IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2005 Published by the IEEE Computer Society
Vol. 6, No. 9; September 2005

DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware

Matthieu Leclercq, *INRIA*
Vivien Quéma, *INRIA*
Jean-Bernard Stefani, *INRIA*

The DREAM component-based framework lets developers build, statically or dynamically, a variety of message-oriented middleware. Experiments show favorable results using it, compared with nonconfigurable, functionally equivalent middleware.

Asynchronous communication is one way we achieve scalability in applications made of loosely coupled autonomous components that communicate across large-scale networks.[1] Several *message-oriented middleware* systems have been developed in the past 10 years.[2,3] Research has focused primarily on supporting nonfunctional properties such as message ordering, reliability, security, and scalability; configurability has received less emphasis. From the functional point of view, MOMs implement a fixed API that provides a fixed subset of asynchronous communication models (publish-subscribe, event-reaction, message queues, and so on).

From the nonfunctional point of view, MOMs often provide the same nonfunctional properties for all message exchanges. This reduces their performance and makes them difficult or impossible to use with devices having limited computational resources. Because these nonfunctional properties haven't been developed as independent (removable) modules, removing them often requires the code to be totally reengineered.

Modular, composable architectures can overcome these limitations. Work on configurable systems has led, in particular, to the development of component-based and reflective middleware.[4] The idea is to build middleware as an assembly of interacting components, which we can statically or dynamically configure to meet different design requirements or environment constraints. Although in principle this approach applies to different forms of middleware, existing component-based middleware has mostly been used to implement synchronous communication paradigms and, with a few exceptions,[5,6] have not dealt systematically with resource configurability. Modular architectures have also been proposed to build routers[7] and communication subsystems.[8] Their main limitation is their restricted component model, which supports static configuration but not hierarchical composition or control capability. These limitations make it hard to administer and configure systems during execution. (See the sidebar "Related Work in Communication Middleware" for information about related work.)

DREAM (*d*ynamic *re*flective *a*synchronous *m*iddleware) is a software framework for building asynchronous middleware from components, which you can assemble statically or dynamically (at deployment time or at runtime). DREAM 's component library and set of tools let you build, configure, and deploy middleware that implements various asynchronous communication paradigms, including message-passing, event-reaction, and publish-subscribe. We'll show how to use our framework to dynamically control resource consumption and concurrency. The performance of dynamically configurable MOMs built with the DREAM framework compares favorably to monolithic, functionally equivalent middleware.

DREAM component architecture

The component model in DREAM is an extension of Fractal, a Java-based component model. It builds on Fractal's generic component framework,[9] which supports *hierarchical* and *dynamic* composition. Hierarchical composition supports system construction by assembling structured (hierarchical) sets of components. Dynamic composition provides the basis for dynamic reconfiguration, a useful feature for long-running systems. DREAM achieves dynamic composition through reflection mechanisms that enable system execution to be monitored and controlled at the level of individual components.

We distinguish two kinds of components: *primitive* components and *composite* components (which let you deal with a group of components as a whole). An original feature of the model is that you can include a given component in several other components. Such *shared* components are useful for modeling access to low-level system resources.

DREAM components communicate through either *server interfaces*, which correspond to access points accepting incoming method calls, or *client interfaces*, which correspond to access points supporting outgoing method calls. Particular input and output interfaces allow components to exchange messages, which are always sent from outputs to inputs. In the *push* connection mode, an output initiates a message exchange; in the *pull* connection mode, an input initiates an exchange.

Each component has two parts: a *controller*, which comprises interceptors as well as controllers, and *content*, which can be either a standard Java class in the case of a primitive component, or *subcomponents* in the case of a composite component. The controller can provide different levels of structural and behavioral reflection. For example, a component can provide

- a BindingController interface to allow binding and unbinding its client interfaces to server interfaces;
- a ContentController interface to list, add, and remove subcomponents in its contents; or
- a LifecycleController interface to enable control over its main behavioral phases in support of dynamic reconfiguration—for example, to start and stop a component's execution.

Figure 1 illustrates the various constructs in a typical DREAM component. The thick gray boxes denote the component's controller, and the boxes' interiors correspond to the component's content. The two shaded boxes represent a shared component. Arrows correspond to bindings, and the T-like structures protruding from the boxes are interfaces. Blue triangles represent input and output interfaces. Interfaces appearing on the top of a component represent controller interfaces such as a binding controller or a content controller.

