0
    ControlData ctrlData; // data from stream d1
    ControlData cdTempl = "appropriate template for ctrlData";
    FOREVER DO
        rawData = "get data from hardware/buffer";
        PUT(rawData);
        ctrlData = TAKE(cdTempl);
        "handle commands of ctrlData";
    END FOREVER

DETECT:
    SensorData rawData; // data from stream d0
    Coordinates objPos; // data for stream d2
    SensorData rdTempl = "appropriate template for rawData";
    FOREVER DO
        rawData = TAKE(rdTempl);
```

```
        IF rawData != null THEN
            // there was data available
            objPos = "compute the positions from rawData";
            PUT(objPos);
        ELSE
            // no new data, let the designer take appropriate
            // corrective action
        END IF
    END FOREVER

PREDICT:
    Coordinates predPos; // data for stream d4
    Track track; // data from stream d3
    UserCommand command; // data from stream d5
    Track trTempl = "appropriate template for track";
    UserCommand comTempl = "appropriate template for command";
    Track localTrack; // local variable
    FOREVER DO
        track = READ(trTempl);
        IF track != null THEN
            // there was data available
            localTrack = track;
            predPos = "predicted new position"
        ELSE
            // no new track, predict according to old track
            predPos = "predicted new position"
        END IF
        PUT(predPos);
        command = READ(comTempl);
        IF command != null THEN
            "handle command"
        END IF
    END FOREVER

USER INTERFACE:
    Coordinates objPos; // data from stream d2
    Track track; // data from stream d3
    UserCommand command; // data for stream d5
    Coordinates opTempl = "appropriate template for objPos";
    Track trTempl = "appropriate template for track";
    FOREVER DO
        command = "get data from hardware/buffer"
        PUT(command);
        objPos = READ(opTempl);
```

