*The ObjectWeb Consortium*


# Dream Framework Core

**Authors**:

| | |
|---|---|
| M. Leclercq | (INRIA) |
| V. Quéma | (INPG) |
| J.-B. Stefani | (INRIA) |


| | |
|---|---|
| Released | September 20, 2005 |
| Status | Draft |
| Version | 0.3 |

### General Information

Please send comments on this document to `dream@objectweb.org`. Authors would be glad to hear from people using or extending Dream.



### Trademarks

All product names mentioned herein are trademarks of their respective owners.

# Contents

# List of Figures

# 1 Introduction

## 1.1 Motivations

Several communication middleware (CMs) have been built in the past ten years [1, 2, 3, 4, 5, 6, 7]. The research work has primarily focused on providing various communication paradigms (RPC, message passing, event-reaction, publish-subscribe, ...) and on the support of various non functional properties like message ordering, reliability, security, scalability, etc. Less emphasis has been placed on configurability. From the functional point of view, existing CMs implement a fixed programming interface (API) that provides a fixed subset of communication models.

From the non-functional point of view, existing CMs often provide the same non-functional properties for all message exchanges. This reduces their performance and makes them difficult or impossible to use with devices having limited computational resources. As these non-functional properties have not been developed as independent (removable) modules, removing them often requires the code to be totally re-engineered.

To overcome these limitations, it is necessary to build modular and composable architectures. Work on configurable systems has lead, in particular, to the development of component-based and reflective middleware [8]. The idea is to build a middleware as an assembly of interacting components, which can be statically or dynamically configured to meet different design requirements or environment constraints. While in principle applicable to different forms of middleware, existing component-based middleware have mostly been used to construct classical middleware with synchronous interactions, and with a few exceptions [9, 10], have not dealt systematically with resource configurability (i.e. the ability to control the use of resources within the middleware). Modular architectures have also been proposed to build routers [11] or communication sub-systems [12]. The main limitation of these systems lies in their restricted component model, which can mainly be used for static configuration, does not support hierarchical composition, and which does not provide any control capabilitiy, thus making it hard to administer and configure systems during execution.

## 1.2 The Dream framework

DREAM is a software framework dedicated to the construction of CMs. It provides a component library and a set of tools to build middleware implementing various communication paradigms: message passing, event-reaction, publish-subscribe, etc. Figure 1 gives a general picture of the framework. DREAM builds upon JULIA, a Java implementation of the FRACTAL component model [13]. FRACTAL provides support for hierarchical and dynamic composition. Hierarchical composition supports the construction of systems from the assembly of structured (hierarchical) sets of components. Dynamic composition provides the basis for dynamic reconfiguration, an essential feature for long-running systems.

DREAM defines 3 modules:

- *Dream Core* provides FRACTAL components with abstractions and tools to handle resources commonly found in CMs: messages and activities.

- *Dream ADL* extends JULIA's architecture description language with features for distributed deployment and dynamical reconfiguration of CMs.
- *Dream Library* provides a library of components implementing functions commonly found in CMs.

Several personalities have been built on top of these 3 modules: AAA[1] is a message-oriented middleware (MOM) allowing the deployment of *agents* behaving according to an event → reaction pattern; Tribe is a framework for building various group communication middleware; SEDA[2] is an implementation of the Staged Event-Driven Architecture; LeWYS is a framework for building distributed monitoring systems ...

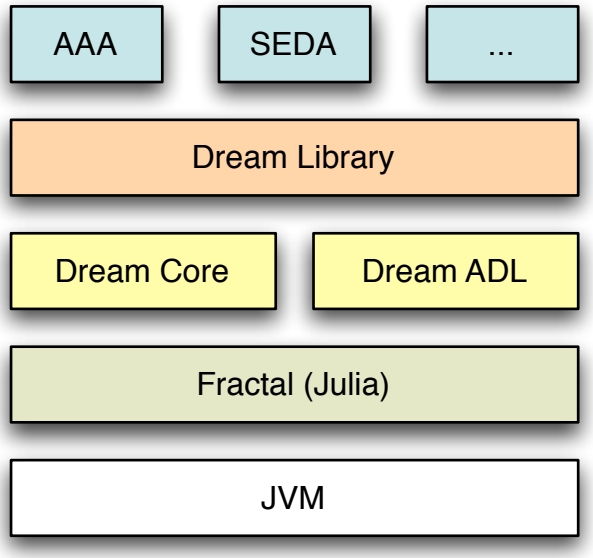


Figure 1: DREAM framework overview

## 1.3 Structure of this document

This document presents the core of the DREAM framework. TBD.