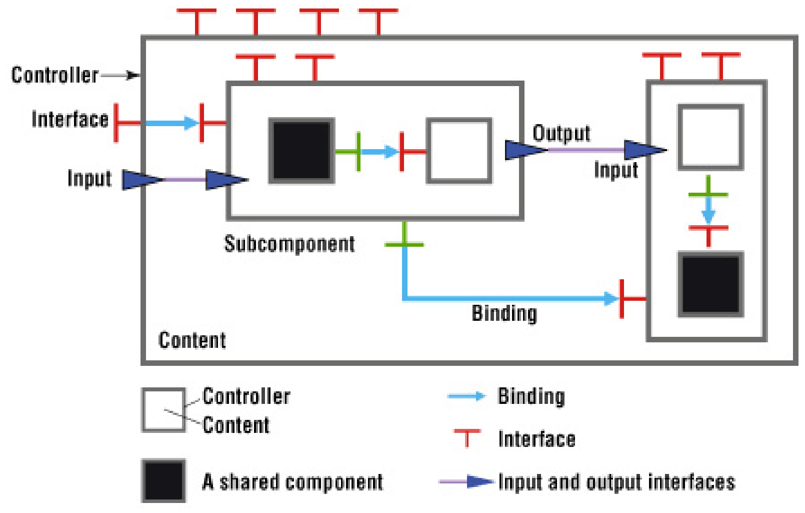


Figure 1. Architecture of a DREAM component.

The DREAM library

The DREAM library contains abstractions and components for resource management as well as functional components—that is, the components that implement the functions and behaviors that are embedded in asynchronous middleware.

Abstractions and components for resource management

The DREAM library defines abstractions and provides components for managing resources—that is, *messages* and *activities*. These components allow fine-grained control over consumed resources, a required feature for building scalable MOMs.

Message management. DREAM messages are Java objects that encapsulate named *chunks*. Each chunk implements an interface that defines its type. For example, messages that need to be causally ordered have a chunk implementing the Causal interface. This interface defines methods to get and set a matrix clock. A message can also encapsulate other messages and is identified by an interface called Message, which lets you access, add, or remove chunks and encapsulated messages.

Shared components, called *message managers*, enable DREAM components to create, duplicate, or delete messages, thus managing the memory resources in a MOM. By implementing pools of messages, the memory managers avoid unnecessary object creation and allow resource management policies to be implemented. For that purpose, all the methods defined in the MessageManager interface have a parameter (called `consumer`) that specifies the component calling the method. Message managers can use this parameter, for instance, to control the number of messages created for the different components.

Activity management. A DREAM component can either be *passive* or *active*. Active components define tasks to be executed, such as calls to other component interfaces; passive components don't. (Active components must implement the TaskController interface, which allows third parties to access the component's tasks.) 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 components that map higher-level tasks (to which its Execute client interface is bound) onto lower-level tasks (that are bound to its Schedule server interface). The number of scheduler levels is not limited. The DREAM framework currently provides various schedulers, including round-robin, FIFO (first-in, first-out), and periodic.

Tasks are components with an Execute server interface (defining an `execute` method) and a Schedule client interface. We distinguish three kinds of tasks:

- *Highest-level tasks* are registered by the MOM's components and contain functional code.
- *Lowest-level tasks* wrap Java threads.
- *Interscheduling (IS) tasks* are created by schedulers, to be scheduled by lower-level schedulers.

Figure 2 depicts an example of activity management. Components A and B have registered three tasks that are scheduled by two hierarchically organized schedulers. This produces the following result: the periodic scheduler periodically executes the IS and B tasks. Executing the IS Task triggers the sequential execution of tasks A1 and A2 (using the FIFO Scheduler). Note that the Periodic Scheduler is executed by two lower-level tasks that wrap threads.