```
    track = READ(trTempl);
    "display data from objPos, predPos and track"
END FOREVER
```

As can be seen, all communication goes through the distributed shared data space. Please note that while the control data from the `CONTROL` component is removed by the `RADAR` component, this cannot be done if the data is shared among components, such as track data from the `TRACK` component.

If we look at data distribution, we see the following data types and the components where they are needed:

- `SensorData` is needed only in the `RADAR` and `DETECT` components.
- `ControlData` is needed only in the `RADAR` and `CONTROL` components.
- `Coordinates` is needed by all components except `RADAR`.
- `Track` is needed by the `TRACK`, `PREDICT` and `USER INTERFACE` components.
- `UserCommand` is need by the `CONTROL`, `TRACK`, `PREDICT` and `USER INTERFACE` components.

If every machine runs one node from the data space and one component, then the following distribution policies could be devised:

Data type	**Distribution policy**
SensorData	Push-to-one
ControlData	Push-to-one
Coordinates	Push-to-many
Track	Push-to-many
UserCommand	Push-to-many

Some notes: for `SensorData` and `ControlData`, there are two candidates for sending the data to, `RADAR` and `DETECT/CONTROL`. Which one is chosen does not really matter at this time.

The `Push-to-many` distribution policy is a limited version of the `Push-to-all` distribution manager (see Table 3.1); it will push data to those nodes that need it. This policy has been successfully implemented for GSpace[1].

The following timing constraints could be devised[2]:

[1]The GSpace implementation is called `StaticReplicate` instead.

[2]These are just randomly chosen numbers.

Component	Operator	Period	Deadline
RADAR	PUT(SensorData)	250ms	50ms
RADAR	TAKE(ControlData)	250ms	100ms
DETECT	TAKE(SensorData)	500ms	100ms
DETECT	PUT(Coordinates)	500ms	200ms
PREDICT	READ(Track)	500ms	200ms
PREDICT	PUT(Coordinates)	500ms	50ms
PREDICT	READ(UserCommand)	500ms	150ms
USER INTERFACE	PUT(UserCommand)	500ms	100ms
USER INTERFACE	READ(Coordinates)	500ms	150ms
USER INTERFACE	READ(Track)	500ms	150ms

Please note that all operations within the same run/FOR loop must have the same period since they will occur with the same frequency. However, a PUT can have a shorter period than a READ of the same type, leading to some data not being used, or a larger period, leading to old data being reused.

What remains is error handling: what must be done if a deadline is missed. We have devised the following error handling policies:

Component	Operator	Error Handling Policy
RADAR	PUT(SensorData)	Safely shut down.
RADAR	TAKE(ControlData)	Safely shut down.
DETECT	TAKE(SensorData)	Safely shut down.
DETECT	PUT(Coordinates)	Safely shut down.
PREDICT	READ(Track)	Use the old value.
PREDICT	PUT(Coordinates)	Safely shut down.
PREDICT	READ(UserCommand)	Use the old value.
USER INTERFACE	PUT(UserCommand)	Safely shut down.
USER INTERFACE	READ(Coordinates)	Safely shut down.
USER INTERFACE	READ(Track)	Safely shut down.

If data for prediction cannot be delivered in time, this is not that bad. Old data can be used, and the prediction will be somewhat more imprecise, but it will probably still be a valid prediction.

Any other missed deadline will jeopardize the safety of passengers and crew aboard the airplanes, and thus safely[3] shutting down the system is the possibly the most responsible action to take.

[3]Safely means that currently landing planes should be guided to a safe runway, and planes taking off take off successfully. However, planes that are not yet landing should not land and grounded airplanes should stay grounded.

Chapter 5

RGSpace

While GSpace (see Chapter 3) is well suited for regular systems, it lacks necessary properties for real-time systems. Handling requests is done in a best-effort fashion, giving no time guarantees. For hard real-time systems, it is essential for tasks to finish before the given deadline. GSpace's best-effort handling is clearly unacceptable. Therefore, we have worked on an extension to GSpace that does have timing properties. We call this extension **RGSpace**.

In this chapter we introduce the RGSpace architecture and its current implementation. We also give some test results.

5.1 RGSpace architecture

Figure 5.1 shows the separation of concerns in RGSpace. Apart from the functional specifications and distribution requirements, RGSpace also takes into account timing constraints and timing error handling policies. The former declare periods of requests and deadlines whereas the latter specify what must happen if a deadline is missed[1] or a task is unschedulable. Examples of timing error handling policies could be shut down the system, reject the task, or reschedule the task with a less strict deadline.

Similar to distribution policies, both timing constraints and timing error handling policies are converted into a *timing constraints descriptor* and an *error handler descriptor* respectively, which are downloaded into RGSpace kernels at boot time.

While GSpace has only one thread for handling requests in a best-effort manner, the timing constraints in RGSpace cannot permit such a strategy. Instead, RGSpace has several threads to handle requests. These threads

[1] Usually new tasks must pass an acceptance test before being scheduled. In such cases, missing a deadline after a task has been scheduled will be an exceptional occurrence.

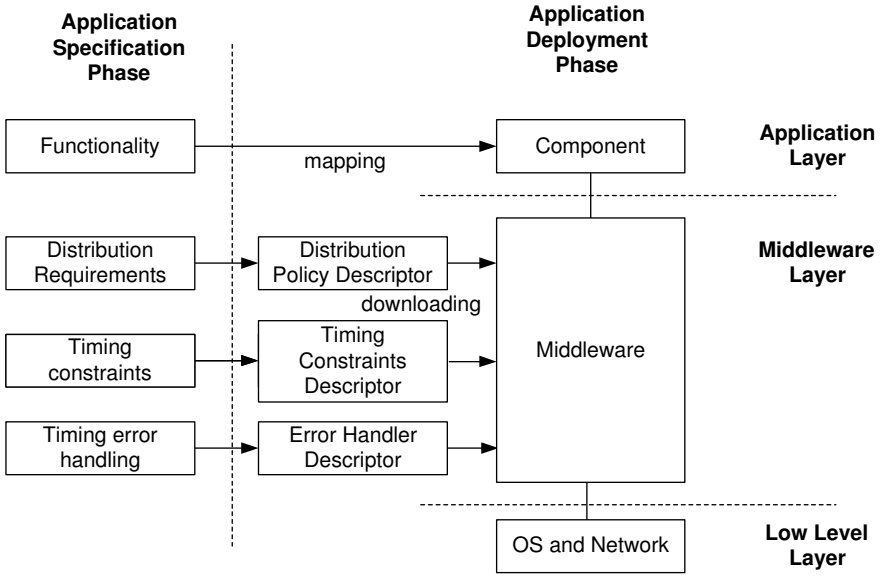


Figure 5.1: Separation of concerns in RGSpace

are controlled by a scheduler that preempts threads when necessary. This ensures no request misses its deadline unnecessarily.

The internal architecture of an RGSpace kernel (see Figure 5.2) is very similar to that of a GSpace kernel (see Figure 3.3 in Chapter 3). The former differs from the latter for the inclusion of additional modules, described as follows:

- The **Scheduler** is responsible for scheduling all threads in the kernel, making sure that no task misses its deadline unnecessarily.
- The **Thread Pool** keeps a collection of threads, ready to handle requests. If either an application component or another kernel issues a request, a thread is taken from the thread pool to handle the request. When that thread completes the request, it is put back into the thread pool.
- A more general **Descriptor Loader** has been introduced to download all three descriptors.

Where in GSpace the `Controller` passes requests to the `Dynamic Policy Selector` (DPS), it cannot do so in RGSpace: we would again get the best-effort behaviour. Instead, it should take a thread from the `Thread`

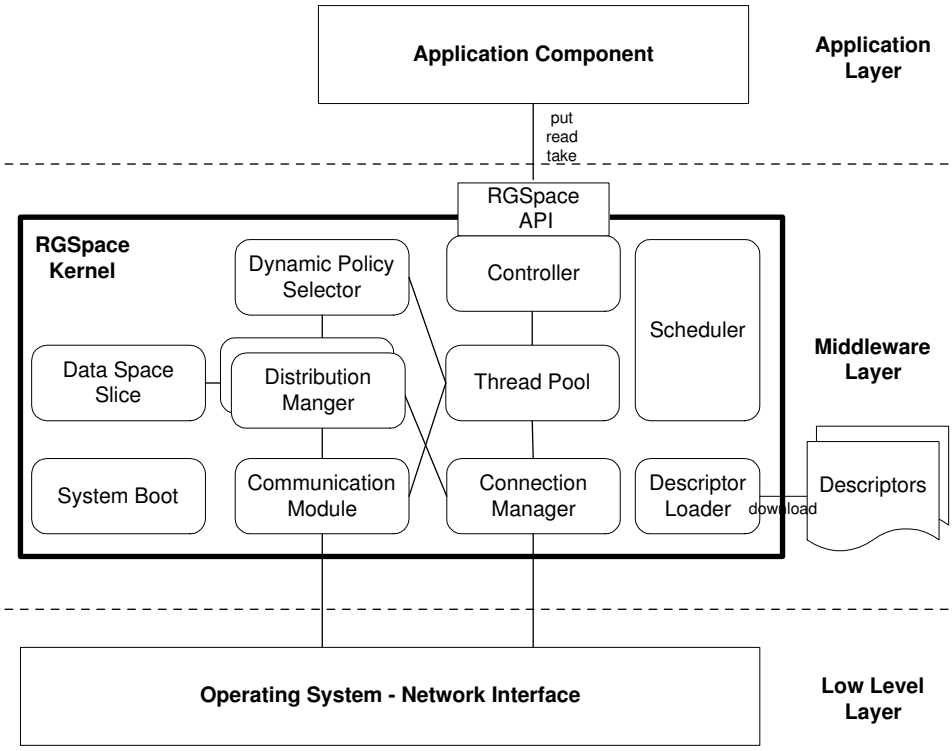


Figure 5.2: Internal structure of a RGSpace kernel

`Pool` and let the thread handle the request instead. Upon activation by the `Scheduler`, the thread should call the `DPS`, which works the same as in GSpace (see Section 3.2). When the `DPS` returns the tuple, the thread should notify the `Controller`, which then returns the tuple to the application component.

5.2 RGSpace implementation

The RGSpace architecture is a general architecture for hard real-time applications. This is usually quite complex to achieve however. Therefore, for the implementation, we have limited the application domain to accept periodic requests only. Recall from Section 1.1 that the main focus of the implementation is predictability, not performance.

Another limitation is the number of periodic requests the system can handle. Application components have no knowledge of the timing constraints of the requests they issue. The only data that can be used for

communication between an application component and RGSpace is tuples. If a component could issue multiple periodic requests of the same type and operator (`read`, `take` or `put`), there would be need of multiple constraints per type and operator. However, RGSpace could have no way of knowing which timing constraints belong to the request. After all, since the constraints are for a specific periodic request (identified by type and operator), the timing constraints descriptor would be ambiguous; which constraint should be used? As a result, only one periodic request of a specific type and operator can be used per kernel.

Since threads now do most of the work, they are (together with the scheduler for scheduling them) the most important of all modules in RGSpace. Therefore, choosing an appropriate policy for creating and (possibly) reusing threads is essential. We have come up with the following policies, each with its pros and cons:

- Creating threads when they are needed, and reuse them for serving future requests. This way, no knowledge about requests is needed beforehand: a thread is only created when the first period of a request is executed. Each next period the same thread is reused. Major cons are that both creating threads and scheduling tasks are *online*[2], and therefore increase execution time.

- Using a pre-created set of threads, each with its own period, deadline and cost. This way threads can both be created and scheduled *offline*[3]. Again, no knowledge about requests is needed beforehand: when a thread is needed, the best matching (with regards to period, deadline and cost) is taken. However, if threads differ either for periods, deadlines or costs from the request, no matching thread may exist. This may lead to periodic requests missing their deadlines some of the time.

- Per request ad-hoc created thread. This solution solves the issue of the previous policy since each thread in the thread pool matches one periodic request exactly. Although this needs a-priori knowledge about requests, it inherits all other advantages from the previous policy. Moreover, the total set of requests will already be known from the timing constraints descriptor.

Therefore, we have adopted the last policy in the current RGSpace implementation.

[2]Online means while the system is running.

[3]Offline means during initialization, while the system has not yet entered an active state.

Another major issue is the algorithm to use for scheduling the different threads. Since the current implementation will have to accept periodic requests only, we can limit ourselves in our choice of a suitable algorithm. Appendix B lists four suitable algorithms: Rate Monotonic (RM, see Section B.2), Deadline Monotonic (DM, see Section B.3), and two versions of Earliest Deadline First (EDF, see Section B.4). RM and the standard version of EDF are limited to constraints with deadlines equal to periods. This is too much of a limitation. Also, EDF has dynamic priorities. This requires extra runtime execution time to change the priorities. RM and DM, having static priorities, do not have this extra requirement. As a result, we have chosen for DM as the scheduling algorithm.

In GSpace, the `Controller` opens one connection for each request. For aperiodic requests this is fine, but for periodic requests it is a lot more efficient to have one connection for each periodic request and reuse that connection in each period. However, this introduces another problem: how does the `Controller` know from which connection to read at a given moment?

The solution we have chosen is not the most elegant, but it is quite efficient: since there is already one thread for each periodic request, the threads themselves receive the requests directly. Since the threads already know when to handle requests, they also know when to receive a request. As a result, the `Controller` loses its purpose, and therefore it was not included in the implementation. This relieves the scheduler from yet another task. Figure 5.3 shows the new architecture.

Next we will describe the modules at a lower level. Appendix C gives the class diagrams.

5.2.1 Kernel and System Boot

The `Kernel` class does not provide any methods. In fact, it is nothing more than a container for all modules. There is no way to instantiate it, to ensure there is only one kernel in each application that uses the kernel directly[4]. Because of this, all fields are static.

There is one field for every module inside the RGSpace kernel (see Figure 5.3). They are public, to save on function calls[5]. They are initiated by the System Boot (see below).

Kernels have a unique identifier, `kernelID`. This field is used for communication among kernels, and is the way kernels identify each other. Also, this identifier makes it possible to have multiple periodic requests of the

[4]Unfortunately, there is no way to ensure only one kernel is used on each machine, which would be ideal.

[5]Function calls take time, and are therefore eliminated as much as possible.

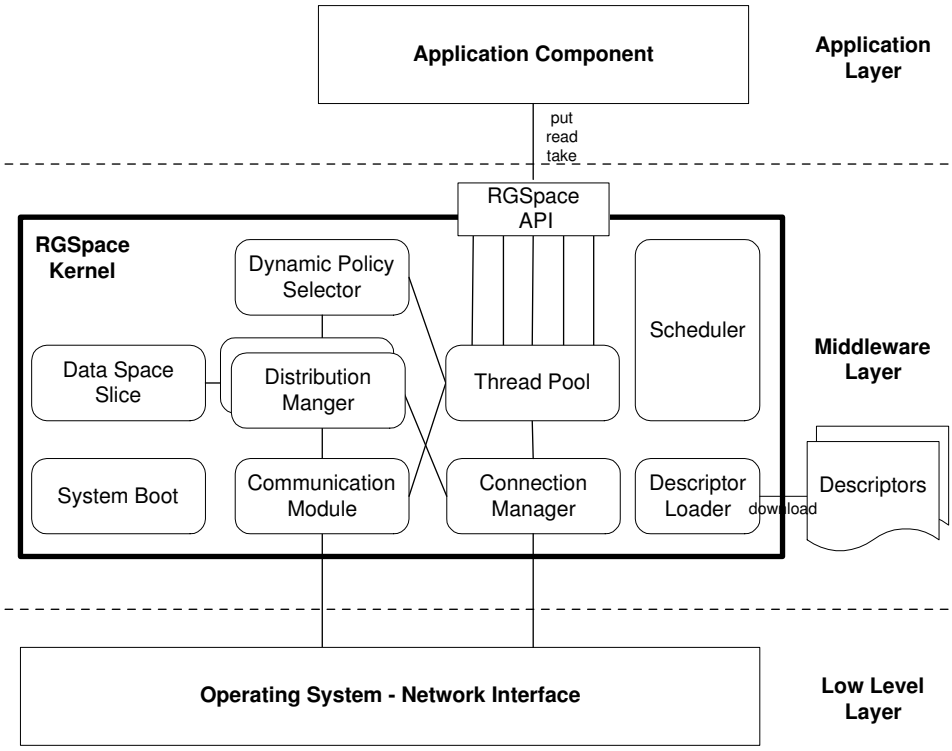


Figure 5.3: Internal structure of a RGSpace kernel in the current implementation.

same type and operator for RGSpace; as long as they are initially sent to different kernels, the kernel identifier can be used alongside the tuple type and operator to distinguish requests.

The `Kernel` class also has three tables (maps) which contain the distribution policies, timing constraints and error handlers:

- `distributionPolicies` is a mapping of tuple types to names of Distribution Managers.
- `timingConstraints` is a mapping of requests to timing constraints.
- `errorHandlers` is a mapping of requests to error handlers.

Finally, for profiling purposes, there is a field of type `MicroClock`. This clock can get and set the current time, in microseconds accurate.

Class `SystemBoot` has two methods:

- `boot` initializes the variables of the `Kernel` class, and then blocks until RGSpace is shut down.

- `halt` shuts down the kernel.

Because the `Kernel` variables are static, these methods are static as well. This way there is no need to instantiate the System Boot.

5.2.2 API

The API module is used on the application component side, and is not an actual part of the kernel. Application components use an instance of class `RGSpace` to communicate with RGSpace. This class provides the `read`, `take` and `put` operations to the application components.

The actual sending of messages (request) between application components and an RGSpace kernel is done using native UNIX sockets. This way, the network stack will not be used for communication between application components and kernels, so it can be used exclusively for communication among kernels (see Section 5.2.3). As a result, application components need to be located on the same physical machines as a RGSpace kernel.

To be able to use native UNIX sockets a Java wrapper has been written: class `LocalServer` creates the UNIX socket upon instantiation, and class `LocalClient` connects to such a socket upon instantiation. The common name is based on the request that uses the socket. The `RGSpace` class uses client instances, the threads inside the thread pool use server instances.

Both classes have two public methods: `writeObject` sends an object from the server to the client or vice versa, and `readObject` receives a sent object. Because a thread cannot wait forever for an application to send a request, it has a timeout for the reading. However, since tests have shown these timeouts are not accurate[6], the timeout is currently set to 0. A `read` or `take` will always return before a new request[7], so a read on the client side is blocking.

5.2.3 Communication Module

The `CommunicationModule` class is responsible for the communication between kernels. Where in GSpace (see Chapter 3) TCP/IP sockets are used for communication, this is not suitable for real-time applications/systems such as RGSpace. Even with Quality of Service turned on, it cannot guarantee the arrival of messages within a given deadline. Therefore, some real-time network protocol is needed.

Although we have looked at different protocols, such as Controller Area Network (CAN) and isochronous transfer over IEEE1394 (FireWire), we

[6]For example, if a timeout of 10ms was specified, it often took near 20ms to return from the call.

[7]In case of a missed deadline, the error handler must ensure this!

have chosen to have a new network protocol designed and implemented for us, so it will match RGSpace's needs. This real-time network protocol is called **T**oken based **R**eal-time l**I**ght **P**rotocol over Ethernet (TRIP) [11].

As the name implies, TRIP is token based. Only the node with the token can send messages, either point-to-point or broadcasted, ensuring no message collisions occur on the network[8]. Because messages can only be sent when the node has the token, TRIP has a sending buffer; applications put messages inside the buffer, and TRIP sends the messages from the buffer when it has the token. There is also a receiving buffer to store messages read from the physical network by TRIP but not yet by the application that is using TRIP. TRIP can also calculate whether a message can be delivered within a given deadline; if it can it will be, or otherwise it will not be sent at all. This behaviour resembles acceptance tests for aperiodic requests.

TRIP will periodically turn the token around once; at the start and ending of each turn the first node will have the token. Since it is periodical, we will create a thread for it which will be scheduled by the scheduler, and run alongside request handling threads.

To make it easier to replace TRIP by a similar network protocol, we have created an abstract class called `RealtimeNetwork`, with TRIP as a sub class. In the rest of this chapter, we will talk about `RealtimeNetwork` instead of TRIP.

The `CommunicationModule` class has a field of type `RealtimeNetwork`, which is responsible for the actual sending. To interact with this network, `CommunicationModule` provides the following methods:

- `read` reads a tuple of a given type from the real-time network's receiving buffer. If there is no such tuple present in the network's receiving buffer the reading thread will block until such a tuple is present.
- `write` puts messages in the network's sending buffer. There are two versions: one for broadcasting and one for point-to-point communication.
- `awaken` reads all messages from the network's receiving buffer and wakes up all threads blocking on a `read` if a matching tuple is read. It is called after each network turn.

Unlike the `Controller`, the `Communication Module` is not removed. Whereas communication between application components and kernels can consist of multiple connections over UNIX sockets, this is not possible for the communication among kernels; there is only one "connection" over the

[8]Provided the network is isolated, which is a requirement for TRIP.

real-time network. Therefore, the `Communication Module` does need to distribute the messages over the threads that are waiting for them. We could have adopted the same technique for the `Controller` but that would be less efficient; there would be need for another thread (the `Controller` thread), with a short period to ensure threads are not waiting too long. Unfortunately, this long waiting cannot be eliminated in the communication among kernels, although the real-time network's period can be as short as possible.

5.2.4 Scheduler

The scheduler is the `javax.realtime.PriorityScheduler`, as provided by jRate (see Section D.2). This scheduler (the only scheduler available at present) will schedule threads according to priorities: at any time the thread with the highest priority that can be active will be active. This scheduler interacts with the operating system scheduler directly to ensure threads start in time. The necessary parameters, such as priority and deadline, are passed through the initialization of jRate's `javax.realtime.RealtimeThread` (see Section 5.2.5 and Section D.2).

Unfortunately, jRate's `PriorityScheduler` does not implement a schedulability test. Therefore, this test is moved to the `ThreadPool` class (see Section 5.2.5), since this is the only other class that has to deal with the threads.

5.2.5 Thread Pool

The Thread Pool module consists of the `ThreadPool` class and the threads inside this pool. These threads are instances of `javax.realtime.RealtimeThread`, as provided by jRate (see Section D.2). They provide their own interaction with the scheduler, also provided by jRate (see Section 5.2.4 and Section D.2). They are initialized with a set of parameters such as priority, period, cost and deadline. They provide two methods that are used in RGSpace:

- `waitForNextPeriod` will block until the start of the next period unless the thread has missed a deadline. It is called *before* every instance to ensure the first instance does not start too early.
- `schedulePeriodic` must be called by an error handler (see Section 5.2.6) when the thread has missed a deadline. It will then be scheduled as normal again.

`ThreadPool` has the following methods:

- `addThread` creates and adds a thread for a specific request.

- `addNetworkThread` creates a thread for the real-time network (see Section 5.2.3).
- `assignPriorities` assigns priorities to the threads, according to deadlines. This is necessary because of the used scheduler (see Section 5.2.4) and the fact that constraints do not have to be ordered in the descriptor.
- `startThreads` starts all threads.
- `stopThreads` tells all threads to stop.
- `waitForEnd` blocks until all threads have ended.
- `schedulable` returns whether the set of threads is schedulable. This should be part of the scheduler, but the used scheduler does not implement a feasibility test (see Section 5.2.4).
  Please note that this schedulability test does not take into account the time application components need for their computations. Ideally, RGSpace is run on machines with multiple processors; one can be used for the RGSpace kernel, and the other processors can be used for components.

This schedulability test is the test from Section B.3. This test needs the period, deadline and worst-case execution time (WCET) of the tasks. The period and deadline can be retrieved from the constraints, but the WCET cannot. Instead, this WCET is calculated: it is a summation of the WCET's of the individual actions that together make up the entire handling of a request. Typical actions are looking in the local slice or sending a message to another kernel.

Since the actions to take are dependent of the distribution policy, the Distribution Manager calculates this policy-dependent part of the WCET. For instance, if the distribution policy dictates that only the local slice needs to be searched for a matching tuple, the WCET will be a lot shorter than if the request needs to be forwarded to one or more other kernels.

Because the network sends messages periodically, the actual sending may be delayed. Since the network will immediately transfer the token if there are no messages, the worst-case time before a message is actually sent is the network's period[9]. This time is included in the total WCET for every time a message needs to be sent between kernels.

Besides the previously mentioned parameters, the threads are also initialized with instances of `RGSpaceLogic`. This is an abstract base class for all

[9]If the message is sent immediately after sending the token, the network thread will not get active for a time nearly its period.

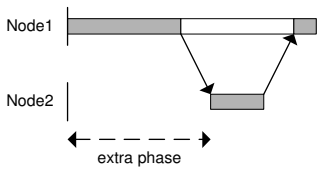


Figure 5.4: Different phases for two kernels. The white piece means the thread is waiting for a reply. The network thread is not displayed.

thread logics inside RGSpace, and implements `java.lang.Runnable`. Besides the necessary `run` method, it also has a `stop` method to stop the logic (and therefore the thread). For handling requests on the kernel that received the request from an application component, sub class `LocalRGSpaceLogic` of class `RGSpaceLogic` is used.

If a request will be forwarded to another kernel, this other kernel must have a thread for handling this request. This thread's logic is another sub class of `RGSpaceLogic`, called `RemoteRGSpaceLogic`. It first reads a request from the `Communication Module` (see Section 5.2.3) and then passes the request to the `Dynamic Policy Selector` (see Section 5.2.9). If necessary, the answer is sent back through the `Communication Module`.

These remote threads must be scheduled in such a way that the match the local thread on the initiating kernel; otherwise one or both could be waiting unnecessarily, and even miss their deadlines. We have come up with two solutions:

- The local thread is scheduled as usual. The remote thread is scheduled with a different phase (see Section B.1). This phase will be the phase of the local thread (usually 0) plus the time the local thread needs to deliver the message (see Figure 5.4). This way, processor time can be optimally used. However, it is more difficult to determine the schedulability, since all kernels have different schedules. To determine overall schedulability extra communication is needed. Also, the overall deadline has to be divided between both threads; the remote thread must have an earlier deadline than the local thread (in case of a `read` or `take`). How is the best deadline for the remote thread calculated?

- Both local thread and remote thread are scheduled as if they are equal. They have the same phase, period, priority, deadline and worst case execution time (WCET). This WCET is the worst case time for completing the entire handling of the request on all kernels. Figure 5.5 shows an example of this policy. This will most likely waste processor time, since actions taken on only one kernel are scheduled for all kernels. This is especially apparent if a kernel does not take part in the

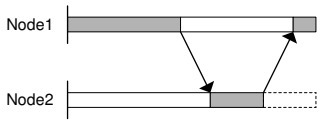


Figure 5.5: Threads scheduled as if they are equal. The white piece means the thread is waiting for a reply. The dashed piece is scheduled but not used by this thread. The network thread is not displayed.

handling of the request. Although it will consume no processor time, it is scheduled with the cost of handling a request. However, schedulability is a lot easier to test: all kernels have the same schedule, and therefore only have to test their own local schedule.

We have chosen for the second policy. Not only because it is easier to test for schedulability, but also because we are interested in predictability over performance. The wasted processor time is not that bad when the schedulability test (and therefore predictability) is better and easier.

Another sub class of `RGSpaceLogic` has been created for kernels that do not take part in the handling of a request: `IdleRGSpaceLogic`. Upon activation, it will not do anything at all, and its scheduled execution time becomes idle time or is used by another thread. Finally, to match it with other threads in RGSpace, we have made `RealtimeNetwork` (see Section 5.2.3) a sub class of `RGSpaceLogic` as well.

With this distributed scheduling policy, it is essential that threads start at the same time. Therefore, all kernels must be synchronised. The first time the `Connection Manager` (see Section 5.2.7) takes care of this, so all threads start at the same time. To keep the kernels synchronised the network protocol should be used, since synchronisation requires network messages. The idea was to put the entire synchronisation inside the network protocol, but unfortunately we had no time to adapt it. As a result, the machines will slowly get out of synch.

Please note that if a thread's execution on one kernel ends earlier than its matching thread on another kernel, this does not matter. If another thread becomes active immediately it may have a longer execution time than scheduled (because the receiving thread may not yet be ready), but it will be executing within the processor time of the first thread. See Figure 5.6 for an example.

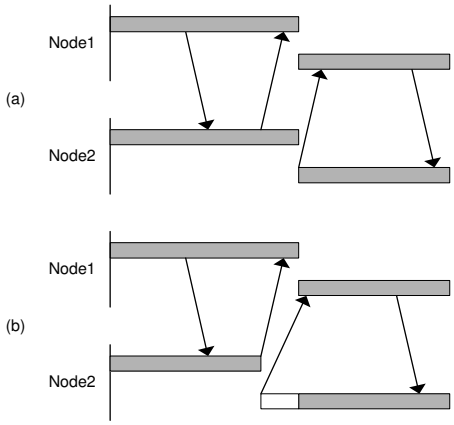


Figure 5.6: What happens if one thread's execution ends early. (a) The ideal situation. (b) A more real situation. The white part is processor time taken from the other thread. The network thread is not displayed.

5.2.6 Error Handlers

All threads in the thread pool (see Section 5.2.5) have an associated error handler (if one is specified in the error handler descriptor). This error handler will be activated when a thread misses its deadline.

Error handlers must extends abstract base class `ErrorHandler`. This is an implementation of `java.lang.Runnable`. All error handlers must implement the required `run` method; this method is the error handling method.

5.2.7 Connection Manager

The used real-time network protocol, TRIP (see Section 5.2.3), requires a static number of nodes. As a result, it is not possible to add or remove RGSpace kernels. The `ConnectionManager` class therefore does not need to update the number of kernels. As a result, its role changes drastically: it will no longer (periodically) listen for new kernels to arrive or for existing kernels to leave. Because of the synchronisation needs (see Section 5.2.5) that will be its only purpose.

5.2.8 Distribution Managers

Distribution Managers must implement the `DistributionManager` interface. This interface requires the following methods:

- `read`, `take` and `put`, for the three RGSpace operators.
- `externalRead`, `externalTake` and `externalPut`, for handling requests from another kernel. These methods will usually be a lot different

from the "normal" operators. For example, a Store locally Distribution Manager will, on the initial kernel, first search the local slice and if that fails pass the request to other kernels. In this case, the `external` methods will only search the local slice.

- `isUsed`. In case of e.g. a Push-to-one Distribution Manager, not all kernels will be needed to handle a request. This method returns whether an RGSpace kernel is going to be used for handling the request.

5.2.9 Dynamic Policy Selector

The DPS module also consists of one class, `DynamicPolicySelector`. It has the same methods as the `DistributionManager` interface, as well as `createDistributionManager` and `getDistributionManager`, which respectively create a Distribution Manager from a string and return the Distribution Manager for a tuple type. Upon invocation of one of the operators it will simply pass the request to the appropriate Distribution Manager.

5.2.10 Data Space Slice

The `Slice` class has methods `read`, `take` and `put` for the usual reading, taking and putting. However, it also has three additional methods, `singleRead`, `singleTake` and `overwrite`. These methods should be used if at most one tuple of a specific type is needed. Putting a tuple overwrites the previous tuple instead of adding the new tuple (hence the name `overwrite`). `singleRead` returns this tuple, regardless of the contents, and `singleTake` removes this tuple. So, after a `singleTake`, there will be no more tuple until an `overwrite`. The two sets of methods should not be mixed!

5.2.11 Descriptor Loader

The descriptor loader consists of class `DescriptorLoader`, which has three methods for reading and storing the different policy descriptors:

- `loadDistributionPolicies` loads the distribution policy descriptor. This descriptor has one line for each tuple type. This line looks like this:

  ```
  tuple class:distribution manager class
  ```

  These classes should be full class names, including Java packages.

- `loadTimingConstraints` loads the timing constraints policy descriptor. This descriptor has one line for each constraint. This line looks like this:

```
kernel ID:operator:tuple class:phase:period:deadline
```

Again, the tuple class should be the full class name.

- `loadErrorHandlers` loads the error handler descriptor. This descriptor has one line for each error handler that is declared for a periodic request[10]. This line looks like this:

```
kernel ID:operator:tuple class:error handler class
```

The classes should again be the full class names.

These methods have direct access to the three tables of class `Kernel` (see Section 5.2.1), and fill these tables as the descriptors are read. `loadErrorHandlers` also initializes the read error handlers.

5.2.12 Types

The `Tuple` class is the base class for all tuples for RGSpace. It has two fields, `__id` and `__homeAddr`, which are for possible future use (they are copied from GSpace's tuples). The former should be a unique identifier, the latter an identifier for the kernel that is regarded as the "home" of the tuple (i.e. the kernel . Caching Distribution Managers may need this. Because the data space slice or Distribution Managers may need to access these fields, they are public. For the same reason, **all** fields of sub classes of `Tuple` need to be public.

The `Request` class represents requests, and is used as communication between kernels. It has the following fields:

- `operator`: the operator for the request: `read`, `take` or `put`.
- `tuple`: the tuple to put or the template used for reading or taking.
- `tupleClass`: the tuple type of the request, as a Java class. This is needed because the `tuple` field may become `null`.
- `kernel`: the identifier of the kernel on which the request was initiated (see Section 5.2.1).
- `deadline`: the deadline of the request. It is used only for sorting purposes.

[10] It is possible to have a periodic request without a error handler, although it is not recommended.

`Request` equality will ignore the `tuple` and `deadline` fields, and is based on operator, tuple type and kernel identifier. There will be one thread for each different `Request` instance.

Since it contains more data than just a tuple, and has to be sent through the `Communication Module` anyway, `Request` objects are used for communication inside a kernel.

Finally, there is the `Constraint` class, which simply stores constraints from the timing constraints descriptor: period, deadline and phase.

5.3 Control Flow in RGSpace

Figure C.13 of Appendix C shows an example of the control flow of a `read` operation using a Push-to-one distribution policy where the tuple type is pushed to another kernel. When an application component sends a template to RGSpace through an `RGSpace` instance, it will write this template to the matching UNIX socket through the `LocalClient` instance. The `RGSpace` instance will then read the result from the UNIX socket, blocking until the result is available.

When the thread for this request inside the `ThreadPool` is activated by the scheduler, its `LocalRGSpaceLogic` will read the template from the UNIX socket through the `LocalServer` instance. It will then pass the template, wrapped inside a `Request` object, through the `DynamicPolicySelector` to the Push-to-one Distribution Manager. This Distribution Manager will send the `Request` object to the `CommunicationModule`, which puts it inside the sending buffer of the `RealtimeNetwork`. Next it will block on reading the result from another kernel. Upon returning, it will return this result back to the `LocalRGSpaceLogic`, which writes it back to the UNIX socket. The application component will then return from the blocking read.

When the `RealtimeNetwork`'s thread is actived it will send the messages in its sending buffer and transfer the token. It will then receive and store messages in its receiving buffer until it gets the token back, and calls the `CommunicationModule`'s `awaken` method. This will read messages from the `RealtimeNetwork`'s receiving buffer, waking up the necessary threads, which will then return from their blocked reading of results.

Figure C.14 of Appendix C shows this example from the perspective of the kernel with the tuples. When the thread inside the `ThreadPool` is actived (at the same time as the thread on the other kernel), its `RemoteRGSpaceLogic` will read the template (wrapped in an `Request` object) from the other kernel, blocking because it is not yet available. It will return when the

`RealtimeNetwork` wakes up the thread. The template is then passed through `DynamicPolicySelector` to the Push-to-one Distribution Manager, which reads a matching tuple from the `Slice`. This tuple is then returned to the `RemoteRGSpaceLogic`, where it is wrapped into an `Request` object again and sent back to the initiating kernel.

5.4 Results

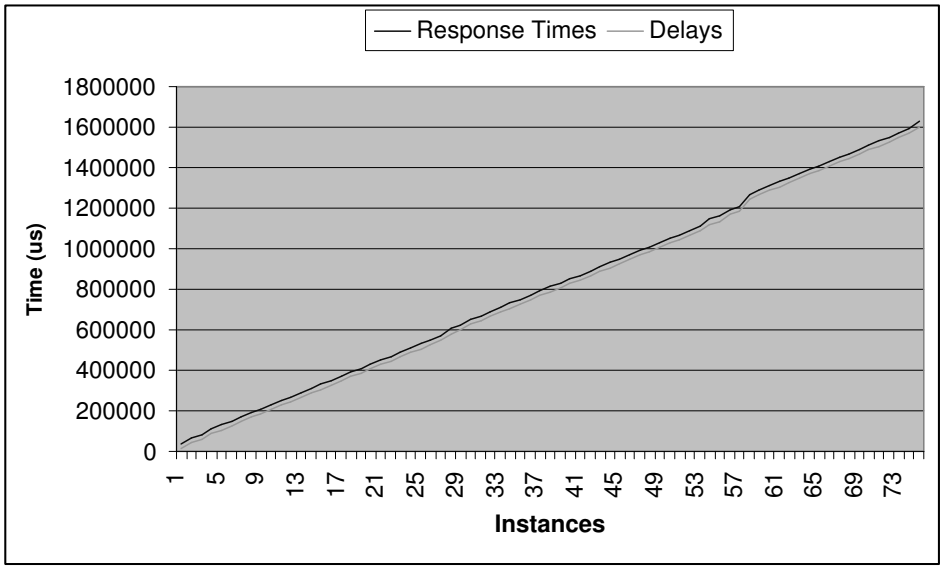
Our test consisted of only one producer and one consumer. The producer produced its data every 200ms, and the consumer read this data every 500ms. A Push-to-one Distribution Manager was used, with data stored on the producer. The network thread's period was 150ms. Deadlines were equal to periods.

Figure 5.7 shows the response times and delays on the producer and consumer. The response time is the time between activation and end of an instance of a task, the delay the time between activation and actual start. The activation times are calculated using the initial phase (see Section B.1).

As can be clearly seen, the response time increases steadily. In fact, after approximately 150s, the response time in the consumer had increased to over 6s, and the response time in the producer was even higher at 15s. This is of course not very good behaviour.

However, if you take a closer look at Figure 5.7, you can also see the delay increases just as steadily. In fact, if we subtract the delays from the response times, we get the difference between start and end times (see Figure 5.8). These times are far more constant; we believe the peaky behaviour on the consumer is caused by preemptions and waiting for messages to be sent.

Because of this, we have examined the starting times closer. Figure 5.9 shows the differences between sequential instances on the producer and consumer. These figures show that the difference between starting times match the periods of 200ms and 500ms quite well, being "only" 20ms higher on average. We therefore believe that the increasing response times should be blamed on the scheduler; apparently it does not activate threads based on the activation time of the previous instance, but on the actual starting time of that instance. One explanation could be the fact that the operating system (TimeSys Linux, see Appendix D) is not a pure real-time operating system, but a general purpose operating system with real-time extensions. Licensing problems have even limited the number of extensions we could use. A pure real-time operating system such as VxWorks or QNX should provide better results.



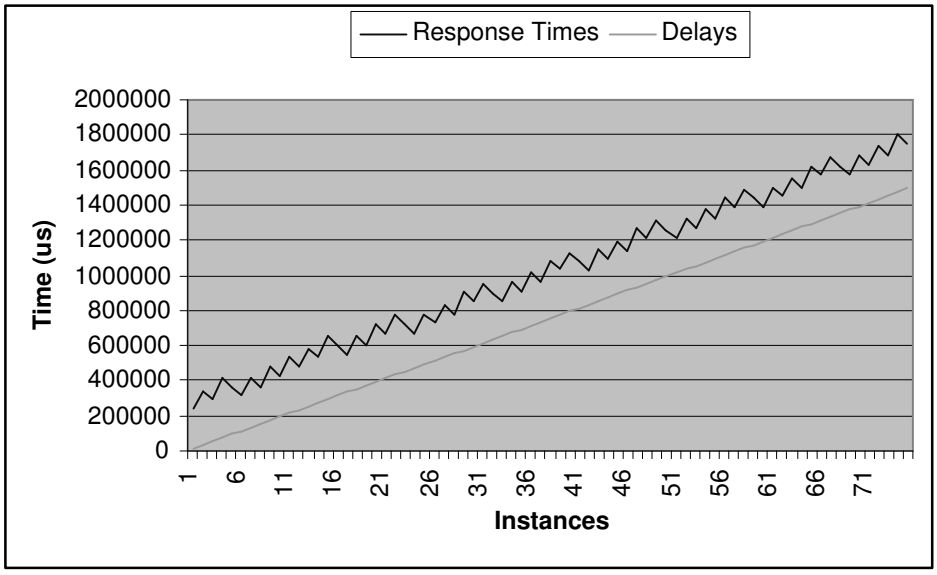
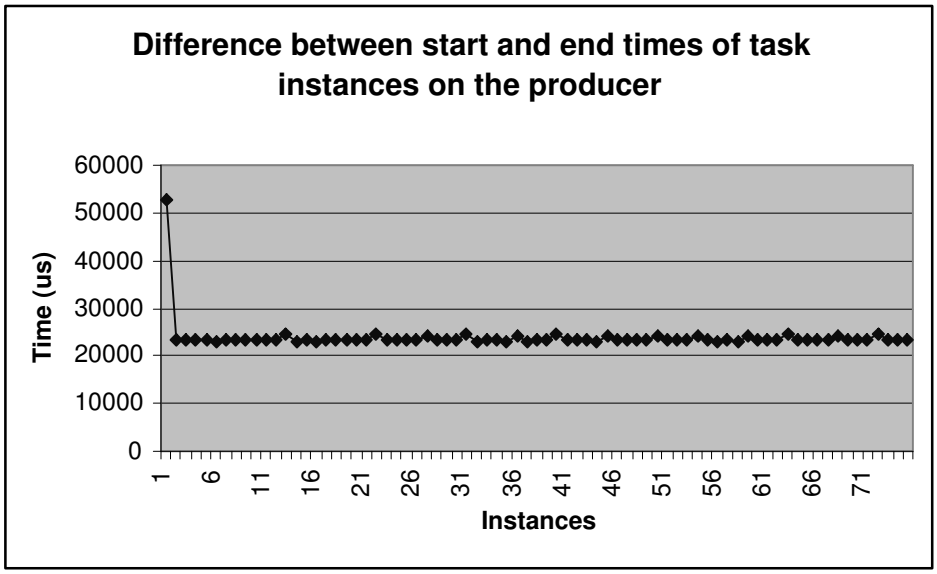


Figure 5.7: Theoretical response times and delays on the producer (upper diagram) and consumer.



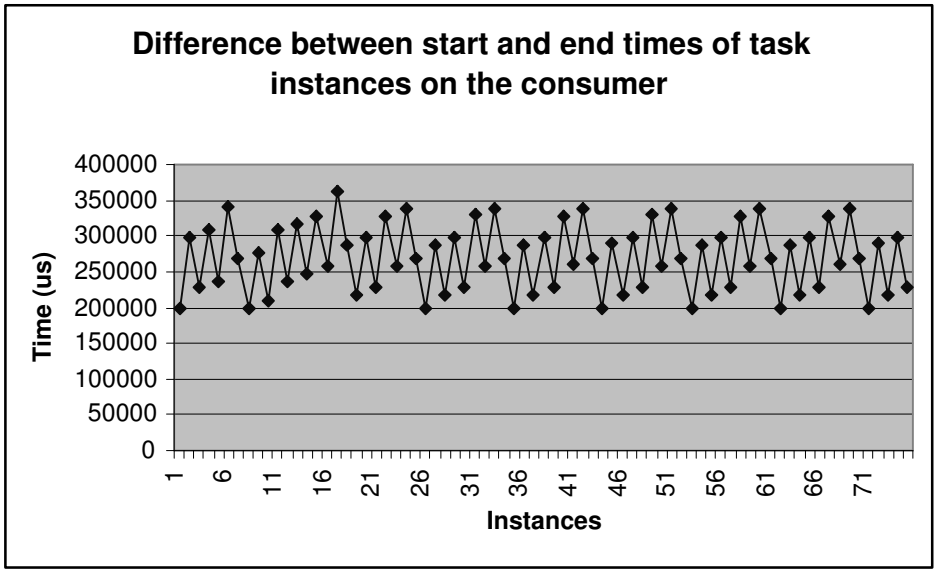
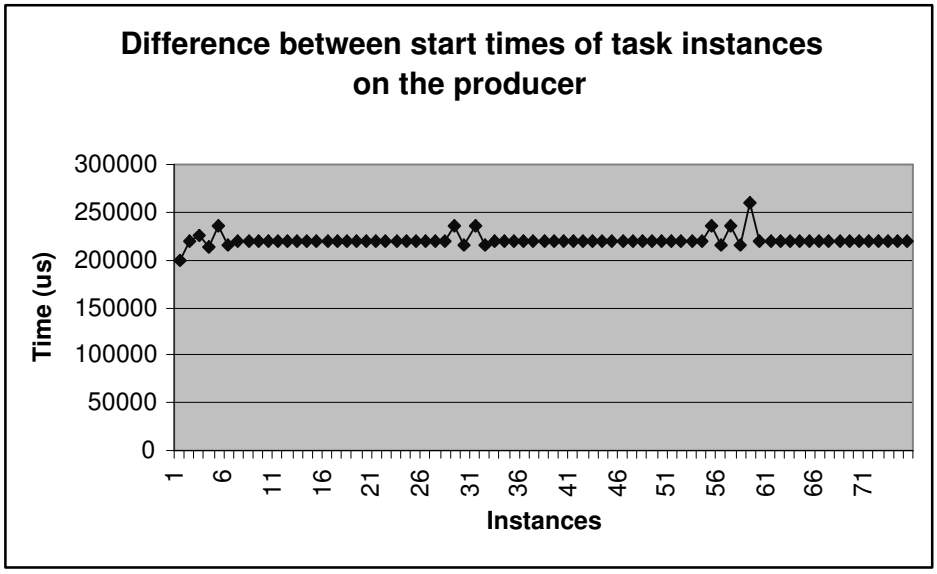


Figure 5.8: Differences between start and end times on the producer and consumer.



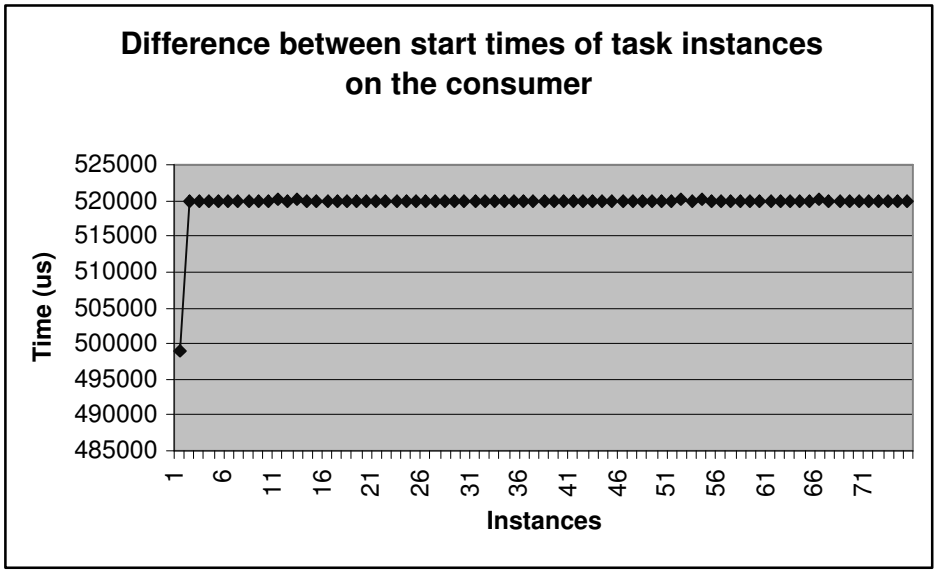


Figure 5.9: Differences between start times on the producer and consumer.

5.5 Differences between GSpace and RGSpace

Although RGSpace is based on GSpace, introducing the timing requirements also introduces necessary differences. The following is a list of some of those differences:

- GSpace can handle both periodic and aperiodic tasks, all in a best-effort mode. The current RGSpace implementation can only handle periodic tasks.
- Where `read` and `take` operations in GSpace are blocking, waiting for a matching tuple to be `put` would lead to missed deadlines. Therefore, `read` and `take` operations in RGSpace are non-blocking; if no matching tuple is present immediately, `null` is returned.
- For scheduling a set of tasks, the worst case execution time (WCET) of each task is needed. Most Distribution Managers will search the local slice for tuples. The time needed for this search is dependent on the number of tuples inside the local slice. If the number of tuples inside the local slice would be theoretically unlimited, as is the case in GSpace, the WCET for searching the local slice would be unknown. In order to be able to calculate this WCET, an upper bound on the number of tuples inside the local slice is needed. Therefore, in the current RGSpace implementation, each local slice has a fixed maximum number of tuples. The WCET for searching the local slice now is the execution time it takes to search a completely filled local slice.
- Another unknown factor in GSpace is the number of kernels. Some Distribution Managers need requests to be passed to other kernels one at a time (e.g. Store locally, see Table 3.1). To be able to know the WCET for passing these requests, the number of kernels needs an upper bound as well.
  For this reason (and the fact that the used network protocol only supports a static number of nodes), the number of kernels has to be bound to a fixed value during execution. As a side effect of this design decision, the `Connection Manager` no longer needs to perform its task periodically, relieving the strain on the schedule.
- In GSpace, the distribution of tuples can change at runtime. During this transition all requests are frozen until the transition period has finished (see Section 3.3). Adopting the same technique in RGSpace could lead to missed deadlines, and is therefore unacceptable. Another option is to regard the transition as a task and schedule it. However, there will only be a transition if a distribution policy has changed. It would therefore make sense if it were an a-periodic task. RGSpace does not support a-periodic tasks however.

It is possible to create a periodic task for the transition. If there is no need for the transition, it can simply skip the execution. There are some drawbacks however:

- The scheduler needs worst-case execution times (WCET). The WCET for transition tasks is very pessimistic[11], and therefore relatively long. Thus, scheduling the periodic transition task will waste a lot of processor time, especially if the execution is often skipped.
- There is a transition for every tuple type. This means that there will be many transition tasks, if there are many tuple types. Combined with the previous drawback, it may lead to a lot of unnecessary idle time.