## 1.4 Warning

A good knowledge of the Fractal component model and its Java reference implementation – JULIA – is necessary to understand this document. Documentation about FRACTAL and JULIA is freely available at the following URL: `http://fractal.objectweb.org`.

[1] AAA is freely inspired by [14]
[2] SEDA is freely inspired by Sandstorm [15]

# 2 Component model

This section describes the component model that is used to build DREAM components. This model, called FRACTAL [16, 17], is developed within the OBJECTWEB consortium. It is a general component model that has been implemented in multiple languages. We focus on its Java reference implementation, JULIA, that is used to build DREAM components.

## 2.1 The Fractal component model

Unlike other Java-based component models such as Jiazzi [18] or ArchJava [19], FRACTAL is not a language extension but consists in a runtime library which supports the creation and manipulation of components and architectures.

FRACTAL distinguishes between two kinds of components: **primitive components**, which are essentially standard Java classes conforming to simple coding conventions, and **composite components**, which provide a means to deal with a group of components as a whole, while potentially hiding some of the features of the encapsulated subcomponents. An original feature of the Fractal component model is that a given component can be included in several other components. Such a component is said to be **shared** between these components. Shared components are useful, paradoxically, to preserve component encapsulation: there is no need to expose interfaces in higher-level components to allow access to a shared component by a lower-level one. Shared components are useful in particular to faithfully model access to low-level system resources.

A component is made of two parts: a **controller part**, which exposes the component interfaces and comprises controller and interceptor objects, and a **content part**, which can be either a standard Java class in the case of a primitive component, or other components (called subcomponents), in the case of a composite component.

Interfaces in Fractal correspond to access points to a component. They can be of two kinds: **server interfaces**, which correspond to access points accepting incoming method calls, and **client interfaces**, which correspond to access points supporting outgoing method calls. The signatures of both kinds of interface can be described by a standard Java interface declaration, with an additional role indication (server or client). Moreover the model allows particular client interfaces, called **collection interfaces**, to be bound to several server interfaces.

Communication between Fractal components is only possible if their interfaces are bound. Fractal supports both primitive bindings and composite bindings. A **primitive binding** is a binding between one client interface and one server interface in the same address space, which means that method calls emitted by the client interface should be accepted by the specified server interface. A primitive binding is called that way for it can be readily implemented by direct Java references. A **composite binding** is a communication path between an arbitrary number of component interfaces. These bindings are built out of a set of primitive bindings and binding components (stubs, skeletons, adapters, etc). The Fractal model thus provides two mechanisms to define the architecture of an application: bindings between component interfaces, and encapsulation of a group of components in a composite.

Fractal does not impose any limit on the levels of composition, and does not constrain the nature of component controllers. The Fractal library contains several kinds of controllers,

which can be combined to yield components with different reflective features, and which can be extended or specialized to yield other forms of reflection and control over components. The following are examples of Fractal controllers.

- **Attribute controller:** an attribute is a configurable property of a component. A component can provide an `AttributeController` interface to expose getter and setter methods for its attributes.
- **Binding controller:** a component can provide the `BindingController` interface to allow binding and unbinding its client interfaces to other server interfaces by means of primitive bindings.
- **Content controller:** a component can provide the `ContentController` interface to list, add and remove subcomponents in its contents.
- **Lifecycle controller:** a component can provide the `LifecycleController` interface to allow explicit control over its main behavioral phases, in support for dynamic reconfiguration. Basic lifecycle methods supported by the interface include methods to start and stop the execution of the component.