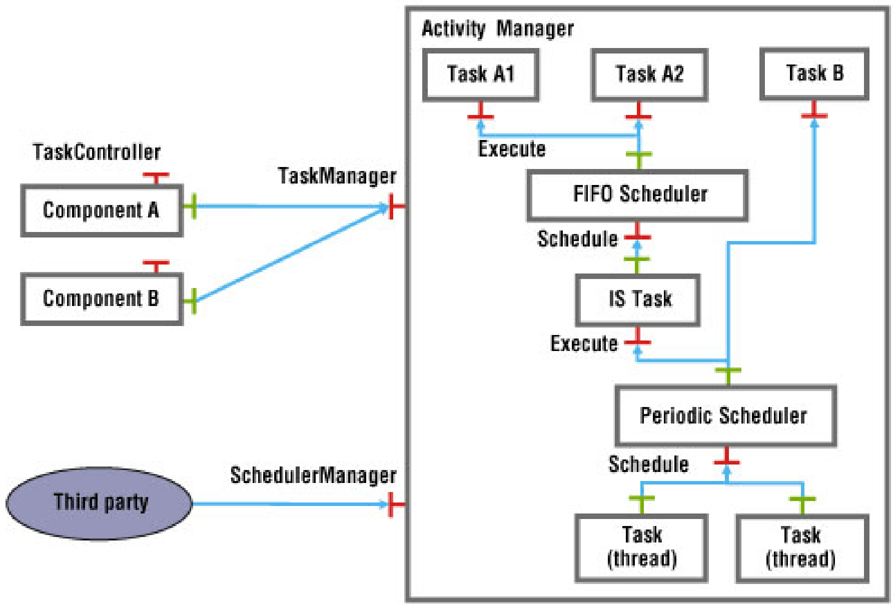


Figure 2. Activity management.

Functional components

The DREAM library's core components encapsulate functions and behaviors that are commonly found in asynchronous middleware. The library also contains specific components developed for particular middleware—for instance, components implementing event-reaction processing. (For space limitations, we do not present these components here.)

Message queues are used to store messages. Queues differ by the way messages are sorted (FIFO, LIFO, causal order, and so on) and by their behavior when capacity is exceeded (such as blocking versus removing messages) or when the queue is empty.

Transformers have one input and one output. They transform every message received on the input and deliver it to the output interface. One rule governs this transformation: a transformer must preserve the message's identity—that is, it can't change the Message interface that uniquely identifies the message. As a consequence, transformers can't deliver newly created messages. On the other hand, transformers can change the message's content: it may encapsulate new chunks and new submessages.

Pumps have one pull input and one push output. Their role is to pull messages from the input and then push them to the output.

Routers have one input and several outputs (also called "routes"); they forward messages received on their input to one or several routes. The routing process might involve message transformations (for example, to remove or manipulate routing information).

Duplicators have one input and several outputs. They duplicate messages received on their input to all their outputs.

Aggregators have one or several inputs to receive the messages to be aggregated, and one output to which to deliver the aggregated message.

Deaggregators reverse aggregators' behavior—that is, they take an aggregated message and generate appropriate individual messages from it.

Channels allow message exchanges between different address spaces. Channels are distributed composite components that encapsulate at least two components: a Channel Out, which sends messages to another address space, and a Channel In, which receives messages sent by the Channel Out.

DREAM tools

The DREAM framework currently provides two tools:

- A *deployment tool* uses a description of the component configuration, expressed in an Architecture Description Language, to proceed to its distributed deployment and configuration. This tool resembles other ADL-based deployment tools.[10]
- A *type-checking tool* defines a type system for DREAM messages that lets you check that a component configuration is correct—that is, that each component of the configuration will receive messages containing chunks with appropriate types.

Why a type-checking tool?

DREAM components can

- exchange messages,
- modify messages (for example, adding or removing a chunk), and
- behave differently according to the messages' content (for example, routing a message).

In the current implementation of the DREAM framework, every message has as its type the Message Java interface, independent of its contents. As a consequence, certain assemblages of DREAM components type-check and compile correctly but lead to one of three runtime failures. A chunk either

- is absent when it should be present (for example, for a read, remove, or update),
- is present when it should be absent (for example, for an add), or

- doesn't have the expected type (for example, for a read).

Figure 3a gives an example of an incorrect configuration: component ReadTS expects messages with a TS chunk, whereas component AddTS expects messages without the TS chunk. Because both ReadTS and AddTS receive exactly the same messages (duplicated by the Duplicator component), one of them will fail. The typing annotations are clearly insufficient.