- If the transition is a scheduled task, it may be preempted by handling requests, causing the system to enter an inconsistent state. This could be remedied by giving the transition task a higher priority than other tasks, so it will not be preempted. However, since the current implementation uses deadline monotonic scheduling, this means that the transition task would have to get a shorter deadline than other tasks. Given the probable high execution time for the transition task, it would either not be schedulable, or other deadlines would be quite long.
- What would be a proper policy if a transition task would miss its deadline? If the task instance is aborted RGSpace would enter an inconsistent state. However, continuing the instance could lead to other tasks missing their deadlines.

As a result, chancing distribution policies at runtime is not possible in the current RGSpace implementation.

[11] It is the execution time for the longest possible transition. If instead another transition is performed, the execution time could be a lot shorter.

Chapter 6

Related work

RGSpace is not the first real-time distributed middleware. Most existing middlewares are quite mature already, and some are even deployed in critical systems. The following sections describe some real-time middlewares. We will focus on data distribution (if appropriate) and timing aspects.

6.1 SPLICE

Hollandse Signaal Apparaten B.V. has worked on its own implementation of a distributed real-time shared data space, called **SPLICE**. It was developed for mission-critical environments that require real-time performance, scalability and fault-tolerance. SPLICE has been used in the development of commercially available command-and-control, and traffic management systems.

The SPLICE architecture consists of two types of component applications and a shared data space. The data in this data space are called *data sorts*. These data sorts are structured as labeled records, each with a number of typed fields. These fields can have primitive types such as numbers or strings, or more complex types such as arrays and nested records. Just as in databases, data sorts can have *keys* to uniquely identify instances.

Applications can interact with the shared data space through two operations: `write` and `read`. The former inserts a data sort into the data space, the latter reads an instance of a data sort, satisfying an optional query. This query is a boolean expression over the fields of the data sort, and evaluates to `true` if absent. This `read` operation is blocking until a matching data sort instance is found.

As can be seen, there is no operator for removing data from the shared data space. The reason for this is the fact that "data in SPLICE is regarded as shared information that can be freely consulted by any number of appli-

cations. [...] The coordination model of SPLICE is based on monotonic (temporal) reasoning where information, once established, never becomes invalid"[3].

Data distribution in SPLICE is based on a subscription paradigm. Users subscribe to data once, and every time a new data sort is inserted into the shared data space it is forwarded to all known users of that data sort. As a result, data is selectively replicated across the nodes in the network. Each node stores only instances of those data sorts that are actually read or written by applications that communicate through that node.

"In embedded systems data gradually loses its value as the environment changes and time evolves. Hence, the application processes usually require only a limited temporal view on the overall contents of the shared data space"[3]. Any data that falls outside this temporal view will be no longer of interest and can be safely removed from the shared data space. Data will also be removed if a newly received instance of a data sort has the same key values as another instance. If that is the case, the new instance will be more recent, and the older value will be overwritten.

6.2 TAO: real-time CORBA

One of the best known middlewares is **CORBA**. This platform and programming language independent middleware resides between clients and servers. "It simplifies application development by providing a uniform view of a heterogeneous network and OS layers"[14].

CORBA does not communicate through data, but through Remote Procedure Calls (RPC). Clients can retrieve references to servers through CORBA, after which they can use these references to invoke methods on the servers through a CORBA *Object Request Broker* (ORB). Upon invocation, the method and parameters are converted into a common data-level representation. They are then sent through the network to a server, where they are converted back into typed parameters. On this server, the method is then executed and the result, if any, is sent back to the client. There is no data distribution in CORBA.

CORBA works fine for "conventional RPC-style applications that possess "best effort" quality of service (QoS) requirements"[14]. However, conventional CORBA implementations lack QoS specification interfaces, QoS enforcement and real-time programming features. To this end, *The ACE ORB* (TAO) has been developed by the Department of Computer Science, Washington University, St. Louis, USA [14]. "TAO is a high-performance, real-

time CORBA-compliant ORB endsystem developed using the ACE framework, which is a highly portable OO middleware communication framework. ACE contains a rich set of C++ components that implement strategic design patterns for high-performance and real-time communication systems"[14].

TAO has a *Scheduling Service* for allocating CPU resources to meet the QoS needs of the applications. This Scheduling Service guarantees that all processing requirements for hard real-time tasks are met. It performs an offline schedulability analysis of all operations that register with it. Applications must therefore specify their CPU capacity requirements to TAO's offline Scheduling Service. This is done through the *QoS API*. These requirements consist of among others the worst-case execution time and the period.

6.3 Real-time distributed databases

Although database management systems (DBMS) are not truly middlewares, they do resemble shared data spaces; there is also temporal and referential decoupling [6]. Therefore we have given them a closer look.

Distributed in distributed databases mean that the database copies reside on different nodes in a network. Full replication over all nodes means applications can access any node to retrieve this data. However, these nodes need to be kept consistent with each other. This requires extra communication, therefore increasing execution times.

The DeeDS [2] system has a less strict criteria for consistency, in order to enforce timeliness. Each node has a local consistency, while the distributed database might be inconsistent due to different views of the system on the different nodes.

In the STRIP [1] system, nodes can send (*stream*) views or tables to each other regularly. Users can select if entire tables/views should be streamed, or *delta tables* which reflect changes to the tables/views. Data can be streamed periodically when some data has reached a predefined age, or after an explicit instruction to stream.

BeeHive [15] utilizes the concept of *data deadlines*. Data elements for which the deadline has expired are no longer useful. This deadline is therefore the maximum age of the data elements. The deadline of any transaction is the maximum allowed age of the data elements used in that transaction. If a transaction cannot complete before its data deadline, it is postponed until data elements get updated, giving the transaction a later data deadline. This is called *forced wait*. When an application executes a transaction, it can also specify the real-time requirements of that transaction: the deadline,

start time and the period if it is periodic.

ARTS-RTDB [10] incorporates a new feature called *imprecise computing*. If a query is not finished when its deadline expires, the result so far can be returned to the client if the result is meaningful. For example, if a query must calculate the average of a large number of values, if at the time the deadline expires a high enough percentage of values has been calculated, the result could be considered meaningful. Although it is imprecise, it is returned to the client anyway.

In ARTS-RTDB the `INSERT`, `DELETE`, `UPDATE` and `SELECT` operations are the most critical data operations. Therefore, the system has been tried to be optimized to increase efficiency for these operations. According to [10] many real-time applications almost only use these operations at run-time.

6.4 Conclusion

Although there are other, more mature real-time distributed middlewares, they are quite different from RGSpace.

Some real-time middlewares, such as SPLICE and some real-time DBMS, have another notion of real-time. Instead of having deadlines for operations, it is the data that is timed; as time evolves, the data loses its value and is no longer of interest. An example is sensor readings, which are only useful for a short period of time.

Other middlewares such as TAO and the BeeHive DBMS that do support deadlines for operations miss the separation of concerns of RGSpace. Application components need to pass the timing constraints to the middleware, so functional and extra-functional requirements are both combined in these components. If the constraints change, the application components must change as well.

Although SPLICE does not distribute data in a fixed way, it does not support the diversity of RGSpace, where far more different distribution policies can be specified. In fact, RGSpace is one of very few real-time middlewares (if not the only) that support different distribution policies for different types.

Chapter 7

Conclusions and future work

7.1 Discussion

Section 5.4 shows that response times of requests are increasing. This is very bad for predictability, where a fixed response time is ideal. However, the increase in response times is quite stable. If the period is adjusted with this increase (i.e. the period becomes the specified period plus the steady increase), the response times will be quite stable after all. Figure 5.8 shows the difference between start and end times of task instances. These differences are the response times with the adjusted periods. Apart from one peak on both producer and consumer, these adjusted response times remain within a fixed range. Therefore, if we adjust periods to take the steady increase of response times into account, RGSpace becomes quite predictable after all.

Please note that this adjustment may not be necessary on pure real-time operating systems.