## 2.2 Example

The following figure illustrates the different constructs in a typical Fractal component architecture. Thick grey boxes denote the controller part of a component, while the interior of these boxes correspond to the content part of a component. Arrows correspond to primitive bindings, and tau-like structures protruding from grey boxes are internal or external interfaces. Internal interfaces are only accessible from the content part of a component. Note that, by means of internal interfaces, a composite component can control the exposition of external interfaces of its subcomponents. External interfaces appearing on the top of a component represent controller interfaces such as `AttributeController` or `ContentController` interfaces. The two shaded boxes represent a shared component.

## 2.3 Julia: the Fractal's Java-based reference implementation

### 2.3.1 Main data structures

In JULIA, a FRACTAL component is represented by several Java objects, which can be separated into three groups (Figure 3):

- the objects that implement the content part of the component (not shown). These objects can be sub components (for composite components), or *plain Java objects* (for primitive components).
- the objects that implement the controller part of the component, in black and gray. These objects can be separated into two groups: the objects that implement the control interfaces, called *controller objects*, and optional *interceptor* objects that intercept incoming and/or outgoing method calls on functional interfaces. Since the control aspects

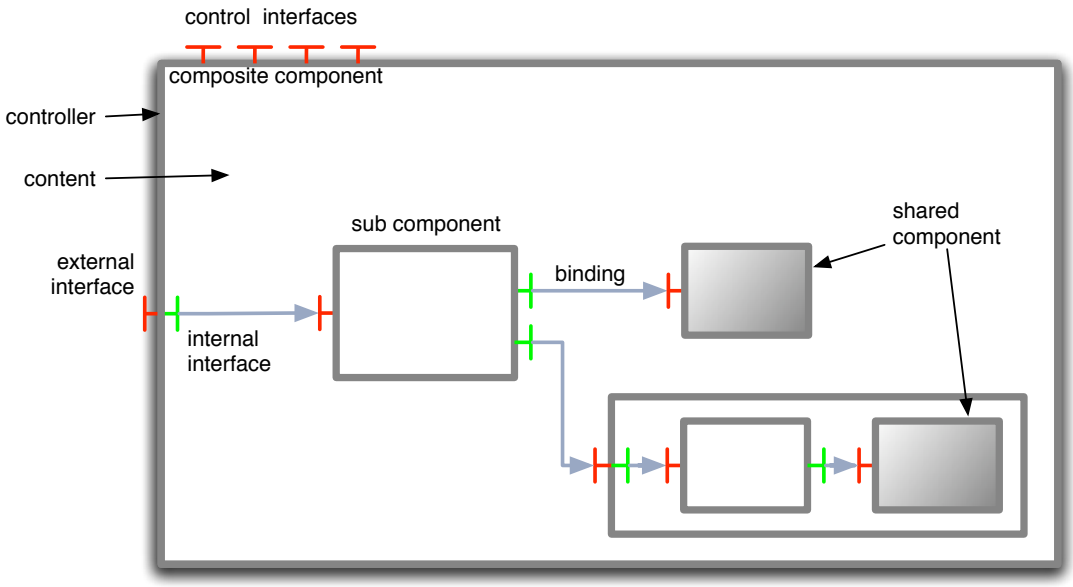


Figure 2: Architecture of a Fractal component

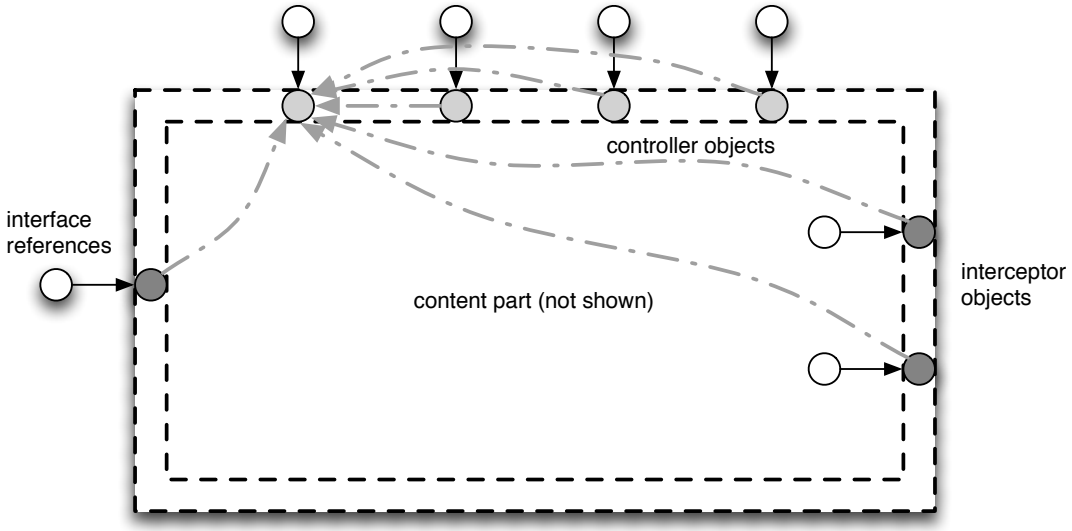


Figure 3: Implementation of a Fractal component

are generally not independent, controller and interceptor objects must generally have references to each other.

- the objects that reference the interfaces of the component, in white. Reference objects are the only objects seen by the content of a component. Moreover, they are not replaceable (but their internal state can be changed). These constraints allow a safe re-

configuration of components: indeed, the only possible references are references between controller objects, between interface reference and controller objects, from interceptors to user objects, and so on, which are all managed by Fractal. It is thus possible, when replacing a controller or an interceptor object, to update all the references to the old object.

These structures are automatically set up when the component is instantiated. This is done by a component factory that takes as parameters the type of the component to be created (i.e. its functional interfaces) and a description of its control part (i.e. its controller and interceptor objects).

### 2.3.2 Optimizations

JULIA provides a *continuum* from static to dynamic configuration, i.e., from unreconfigurable but very efficient configurations, to fully dynamically reconfigurable but less efficient configurations. These optimizations take two forms. The first one consists in a selective merging of the objects that make up a component. The second one creates shortcut bindings between components that bypass controller objects that do not intercept any method calls. A component can only be optimized if its controllers follow some code conventions (in particular, a component cannot have two controllers that implement the same interface).