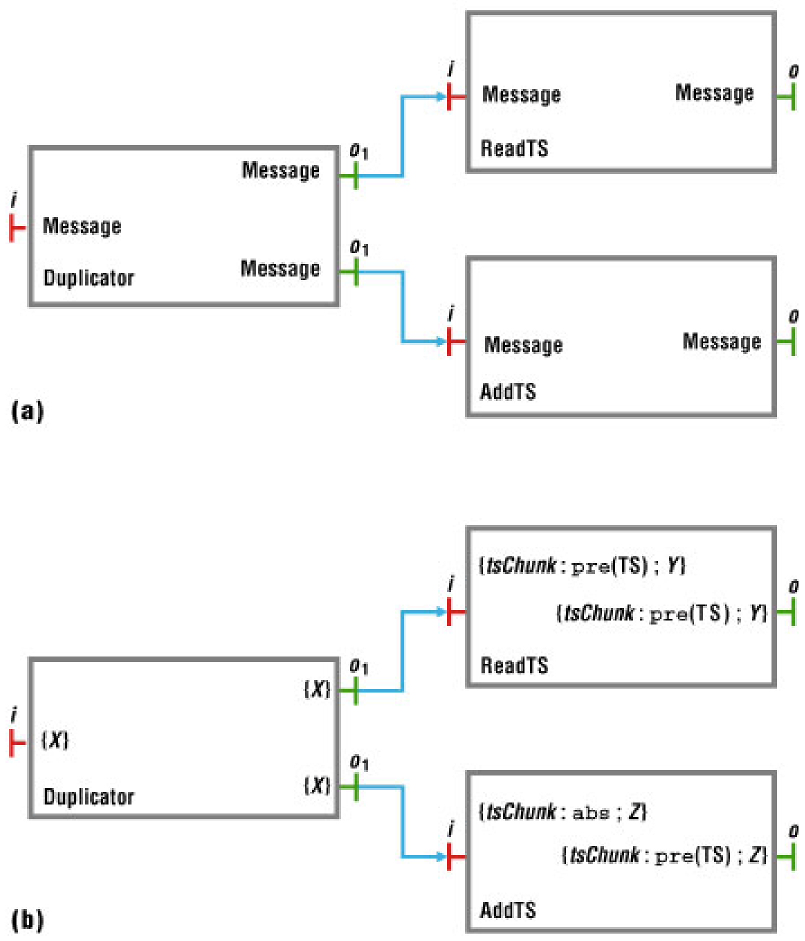


Figure 3. An example configuration (a) using the default type system and (b) using the newly defined type system.

Checking DREAM configurations

A polymorphic type system enables us to specify DREAM components' more common behaviors. It guarantees that, if components conform individually to their type, the composed system will not fail with any of the runtime errors we identified earlier.

The type system is an adaptation of existing work on type systems for extensible records. A record is a finite set of associations between names and values, also called *fields*. Elsewhere,[11,12] Didier Rémy describes an ML language extension that supports all the common operations on records—in particular, adding or removing fields and concatenating records. He then defines a static type system that guarantees that the resulting programs will not produce runtime errors, such as accessing a missing field. DREAM messages can be considered records, in which each chunk corresponds to a field of the record, and DREAM components can be considered polymorphic functions. Polymorphism is necessary because you can then use the same component in different contexts with different types.

Message types. We type messages as extensible records. Informally, a message type consists of a list of pairwise distinct names together with the type of the corresponding chunk, or a special tag if the message doesn't contain a given name. It also specifies the content of the (infinitely many) remaining names. Here are four examples of message types:

$$\mu_1 = \forall\{a: \text{ pre(CausalChunk)}; b: \text{ pre(IPChunk)}; \text{ abs}\}$$
$$\mu_2 = \forall\{a: \text{ pre}(X) \text{ ; abs}\}$$
$$\mu_3 = \forall\{a: Y; \text{ abs}\}$$
$$\mu_4 = \forall\{a: \text{ pre(CausalChunk) };Z\}$$

A message *m* of type μ_1 contains exactly two chunks named *a* and *b* and linked to chunks of type CausalChunk and IPChunk, respectively. (We'll discuss the pre constructor later.) *M* doesn't contain any other name, as specified by the `abs` tag. You can construct richer types using *type variables*. In type μ_2, *X* represents an arbitrary type. Informally, a message of type μ_2 must contain a name *a*, but μ_2 doesn't specify the associated chunk's type. Similarly, in μ_3, *Y* is a field variable. It can be either `abs`, or `pre(`*X*`)` where *X* is a type variable. Hence, the `pre` constructor lets us impose a given field, even if its type is unspecified. Finally, in μ_4, *Z* is a row variable that represents either `abs` or any list of associations.