Even though RGSpace can become quite predictable, the current implementation of RGSpace is still far from mature. Performance of RGSpace is not very good at present; the minimum deadline for requests that need to look on remote kernels is still quote high. One of the biggest culprits is object conversion. Only bytes can be sent through both the network and UNIX sockets. Therefore, objects must be converted into bytes before they are sent, and received bytes must be converted back into objects.

The time it takes to convert an object into bytes and back again takes quite long; the worst-case execution time (WCET) of converting an object of size 1024 bytes[1] and back was 33ms in our tests. The more network communication is needed, the bigger the influence of this conversion. The remote `read` from the test in Section 5.4 needed four conversions: one for sending the template from the component to the kernel, one for sending

[1] This is size of the largest message the TRIP protocol can support.

the request from one node to another, and two for sending the request and answer back. This alone has a WCET of 132ms.

Another big culprit is the network protocol. Recall from Section 5.2.5 that the network period is part of the total WCET. In the test this was another additional 300ms ($2 * 150$). As a result, the total WCET was almost 450ms. For two remote `read`s or `take`s without any other periodic requests the minimum period will be near 1 second. This is quite high compared to other real-time middlewares like TAO (see Section 6.2).

Another problem is the language RGSpace is written in: Java. Although it provides an easy way to interact with the operating system scheduler using jRate (see Appendix D), it also causes another problem: in our tests, the system crashed after a few minutes every time because of a memory overflow. This is caused by Java's dynamic memory allocation and deallocation. When a Java object is no longer needed, it will remain in memory until it is garbage collected. This garbage collection has a lower priority than RGSpace threads, and therefore could not clean up enough memory in time. As a result, the memory was filled faster than it was cleared. We have taken this into account, creating objects as static as possible, but this was not always possible. We believe that rewriting RGSpace in C++ would greatly eliminate this problem, since C++ provides the explicit object destruction Java lacks.

However, the principle behind RGSpace is promising. GSpace has already shown that there is a benefit in having different distribution policies [13]. The separation of concerns allows users to change the distribution strategy without changing the code of application components. RGSpace inherits this benefit, and adds the same approach for real-time constraints. In theory, any non-functional requirement can be separated in a similar way.

7.2 Conclusions

We have presented RGSpace, a real-time distributed shared data space. We have taken the flexible distributed shared data space GSpace [12][13] as a basis, and extended it to feature real-time properties. During this extension, we have kept GSpace's separation of concerns and added the real-time concern. As a result, RGSpace enables the separation of both real-time constraints and distribution policies from application component functionality. This way, it can be used in different application domains without having to change it.

Although it still has its flaws, RGSpace has shown that most likely any extra-functional requirements can be separated from functionality, not just

data distribution. Although RGSpace can only support distribution and timing, it could be extended to support other extra-functional requirements (e.g. memory constraints or user authentication) in a similar way to the way GSpace was extended. This may finally result in probably the most flexible distributed system possible.

7.3 Future work

Apart from extending RGSpace to support other extra-functional requirements, RGSpace can be extended to provide a better support for timing constraints first.

One extension is the notion of transactions. Although GSpace doesn't have transactions either, all operations will finish eventually. This is a result of the blocking `read` and `take` operations. A transaction, which can be seen as a set of `read`, `take` and `put` operations, will therefore also finish eventually.

The same cannot be said about RGSpace, since `read` and `take` operations will not block but return `null` if there is no matching tuple. As a result, the same set of operations that will finish as expected in GSpace, may not work in RGSpace if no matching tuple is found. Scheduling transactions as one could prevent this.

RGSpace could be extended to have a variable scheduling algorithm. This should then be a parameter to the system. As a result, the current limitation of periodic requests only could perhaps be removed, if the scheduling algorithm support this.

At present the number of nodes in RGSpace is fixed (see Section 5.5). Functionality for adding and removing RGSpace kernels during runtime could be added. This would also require another real-time network protocol. Please note that, in order to predict the schedulability of operations, an upper bound on the number of nodes is still needed if distribution managers that access each node one at a time are supported.

Appendix A

Abbreviations

ACE	Adaptive Communication Environment
API	Application Program(ming) Interface
CAN	Controller Area Network
CBSE	Component-Based Software Engineering
CNI	Compiled Native Interface
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DBMS	database management system
DM	Deadline Monotonic
DPS	Dynamic Policy Selector
EDF	Earliest Deadline First
GCC	GNU Compiler Collection
GNU	GNU's Not UNIX
IEEE	Institute of Electrical and Electronics Engineers, Inc.
JNI	Java Native Interface
JVM	Java Virtual Machine
OO	Object-Oriented
ORB	Object Request Broker
QoS	Quality of Service
RM	Rate Monotonic
RPC	Remote Procedure Calls
RTOS	real-time operating system
RTSJ	Real-time Specification for Java
SPLICE	Subscription Paradigm for the Logical Interconnection of Concurrent Engines
TAO	The ACE ORB
TRIP	Token based Real-time lIght Protocol over Ethernet
WCET	worst-case execution time

Appendix B

Scheduling

When a set of tasks has to be executed on a single processor, they need to be *scheduled*: the available processor time has to be assigned to each task. These assignments depend on the *scheduling policy*.

Some scheduling algorithms may *preempt* the current task: if a more important task should become active, the current task is interrupted. The more important task will then become the current task.

Real-time tasks can be either *periodic* or *aperiodic*. Periodic tasks consist of an infinite sequence of similar executions (*instances*) that are regularly activated at a constant rate. Aperiodic tasks also consist of an infinite sequence of executions (*requests*), but their activation times are not regular.

Since the current implementation of RGSpace is limited to periodic tasks only (see Section 5.2), we will describe scheduling algorithms for periodic tasks only. The next sections will first give some definitions, followed by some of those algorithms. All given examples are taken from [5].

B.1 Definitions

Periodic tasks τ_i can be characterized by the following parameters:

- **Arrival time** $a_{i,j}$ is the time at which the jth instance $\tau_{i,j}$ becomes ready for execution. It is also called the **release time** $r_{i,j}$.
- **Period** T_i is the interval between to consecutive activations of τ_i. In other words, $T_i = a_{i,j+1} - a_{i,j}$ for any $j \geq 1$.
- The **phase** $\Phi_i = a_{i,1}$
- **Computation time** C_i is the worst-case time needed for executing any instance without interruption. It is the same for all instances.

- **(Absolute) deadline** $d_{i,j}$ is the time before which $\tau_{i,j}$ should be completed.
- **Relative deadline** $D_i = d_{i,j} - a_{i,j}$ for any j. It is the same for all instances.
- **Start time** $s_{i,j}$ is the time at which $\tau_{i,j}$ starts its execution.
- **Finishing time** $f_{i,j}$ is the time at which $\tau_{i,j}$ finishes its execution.

The set of periodic tasks Γ can generally be denoted as

$$\Gamma = \{\tau_i(\Phi_i, T_i, D_i, C_i), i = 1, \ldots, n\}$$

The release time $r_{i,k}$ and absolute deadline $d_{i,k}$ of any instance k can then be computed as

$$\begin{aligned} r_{i,k} &= \Phi_i + (k-1)T_i \\ d_{i,k} &= r_{i,k} + D_i \end{aligned}$$

Another parameter that can be deducted is the **response time** of an instance k: $R_{i,k} = f_{i,k} - r_{i,k}$.

For periodic tasks, the *processor utilization factor* U provides a measure of the computational load the periodic task incurs on the processor. Since the fraction of processor time spent on task τ_i is C_i/T_i, the utilization factor for tasks $\tau_1, \ldots, \tau_n$ is

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{B.1}$$

B.2 Rate Monotonic

Rate Monotonic (RM) can be used when relative deadlines are equal to the periods: $D_i = T_i$. In RM, tasks with a higher request rate (shorter periods) will have higher priorities. Since periods are fixed, priorities are fixed. As a result, a set of tasks can be tested for schedulability offline. Furthermore, RM is preemptive.

[5] shows that a task set is schedulable if the utilization factor $U \leq n(2^{1/n} - 1)$. For high values, the right hand side converges to $ln\ 2$. If U is between $n(2^{1/n} - 1)$ and 1, nothing can be said about the schedulability of the set.

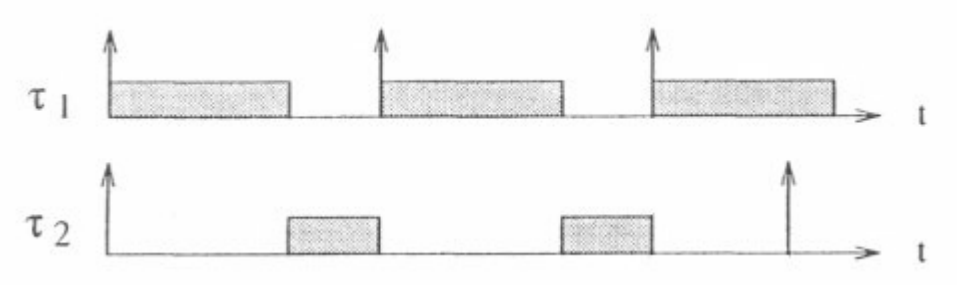


Figure B.1: An example of an RM schedule.

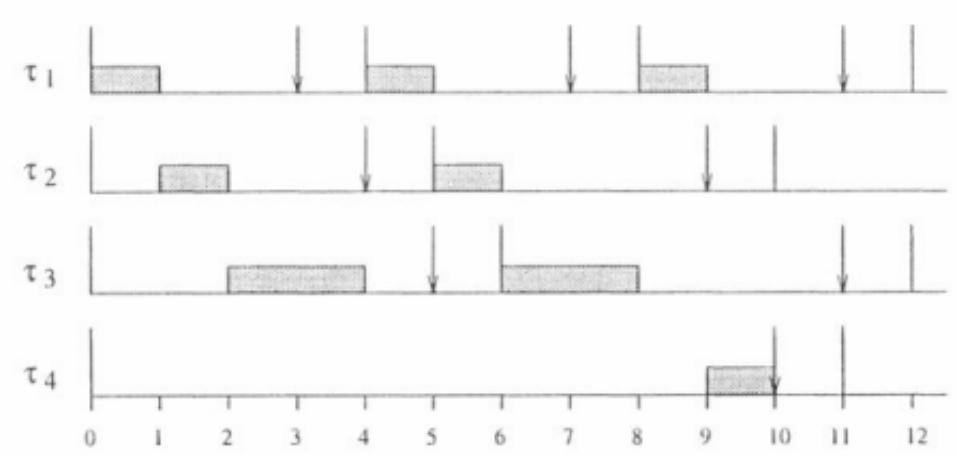


Figure B.2: An example of a DM schedule.

B.3 Deadline Monotonic

Deadline Monotonic (DM) is much like Rate Monotonic (see Section B.2). However, in DM, relative deadlines and periods need not be the same. Instead, $C_i \leq D_i \leq T_i$. (Fixed) priorities are no longer assigned based on periods but on relative deadlines; the shorter the relative deadline, the higher the priority. Schedulability can now be guaranteed if

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1)$$

However, this test is quite pessimistic. A less pessimistic schedulability test can be derived by noting that for each task τ_i, the sum of its processing

time and preemption must not be greater than D_i.

If tasks are ordered on relative deadlines (that is, $i < j \Leftrightarrow D_i < D_j$), the new test then becomes

$$\forall i : 1 \leq i \leq n \qquad R_i \leq D_i$$

where R_i is the smallest solution of x satisfying

$$x = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{x}{T_j} \right\rceil C_j$$

This R_i is the longest response time of a periodic task τ_i. [5] shows both how this test was devised and an algorithm for this test.

B.4 Earliest Deadline First