# 3 Messages

This section describes the abstractions and tools provided in the DREAM framework for handling messages.

## 3.1 Structure of messages

Messages are Java objects that encapsulate **chunks**. Each chunk is a Java object that implements setters and getters. For instance, messages that need to be causally ordered have a chunk, called `CausalityChunk`, that encapsulates a stamp (i.e. a matrix clock), and that defines methods to set and get the stamp. Moreover, each chunk must extends the `AbstractChunk` class. This generic abstract class (figure 4) defines abstract methods to create a new instance of chunk of the same class, and to transfer the state of this chunk into another one.

```
package org.objectweb.dream.message;

public abstract class AbstractChunk<T extends AbstractChunk<T>> {
  ...
  protected abstract T newChunk();
  protected abstract void transferStateTo (T newInstance);
  ...
}
```

Figure 4: The `AbstractChunk` class

A message may also encapsulate other messages, called **submessages**. Note that chunks and submessages are handled differently: messages can be considered as naming contexts for their chunks, they associate chunks with names; this is not true for submessages because they are not associated with names[3].

DREAM components are not allowed to manipulate messages directly, any message operations (get, add, remove chunk and submessages) are made by the *Message Manager* component. This allows the *Message Manager* to implements complex mechanism such as persistency of message content.

## 3.2 Message managers

Message managers are components responsible for the life cycle and the manipulation of messages. They have a server interface, called `MessageManager` (figure 5, that defines methods to create, delete and duplicate chunks and messages; and methods to add, get, remove chunks and submessages from a message.

[3] This choice was driven by the fact that it is necessary to be able to build aggregated messages (i.e. messages that contains a set of submessages) very efficiently. This would not be possible if messages were associated with names since it would require generating different names for each encapsulated message. Note that it is possible to associate names to messages by simply adding a `NameChunk` to each message.

Chunks are created by *Chunk Factory*; these factories are managed internally by the *Message Manager* component. Components implementations only handle `ChunkFactoryReference` which are references (i.e. names) to *Chunk Factory* instances. The `getChunkFactory` allows to retrieve a `ChunkFactoryReference` for the given chunk class; which can be used to create a new chunk instance using the `createChunk` method. Each chunk instance keeps a reference to the `ChunkFactoryReference` that creates it, so when it is deleted the chunk may be recycled in an object pool managed by the chunk factory.

There exist two kinds of message duplication: the *duplication by value* returns a clone of the message; the *duplication by reference* returns the message itself. It is useful when the message manager pools instances of messages: this allows knowing the number of components that have a reference to a message.

```
package org.objectweb.dream.message;

public interface MessageManager {
  // message creation/deletion
  Message createMessage();
  void deleteMessage(Message message);

  // chunk creation/deletion
  <T extends AbstractChunk<T>> ChunkFactoryReference<T>
      getChunkFactory(Class<T> chunkType);
  <T extends AbstractChunk<T>> T
      createChunk(ChunkFactoryReference<T> chunkFactory);
  <T extends AbstractChunk<T>> T cloneChunk(T chunk);
  void deleteChunk(AbstractChunk<?> chunk);

  // chunk manipulation
  addChunk(Message message, String chunkName, AbstractChunk chunk);
  AbstractChunk getChunk(Message message, String chunkName);
  AbstractChunk removeChunk(Message message, String chunkName);
  Iterator<String> getChunkNameIterator(Message message);

  // submessage manipulation
  void addSubMessage(Message message, Message subMessage);
  Iterator<Message> getSubMessageIterator(Message message);
  void removeSubMessage(Message message, Message subMessage);

  // general methods
  boolean isEmpty(Message message);
  Message duplicateMessage(Message message, boolean clone);
}
```

Figure 5: The `MessageManager` interface

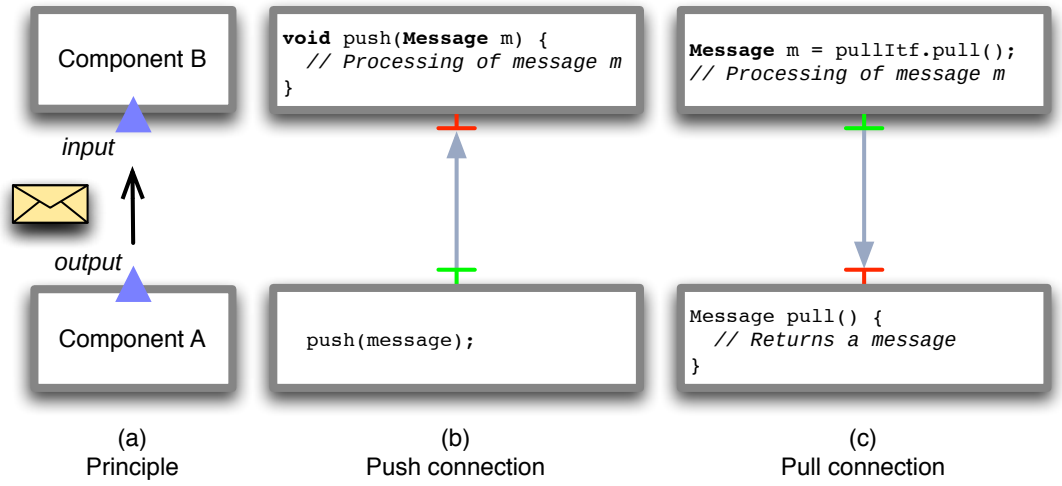


Figure 6: Connection between input and output interfaces

## 3.3 Input/Output interfaces

In order to allow components to exchange messages, the DREAM framework defines Fractal interfaces called **input** and **output**. Messages are always sent from outputs to inputs (Figure 6 (a)). Output and input interfaces come in pairs corresponding to two kinds of connections: *push* and *pull*.

- The **push connection** corresponds to message exchanges initiated by the output. The latter is a client interface bound to the input server interface (`Push`) (Figure 6 (b)). The `Push` interface is depicted in figure 7. It defines a method `push` that allows pushing a message to the input.

```
package org.objectweb.dream;

public interface Push {
    void push(Message message) throws PushException;
}
```

Figure 7: The `Push` interface

- The **pull connection** corresponds to message exchanges initiated by the input interface. The latter is a client interface bound to the output server interface (`Pull`) (figure 6 (c)). The `Pull` interface (figure 8) defines a method `pull` that returns a message.

```
package org.objectweb.dream;

public interface Pull {
   Message pull() throws PullException;
}
```

Figure 8: The `Pull` interface

# 4 Activities

This section describes the abstractions and tools provided in the DREAM framework for handling components' activities. We start this section by an overview of the activity management framework. Then we describe the components it involves. Finally, we describe the integration of activities with components' life-cycle management.