Earliest Deadline First (EDF), more formally known as Horn's algorithm, is a scheduling algorithm that can be used for both periodic and aperiodic tasks. As the name implies, the task with the earliest (absolute) deadline is active at any time. Because of this, priorities are dynamic: at any time, the task with the earliest deadline has the highest priority.

EDF for periodic tasks comes in two variants: one with deadlines equal to periods ($D_i = T_i$) and a less strict variant, where $D_i \leq T_i$. [5] shows that in the first variant, a task set is schedulable if the utilization factor $U \leq 1$. This difference with RM is caused by the changing priorities; Figure B.3 shows a schedule that is schedulable under EDF but not under RM. The utilization factor in this example is

$$U = \frac{2}{5} + \frac{4}{7} = \frac{34}{35} \approx 0.97$$

In contrast, for $n = 2$, $n(2^{1/n} - 1) \approx 0.83$. Since the test for RM is not satisfied, nothing can be said about schedulability. The example in Figure B.3 shows it is not schedulable under RM.

Another test was devised for the case where deadlines can be less than periods. This test is based on processor demand for each interval $[0, L]$. The principle behind this idea is that "the schedulability of a periodic task set is guaranteed if and only if the cumulative processor demand in any interval $[0, L]$ is less than the available time; that is, the interval length L"[5]. The new test is

$$\forall L \geq 0 \qquad L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

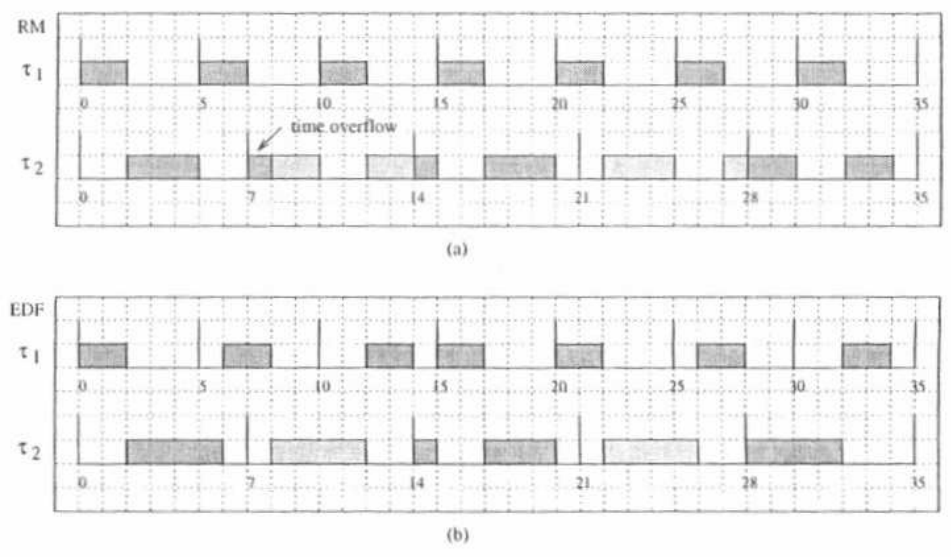


Figure B.3: An example of an RM (a) and an EDF (b) schedule of the same set of periodic tasks.

Appendix C

Class and sequence diagrams

This section contains the class diagrams and sequence diagrams of the implementation of Section 5.2 and the control flow of Section 5.3.

C.1 Class Diagrams

Figure C.1 shows the full class diagram of the current RGSpace implementation. The syntax used is that of Rational Rose, which means interfaces are shown as circles and abstract classes have italic names. To keep the diagram clear, types such as `Request` or standard Java types are not displayed. Also, Java classes such as `java.lang.Runnable`, which is implemented by `RGSpaceLogic` and `ErrorHandler`, are omitted. `Scheduler` (the scheduler super class) and `RealtimeThread` from `javax.realtime` are included however.

It also introduces a new class: `Convert`. Because only bytes can be sent through both the network and UNIX sockets, this class is responsible for converting objects into byte arrays and back again (marshalling and demarshalling). This utility class therefore has two methods, both static: `toBytes` and `toObject`.

To keep the image small, operators and fields are not displayed either. They are included in more detailed class diagrams in Figures C.2 until C.12. These show the public fields and methods of the classes used within RGSpace. `javax.realtime` classes are omitted.

Some additional notes:

- The methods of `RealtimeNetwork` (see Figure C.4) are taken from the network protocol, TRIP (see Section 5.2.3 and [11]). The `configure` method wraps setting and reading both the TRIP configuration file and profiling file.

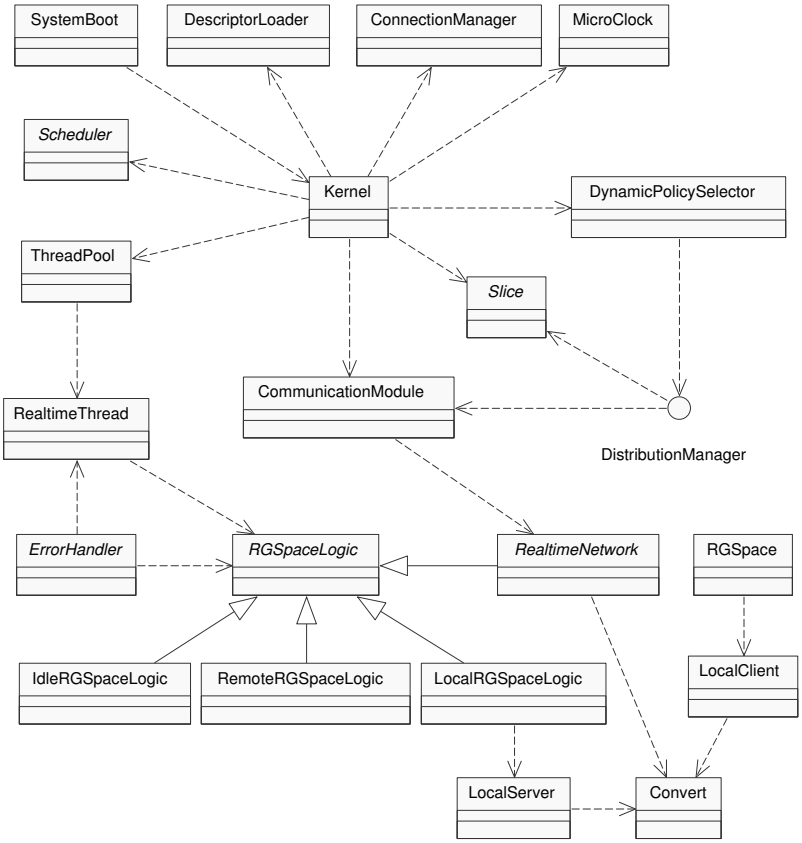


Figure C.1: RGSpace class diagram.

- Class `HighResolutionTime`, as used as a parameter in the `addThread` and `addNetworkThread` methods from class `ThreadPool` (see Figure C.5), are taken from `javax.realtime`. It is used to express time with nanosecond accuracy, and can be both absolute or relative.

- The fields of class `ErrorHandler` (see Figure C.6) are actually protected, not public. They are shown however, because they may be needed in sub classes.

- The `getSize` method of class `Slice` (see Figure C.10) returns the maximum number of tuples that can be stored inside the slice (see Section 5.5). This class is also abstract to make it easier to replace it by another implementation.

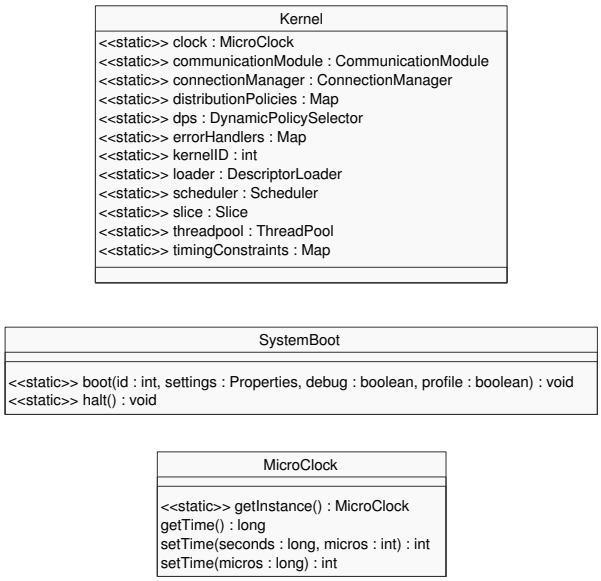


Figure C.2: The `Kernel`, `SystemBoot` and `MicroClock` Classes.

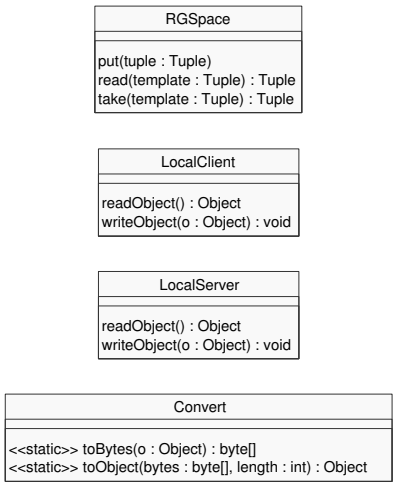


Figure C.3: Classes of the `API` module.

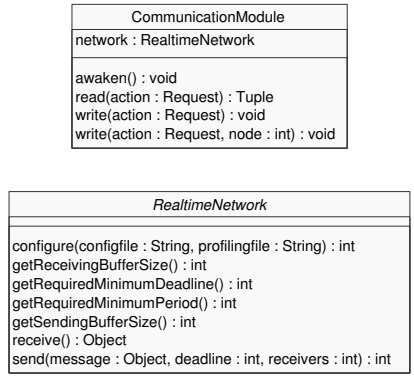


Figure C.4: Classes of the `Communication Module` module.

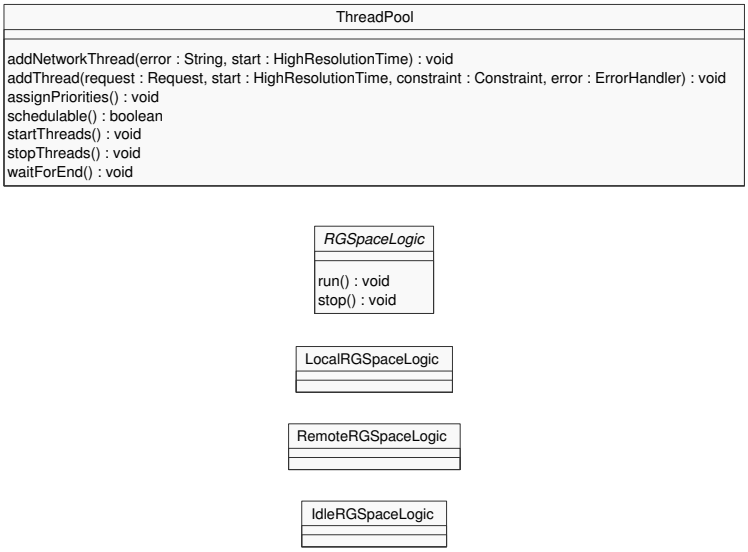


Figure C.5: Classes of the `Thread Pool` module.

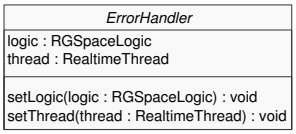


Figure C.6: The `ErrorHandler` class.

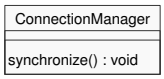


Figure C.7: Classes of the `Connection Manager` module.

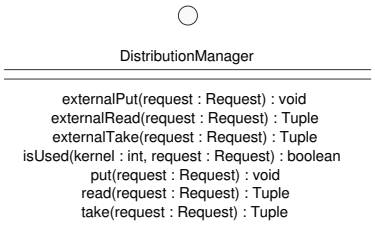


Figure C.8: The `DistributionManager` interface.

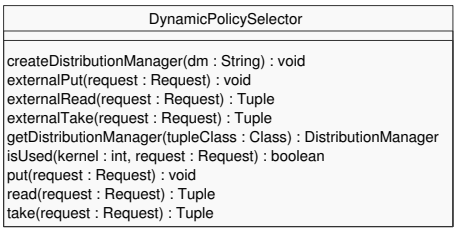


Figure C.9: Classes of the `Dynamic Policy Selector` module.

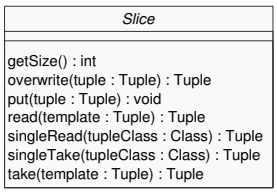


Figure C.10: Classes of the `Slice` module.

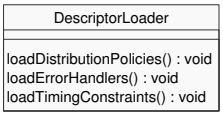


Figure C.11: Classes of the `Descriptor Loader` module.

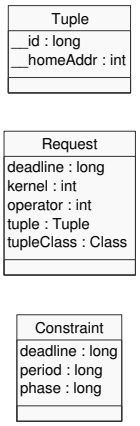


Figure C.12: Types inside RGSpace.

C.2 Sequence Diagrams

Figures C.13 and C.14 show the sequence diagrams for the example in Section 5.3, for both the local kernel and the remote kernel the tuples are pushed to.

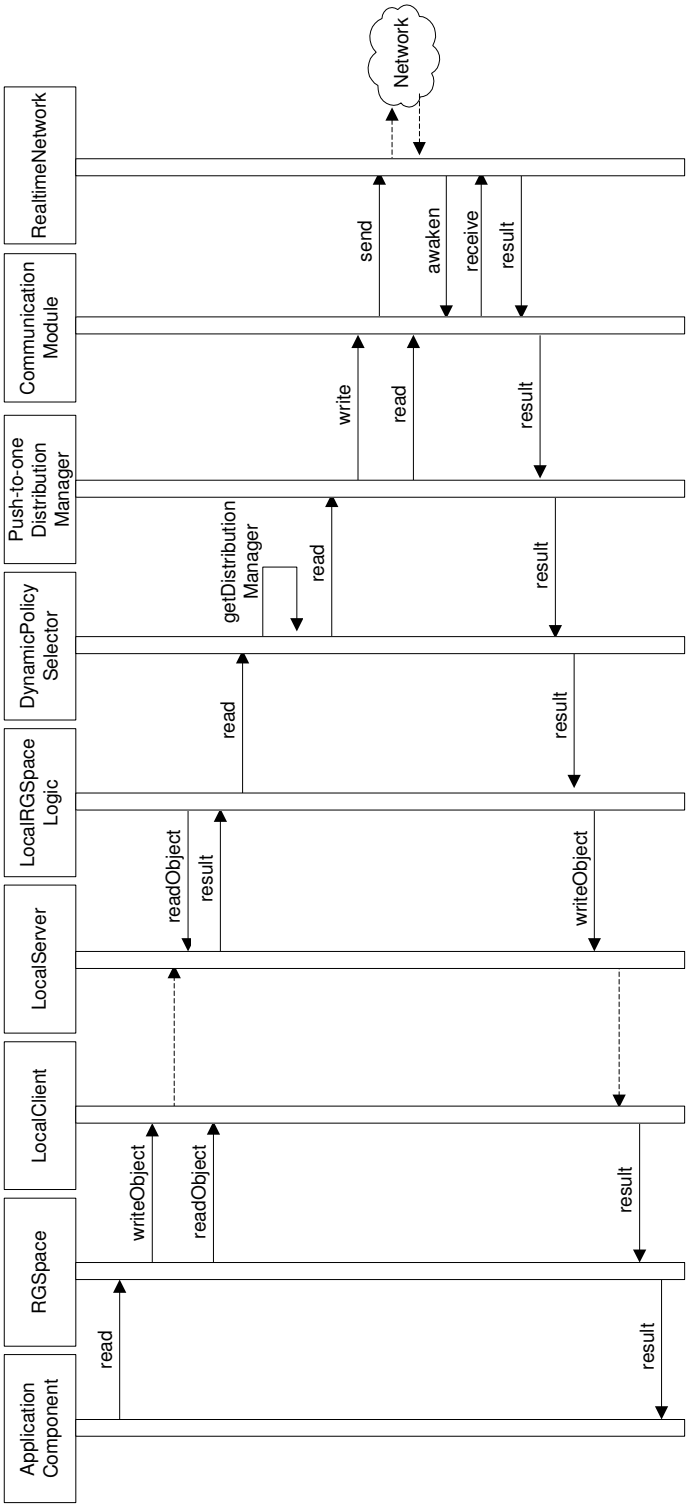


Figure C.13: Handling a read request from an application component.

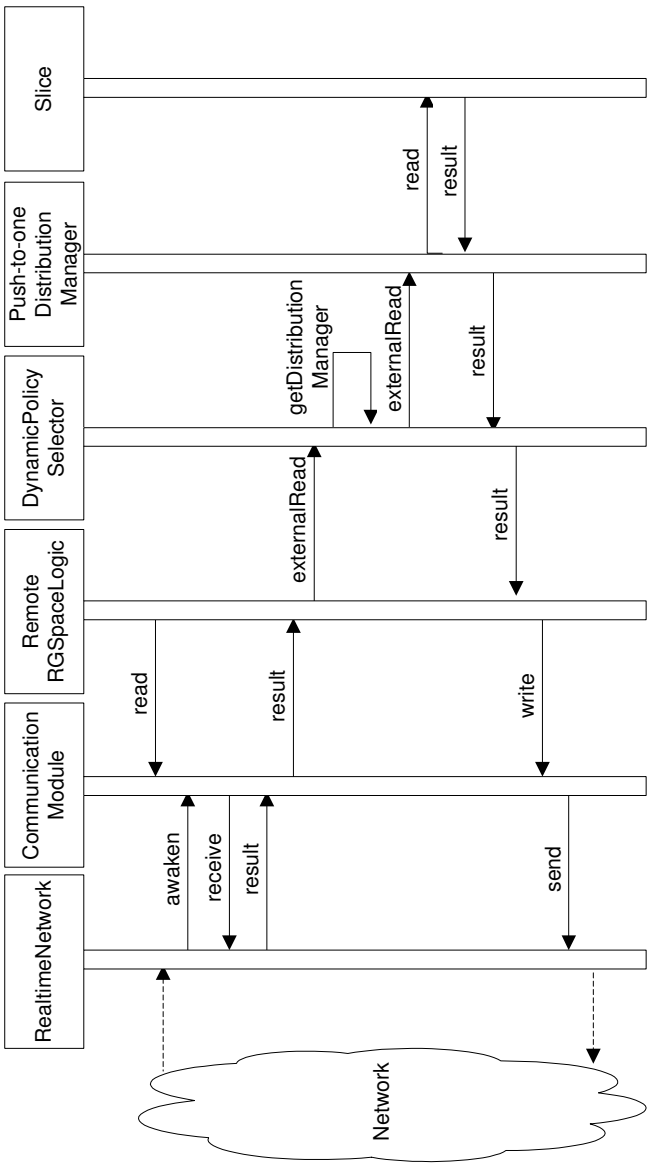


Figure C.14: Handling a read request from an application component on another kernel.

Appendix D

Technical details

This section contains a listing of hardware and software used in the tests. It also contains a complete installation guide.

D.1 Hardware

The test machines had the following configuration:

- An Intel Pentium 2 processor, with speeds of 535MHz
- 256MB of RAM
- One 10GB Western Digital hard disk
- A 10/100MBit full duplex network card

They were connected with a Cisco Systems Catalyst 3500 100MBit switch.

D.2 Software

The current RGSpace implementation is written in Java, using the `javax.realtime` package for extra real-time classes. This package, as specified in the Real-time Specification for Java (RTSJ), provides real-time threads, which can be created with the necessary priorities and real-time constraints. These threads are the threads used in RGSpace. They provide their own interaction with the scheduler classes of the `javax.realtime` package. These scheduler classes should ensure the real-time constraints of threads are not violated. Therefore, application developers only have to create threads with the necessary priorities, periods, costs and deadlines. The system should take care of the rest.

Instead of having a real-time Java Virtual Machine (JVM) that supports this `javax.realtime` package, we have chosen to use `jRate` (see `http://`

tao.doc.wustl.edu/~corsaro/jRate/index.html). This is an extension to GNU's GCC compiler (see http://gcc.gnu.org/) which also implements the javax.realtime package from the RTSJ. As a result, instead of being compiled to Java byte code, RGSpace is compiled to native machine code. This has two major advantages:

- Native machine code will run faster than interpreted Java byte code. Although we have stated that predictability is more important than performance, we feel the difference in performance is too great to ignore.
- Instead of using the Java Native Interface (JNI) to write C/C++ code to interact with the operating system, we can use the Compiled Native Interface (CNI). Whereas JNI leads to libraries being loaded by the JVM, with CNI C/C++ code and Java code are both compiled to native machine code and then linked together, to make one executable/library. This eliminates the relatively long load time.

Using jRate and CNI does lead to less portable code, but since we have chosen one operating system to work with this is not that relevant.

This operating system is RedHat Linux 7.3 with the TimeSys Linux 3.1 kernel (see http://www.timesys.com/). This TimeSys kernel is an enhanced Linux kernel, with the following features:

- The TimeSys Linux kernel is fully preemptible. A scheduling decision is made whenever an interrupt occurs, regardless of whether the processor was executing within the kernel or in user space at the time of the interrupt.
- Interrupt Handlers are run as real-time threads in TimeSys Linux. This provides a prioritized, preemptible interrupt hierarchy.
- SoftIRQ processing, which executes interrupt bottom halves and various other kernel functions, is performed by a real-time thread.
- The TimeSys Linux scheduler makes scheduling decisions in constant time.
- Most of the critical sections in the TimeSys kernel are protected by mutexes.
- TimeSys Linux improves the accuracy of process time accounting.

jRate's implementation of the javax.realtime.PriorityScheduler interacts with the TimeSys Linux scheduler directly, so no extra effort is needed to use these features. We have chosen this operating system because it was

also used in jRate tests. We would have liked to have tested on a variety of (semi) real-time operating systems such as VxWorks, QNX, KURT or RTAI, but we lacked both time and resources to do so. In fact, we could only use the GPL version of TimeSys Linux. TimeSys kernel modules such as TimeSys Linux/Real-Time, TimeSys Linux/CPU and TimeSys Linux/Net would have provided among others the removal of unbounded priority inversion[1], CPU and network reservations.

Unfortunately, there was a RedHat version problem: whereas the TimeSys kernel did not work on RedHat 8 or higher, jRate needed at least RedHat 8 to compile. This problem was solved by creating a hybrid RedHat 7.3/9 installation. Tables D.1 until D.4 list all packages installed after a minimal installation of RedHat 7.3. Section D.3 gives more complete installation instructions.

D.3 Installation

This section gives the installation guide for the test machines and RGSpace. All steps should be performed as user `root`.

The first step is to install RedHat 7.3. Follow the RedHat installation guide, with the following notes:

- Select the Custom installation type.
- Use Disk Druid for partitioning. Create partitions as needed, but make sure to keep `/usr` large enough; 4-5GB should be enough. Make sure to use EXT2 as the file system, since the TimeSys kernel (see Section D.2) cannot load EXT3 file systems.
- Use LILO as the boot loader, installed in the MBR.
- Do not use bootp/dhcp but give the machines a fixed IP address. All machines must be in the same range.
- Choose not to use a firewall.
- When it is time to select the package groups, deselect all groups for a minimal installation. This will be the last step.

Next, the packages from Tables D.1 until D.4 should be installed; once collected in one place (from CD or FTP) this can be achieved by the following command (the h and v can be omitted, but are included so progress can be monitored):

[1] We speak of priority inversion if a high priority thread is blocked on resources acquired by a lower priority thread.