## 4.1 Overview

A DREAM component can either be *passive* or *active*. An active component defines **tasks** to be executed; a passive component doesn't; i.e. calls to other component interfaces can only be made in the tasks of a calling component. For a task to be executed, it must be registered to one of the dedicated shared components, called **activity managers**, that encapsulate tasks and **schedulers**. Schedulers are in charge of mapping higher-level tasks onto lower-level tasks. The number of scheduler levels is not limited. Tasks at the highest level are tasks registered by components. At the lowest-level, tasks wrap Java threads. These concepts are depicted in figure 9. Components A and B have registered three tasks that are scheduled by a FIFO scheduler, which maps them onto two lower-level tasks wrapping threads.

## 4.2 Scheduler, task and activity manager components

### 4.2.1 Schedulers

Scheduler are components with a `Scheduler` server interface, and a `Task` client collection interface. The role of a scheduler is to map higher-level tasks (to which its `Task` client interface is bound) onto lower-level tasks (that are bound to its `Scheduler` server interface).

The `Scheduler` interface (figure 10) defines a method `schedule` that is called by lower-level tasks to schedule the execution of a (set of) higher-level task(s). This method takes two parameters: `executionQuanta` specifies the amount of execution time the scheduler can use (a negative integer allows the schedule looping until it has no task to execute); `hints` specifies additional scheduling parameters. It returns an object representing the scheduling result.

The `Task` interface (figure 11) defines a method `execute` that is called by the scheduler to execute higher-level tasks. This method takes a parameter `hints` that allows specifying execution parameters, and returns an object representing the execution's result.

The DREAM framework currently provides the following schedulers:

- the **forwarder scheduler** is the simplest scheduler: it is bound to only one higher-level task that it executes each time its schedule method is called.
- the **FIFO scheduler**: not yet implemented.
- the **round-robin scheduler**: not yet implemented.

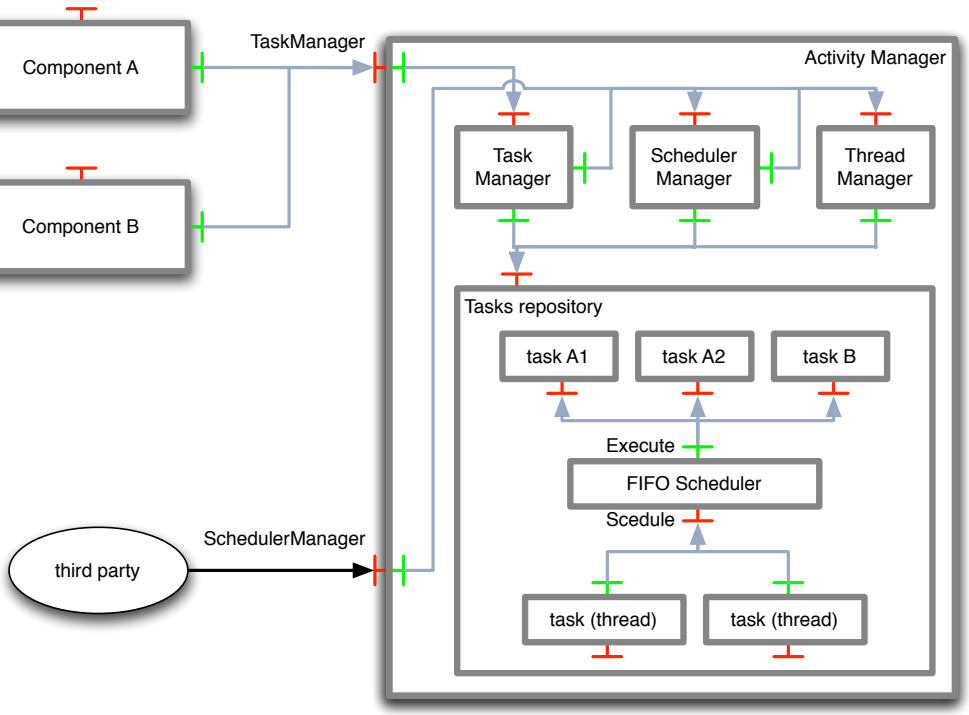


Figure 9: Activity management

```
package org.objectweb.dream.control.activity.scheduler;

public interface Scheduler {
  Object schedule(int executionQuanta, Object hints)
      throws InterruptedException, StoppedSchedulerException;
}
```