```
# rpm -Uhv *.rpm
```

Next, issue the **ntsysv** command and uncheck everything except for `keytable`, `network` and `random`.

The next step is install jRate. First, create directory `/usr/jRateSuite` and go there:

```
# mkdir /usr/jRateSuite && cd /usr/jRateSuite
```

Next, unpack the jRate source:

```
# tar xzf $INSTALLDIR/jRate.0.3.4-3.3.2.tar.gz
```

where $INSTALLDIR is the directory where the jRate source is located. Afterwards, open the `/usr/jRateSuite/jRate/script/jRate-env.sh` file with a text editor (vi or emacs) and fill in the JRATE_SUITE_HOME variable:

```
export JRATE_SUITE_HOME=/usr/jRateSuite
```

Go back to the `/usr/jRateSuite` directory and create directories `jRate-gcc`, `GNU` and `GNU/jRateGCC`:

```
# mkdir jRate-gcc
# mkdir GNU && mkdir GNU/jRateGCC
```

Now unpack the GCC source and rename the unpacked directory to gcc:

```
# cd GNU/jRateGCC
# tar xzf $INSTALLDIR/gcc-3.3.2.tar.gz
# mv gcc-3.3.2 gcc
```

where $INSTALLDIR is the directory where the GCC source is located.

To compile jRate a set of environment variables is needed. Those are defined in the `/usr/jRateSuite/jRate/script/jRate-env.sh` file. However, because more environment variables are needed, skip compiling jRate for now.

The following step is to prepare both RGSpace and the TimeSys kernel for installation:

```
# mkdir /usr/rgspace && cd /usr/rgspace
# tar xzf $INSTALLDIR/rgspace-2.1.tar.gz
# mkdir /usr/timesys && cd /usr/timesys
# tar xjf $INSTALLDIR/tslinux-3-1-x86-bsp214c.tar.bz2
```

where $INSTALLDIR is the directory where respectively the RGSpace and TimeSys source is located.

Now all sources are installed, it is time to set all necessary environment variables. Edit the `/etc/bashrc` file and append the next line:

```
source /usr/jRateSuite/jRate/script/jRate-env.sh
```

Next, edit the `/etc/profile` file and append the following:

```
export JAVA_HOME="/usr/java/j2sdk1.4.2_01"
export RGSPACE_HOME="/usr/rgspace/rgspace-2.1"
export PATH="$PATH:$JAVA_HOME/bin:$RGSPACE_HOME"
export TIMESYS_HOME="/usr/timesys/tslinux-3.1"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$RGSPACE_HOME/lib"
```

Load these variables by reloading the `/etc/bashrc` and `/etc/profile` files:

```
# source /etc/bashrc
# source /etc/profile
```

Now it is time to compile both jRate and RGSpace:

```
# cd $JRATE_HOME && make jRate-gcc
# cd $RGSPACE_HOME && make profiler
```

Finally, it is time to install the TimeSys kernel. Go to the TimeSys directory and execute the `install` script:

```
# cd $TIMESYS_HOME && ./install
```

After a reboot RGSpace is ready to run. However, the machine will boot into the standard RedHat 7.3 kernel by default. Although this is not a bad thing per se, it is more convenient to have the machine to boot into the TimeSys kernel by default. To achieve this, change the `default` in the `/etc/lilo` file to `timesys` and then execute `lilo`:

```
default=timesys
# /sbin/lilo
```

When rebooting, the system will now automatically boot into the TimeSys kernel.

D.4 Running RGSpace

After RGSpace is installed, the kernel can be called as follows:

```
# cd $RGSPACE_HOME
# ./rgspace-kernel <ID> <SETTINGS> [OPTIONS]
```

anacron	apmd	ash	authconfig
autofs	basesystem	bash	bdflush
bind-utils	chkconfig	cipe	console-tools
cracklib	cracklib-dicts	crontabs	cyrus-sasl
cyrus-sasl-md5	cyrus-sasl-plain	db2	db3
dev	dhcpcd	ed	eject
filesystem	fileutils	findutils	finger
gnupg	grub	hdparm	hesiod
hotplug	hwdata	indexhtml	info
initscripts	ipchains	iproute	iptables
iputils	kbdconfig	kernel	ksymoops
kudzu	lilo	logrotate	logwatch
lokkit	losetup	mailx	mingetty
mkbootdisk	mkinitrd	mktemp	modutils
mount	mouseconfig	netconfig	newt
nfs-utils	nss_ldap	ntsysv	openldap
openldap-clients	pam_krb5	passwd	pciutils
perl-CGI	perl-CPAN	perl-DB_File	pidentd
pine	portmap	procmail	procps
psmisc	pwdb	python-clap	python-popt
python-xmlrpc	quota	radvd	raidtools
redhat-logos	redhat-release	reiserfs-utils	rootfiles
setup	setuptool	sh-utils	slang
slocate	syslinux	SysVinit	tcsh
textutils	timeconfig	whois	ypbind
yp-tools			

Table D.1: Packages of RedHat 7.3

at	atk	atk-devel	attr
autoconf213	autoconf	automake14	automake15
automake	binutils	bison	byacc
bzip2	bzip2-devel	bzip2-libs	cdecl
chkfontpath	cpio	cpp	cproto
ctags	cups-libs	curl	curl-devel
cvs	db4	db4-devel	db4-java
db4-utils	dialog	diffstat	diffutils
dmalloc	dos2unix	dosfstools	doxygen
dtach	e2fsprogs	e2fsprogs-devel	elfutils
elfutils-devel	elfutils-libelf	elinks	emacs
ethtool	expat	expat-devel	expect
expect-devel	fam	fam-devel	file
flex	fontconfig	fontconfig-devel	freetype
freetype-devel	ftp	gawk	gcc
gcc-c++	gcc-g77	gcc-gnat	gcc-java
gcc-objc	gd	gdb	gdbm
gdbm-devel	gd-devel	gdk-pixbuf	gd-progs
gettext	glib	glib2	glib2-devel
glibc	glibc-common	glibc-devel	glibc-kernheaders
glibc-profile	glibc-utils	glib-devel	gmp
gmp-devel	gpm	gpm-devel	grep
groff	gtk+	gtk2	gzip
indent	itcl	krb5-devel	krb5-libs
krbafs	krbafs-devel	krbafs-utils	less
lftp	lha	libacl	libacl-devel
libaio	libaio-devel	libart_lgpl	libart_lgpl-devel
libattr	libattr-devel	libcap	libcap-devel
libf2c	libgcc	libgcj	libgcj-devel
libgnat	libIDL	libIDL-devel	libjpeg
libjpeg-devel	libobjc	libogg	libogg-devel
libpcap	libpng10	libpng10-devel	libpng
libpng-devel	libstdc++	libstdc++-devel	libtermcap
libtermcap-devel	libtiff	libtiff-devel	libtool
libtool-libs13	libtool-libs	libungif	libungif-devel
libungif-progs	libunicode	libunicode-devel	libusb
libusb-devel	libuser	libuser-devel	libvorbis
libvorbis-devel	libwvstreams	libwvstreams-devel	libxml
libxml2	libxml2-devel	libxml2-python	libxml-devel
libxslt	libxslt-devel	libxslt-python	linc
linc-devel	lockdev	lockdev-devel	lslk
lsof	ltrace	lynx	m4

Table D.2: Packages of RedHat 9 (part 1)

mailcap	make	MAKEDEV	man
man-pages	mpage	mrtg	mtools
mtr	mt-st	nc	ncftp
ncurses4	ncurses	ncurses-c++-devel	ncurses-devel
net-snmp	net-snmp-devel	net-snmp-utils	net-tools
nmap	nscd	ntp	openssl096
openssl096b	openssl	openssl-devel	openssl-perl
ORBit	ORBit2	ORBit2-devel	ORBit-devel
pam	pam-devel	pango	pango-devel
parted	parted-devel	patch	patchutils
pcre	pcre-devel	perl	perl-Filter
perl-URI	pkgconfig	pmake	popt
pspell	pspell-devel	pstack	pychecker
pyOpenSSL	pyorbit	pyorbit-devel	python
python-devel	python-docs	python-optik	python-tools
pyxf86config	rcs	rdate	rdist
readline41	readline	readline-devel	rhnlib
rhpl	rmt	rpm	rpm-build
rpm-devel	rpm-python	rsh	rsync
samba-client	samba-common	screen	sed
sendmail	sendmail-cf	sendmail-devel	sendmail-doc
setserial	shadow-utils	sharutils	sox
sox-devel	specspo	splint	statserial
strace	stunnel	swig	symlinks
sysklogd	sysreport	sysstat	talk
tar	tcl	tcpdump	tcp_wrappers
telnet	termcap	texinfo	time
tix	tk	tkinter	tmake
tmpwatch	traceroute	ttmkfdir	unarj
unix2dos	unzip	up2date	usbutils
usermode	utempter	util-linux	vim-common
vim-enhanced	vim-minimal	vixie-cron	vlock
wget	which	words	Xaw3d
xdelta	xdelta-devel	XFree86-devel	XFree86-font-utils
XFree86-libs	XFree86-libs-data	XFree86-Mesa-libGL	XFree86-truetype-fonts
XFree86-xfs	xinetd	zip	zlib
zlib-devel			

Table D.3: Packages of RedHat 9 (part 2)

j2sdk-1_4_2_01-linux-i586.rpm	openssh-3.7.1p2-1.i386.rpm
openssh-clients-3.7.1p2-1.i386.rpm	openssh-server-3.7.1p2-1.i386.rpm

Table D.4: Other packages

Settings	Meaning
deadline_factor	The deadline for sending messages is currently the same for all messages, and is a factor of the minimum deadline for TRIP. This setting determines the ratio between the minimum deadline and the actual deadline. It should be at least 1.0.
dp_descriptor	The distribution policy descriptor filename.
tc_descriptor	The timing constraints descriptor filename.
eh_descriptor	The error handler descriptor filename.
slice_size	The maximum number of tuples in the slice (see Section 5.5).
networkclass	The full class name of the network protocol implementation. Currently only `rgspace.network.TRIP` is supported.
networkhandler	The full class name of the error handler used when the network thread misses a deadline.
networkconfig	The configuration file for the network protocol.
networkprofiling	The file to store the profiling information of the network protocol.
profiling	The file to store the profiling information of RGSpace.

Table D.5: Settings for RGSpace.

The ID parameter is the kernel identifier. The SETTINGS parameter is the name of the configuration file. Table D.5 lists the possible settings of this file. The options include `-d` to start the kernel in debugging mode. In this mode, extra information is printed to the screen, which can help discover what is going on. Option `-p` starts the kernel in profiling mode; the start and end times of all instances of the periodic tasks are printed to the screen. Finally, `--help` prints information to the screen on how to start the kernel.

Before RGSpace can be run, some profiling is needed first, so RGSpace will know the costs of operations. The `rgspace-profiler` and `trip-profiler` programs are used for this task. The latter will perform profiling necessary for TRIP (see Section 5.2.3 and [11]). It has one parameter, the RGSpace configuration file. It needs to be run on all machines that will be used for TRIP and RGSpace, with the TRIP profiler on the first machine (with kernel identifier 1) started after the TRIP profilers on all other machines are started.

The RGSpace profiler must profile four costs:

1. The time needed to for the CommunicationModule (see Section 5.2.3) to return after a non-blocking read.

2. The time needed to convert an object into byte code and back again.

3. The time needed to `read` and `take` tuples from the slice and to `put` tuples into the slice.

4. The time needed for each TRIP run.

The profiler can be called as follows:

```
# cd $RGSPACE_HOME
# ./rgspace-profiler <TYPE> <RUNS> <CONFIG>
```

The TYPE parameter is the number from the above enumeration. The profiling has been split because the maximum number of runs, specified by the RUNS parameter, is different for all four types of profiling. Usual number of runs are 100.000 for type 1, 450 for type 2[2], 2500 for type 3 and 100.000 for type 4. Type 4 has the same constraints as the TRIP profiler: it needs to be run on all machines, with the profiler on the first machine started after the profilers on all other machines.

After profiling has finished, the two profiling files (see Table D.5) must be copied from the first machine to all other machines. SSH can be used for this. Now the RGSpace system can be started, again the kernel on the first machine started after the kernels on all other machines.

[2]Because the test is creating many objects within a short interval, a high number leads to the profiler to run out of memory.

Bibliography

[1] B. Adelberg, B. Kao, H. Garcia-Molina, *Overview of the STanford Real-time Information Processor (STRIP).* SIGMOD Record, Vol. 25, No. 1, March 1996, 34-37

[2] S.F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, B. Eftring *DeeDS Towards a Distributed and Active Real-Time Database System.* ACM SIGMOD Record, Vol. 25, No. 1, March 1996, 38-40

[3] Maarten Boasson, Edwin de Jong, *Software Architecture for Large Embedded Systems.* `http://homepages.cwi.nl/~marcello/SAPapers/BJ97.html`; accessed July 8th, 2004

[4] Maarten Boasson, *Control Systems Software.* IEEE Transactions on Automatic Control, Vol. 38, No. 7, July 1993, 1094-1106

[5] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers. Fourth printing 2002. ISBN 0-7923-9994-3

[6] G. Cabri, L. Leonardi, F. Zambonelli, *Mobile-Agent Coordination Models for Internet Applications.* IEEE Computer, Vol. 33, No. 2, Feb. 2000, 82-89

[7] Nicholas Carriero, David Gelernter, *How to Write Parallel Programs: A Guide to the Perplexed.* ACM Computing Surveys, Vol. 21, No. 3, September 1989, 323-357

[8] Nicholas Carriero, David Gelernter, *Linda in Context.* Communications of the ACM, Vol. 32, No. 4, 1989, 444-458

[9] David Gelernter, *Generative Communication in Linda.* ACM Trans. Prog. Lang. Syst., Vol. 7, No. 1, 1985, 80-112

[10] Y-K. Kim, M.R. Lehr, D.W. George, S.H. Song, *A Database Server for Distributed Real-Time Systems: Issues and Expriences.* Proceedings of the Second IEEE Workshop on Parallel and Distributed Real-Time Systems, IEEE Computer Society, April 1994, 66-75

[11] Iñaki Lazarobaster Badiola, Igor Radovanović, Giovanni Russello, Michel Chaudron, *Design and Implementation of a Real-time Protocol over Ethernet.* Technical Report CS-Report 04/32, Department of Computer Science, Eindhoven University of Technology; November 2004

[12] G. Russello, M.R.V. Chaudron, M. v. Steen, *Customizable Data Distribution for Shared Data Spaces.* Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'03), Las Vegas, June 2003

[13] G. Russello, M.R.V. Chaudron, M. v. Steen, *GSpace: Tailorable Data Distribution in Shared Data Space Systems.* Technical Report CS-Report 04/06, Department of Computer Science, Eindhoven University of Technology; 2004-01-30

[14] Douglas C. Schmidt, David L. Levine & Sumedh Mungee, *The Design of the TAO Real-Time Object Request Broker.* Computer Communications, Vol. 21, No. 4, 1998, 294-324.

[15] J.A. Stankovic, S.H. Son, *Architecture and Object Model for Distributed Object-Oriented Real-Time Databases.* Proceedings of The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing Washington D.C., USA, April 1998

[16] Sun Microsystems, Inc., *JavaSpaces^{TM}Service Specification*, June 2003. `http://java.sun.com/products/jini/2.0/doc/specs/html/js-title.html`; accessed June 8th, 2004