Figure 10: The `Scheduler` interface

### 4.2.2 Tasks

Tasks are components that implement the `Task` interface (figure 11). We distinguish:

- *highest-level tasks* are implemented by components. Their `execute` method contains functional code that follows some conventions: it must return 0 for the method to be executed again; it must return a positive integer to force the next execution of the method to be, at least, delayed by the returned number of milliseconds; it must return

```
package org.objectweb.dream.control.activity.task;

public interface Task {
  Object execute(Object hints) throws InterruptedException;

  // call back methods
  void interrupted();
  void registered(Object controlItf);
  void unregistered();
}
```

Figure 11: The `Task` interface

a negative integer to specify that it must not be executed any longer. Note that there is *a priori* no limitation on the number of threads simultaneously executing such tasks. It is the responsability of the task's developer to synchronize access to shared state.[4]

- *lowest-level tasks* wrap Java threads. They have a `Schedule` client interface. The code of their `execute` method simply consists in a loop in which they call the `schedule` method they are client of.
- *middle-level tasks* allow inter-schedulers scheduling. They are executed by schedulers and have a `Scheduler` client interface that they call in their `execute` method to schedule other tasks' execution. Such tasks are only used when schedulers are hierarchically organized. The DREAM framework does not currently provide such tasks.

### 4.2.3 Activity managers

For performance reasons, tasks and schedulers are not Julia components but simple Java objects implementing the `BindingController` FRACTAL interface. They can still be considered as FRACTAL components.

The DREAM framework provides interfaces to manage tasks and schedulers. These interfaces can be used by application components or by a third party (e.g. a process deploying the middleware). The `TaskManager` interface (figure 12) defines methods for registering, unregistering, and interrupting tasks.

[4] Use of synchronization can be avoided by either using schedulers that guarantee that only one thread at a time may execute the task.

```
package org.objectweb.dream.control.activity.manager;

public interface TaskManager {
  Object registerTask(Task task, Map hints)
      throws IllegalTaskException;
  void unregisterTask(Task task) throws IllegalTaskException;
  void interruptTask(Task task, TaskStoppedListener listener)
      throws IllegalTaskException;
  Task[] getTasks();
}
```

Figure 12: The `TaskManager` interface

## References

[1] B. Blakeley, H. Harris, and J.R.T. Lewis. *Messaging and Queueing Using the MQI: Concepts and Analysis, Design and Development.* McGraw-Hill, 1995.

[2] Microsoft Message Queuing (MSMQ), 2002. Microsoft, http://www.microsoft.com/msmq/.

[3] JORAM, 2002. ObjectWeb, `http://www.objectweb.org/joram/`.

[4] sonicMQ, 2002. Sonic software, http://www.sonicsoftware.com/.

[5] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3), 2001.

[6] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proceedings of ISSRE'98*, 1998.

[7] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2), 2003.

[8] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The Case for Reflective Middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[9] G. Blair, F. Costa, G. Coulson, H. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, and Jean-Bernard Stefani. The Design of a Resource-Aware Reflective Middleware Architecture. In *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99)*, pages 115–134, Saint-Malo, France, 1999.

[10] J. Loyall, R. Schantz, J. Zinky, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, J. Gosset, and D. Gill. Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, Phoenix, Arizona, USA, 2001.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), 2000.

[12] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4), 1998.

[13] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland, 2004.

[14] L. Bellissard, N. de Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Platform for Reliable Asynchronous Distributed Programming. In *Proceedings of SRDS'99*, 1999.

[15] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada, october 2001.

[16] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of WCOP'02*, 2002.

[17] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model, v2, 2003.

[18] S. McDirmid, . Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings OOPSLA'01, ACM Press*, 2001.

[19] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *16th European Conference on Object-Oriented Programming (ECOOP)*, 2002.