Component types. Besides traditional interfaces, DREAM components have input and output interfaces that let them exchange messages. Each input or output is characterized by its name and the type of messages it can carry. A component's type is a polymorphic function type relating outputs to inputs. Here are some example components and component types:

$$\text{Duplicator} : \forall X.\{i: \{X\}\} \rightarrow \{o_1: \{X\}; o_2: \{X\}\}$$
$$\text{Add}_a : \forall X.\{i: \{a: \text{ abs}; X\}\} \rightarrow \{o: \{a: \text{ pre(IPChunk)}; X\}\}$$
$$\text{Remove}:_a \forall X, Y.\{i: \{a: Y; X\}\} \rightarrow \{o: \{a: \text{ abs}; X\}\}$$

The Duplicator component has a polymorphic type. Its input and outputs can be used with any type *X*. It duplicates the messages it receives on its input to all its outputs. The Add$_a$ component adds a new IPChunk with name *a* to the messages it receives on its input *i*. Note that these messages do not contain a chunk with name *a*. The Remove$_a$ component removes the chunk named *a* if it's present.

Example

Figure 3b depicts the same configuration as in figure 3a, using the type system we just defined. The configuration will be well typed if and only if we can solve these equations:

$\{X\} = \{tsChunk : \texttt{pre}(\text{TS}) ; Y\}$
$\{X\} = \{tsChunk : \texttt{abs} ; Z\}$

These equations don't have any solution, so the system is not well typed.

Evaluation

We experimented with and evaluated the DREAM framework by implementing the JORAM (Java open reliable asynchronous messaging) MOM.[13] JORAM is a JMS-compliant (Java Message Service) open source middleware. It comprises two parts: the ScalAgent MOM[14] and a software layer on top to support the JMS API.[13] The ScalAgent MOM provides a distributed programming model based on autonomous software entities called *agents* that behave according to an "event $\rightarrow$ reaction" model. The ScalAgent MOM comprises a set of agent servers (see figure 4), each of which contains three entities:

- The Engine component creates and executes agents and ensures their persistency and atomic reaction.
- The Conduit component routes messages from the Engine to the Networks.
- The Network components ensure reliable message delivery and causal order between servers.

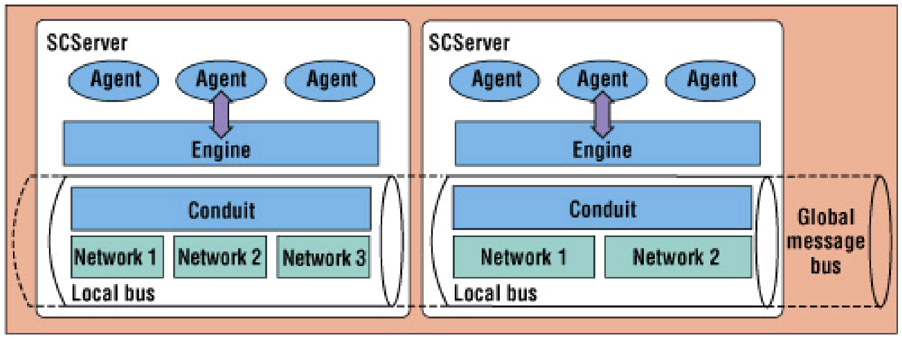


Figure 4. Two interconnected ScalAgent MOMs.

Implementing JORAM using DREAM

We implemented the ScalAgent MOM using DREAM (see figure 5). We preserved its main structures (Networks, Engine, and Conduit) to facilitate our functional comparison between the ScalAgent MOM and its DREAM reimplementation. The Engine has two main components: the Atomicity Protocol composite, which ensures the atomic execution of agents, and the Repository composite, which is in charge of creating and executing agents.

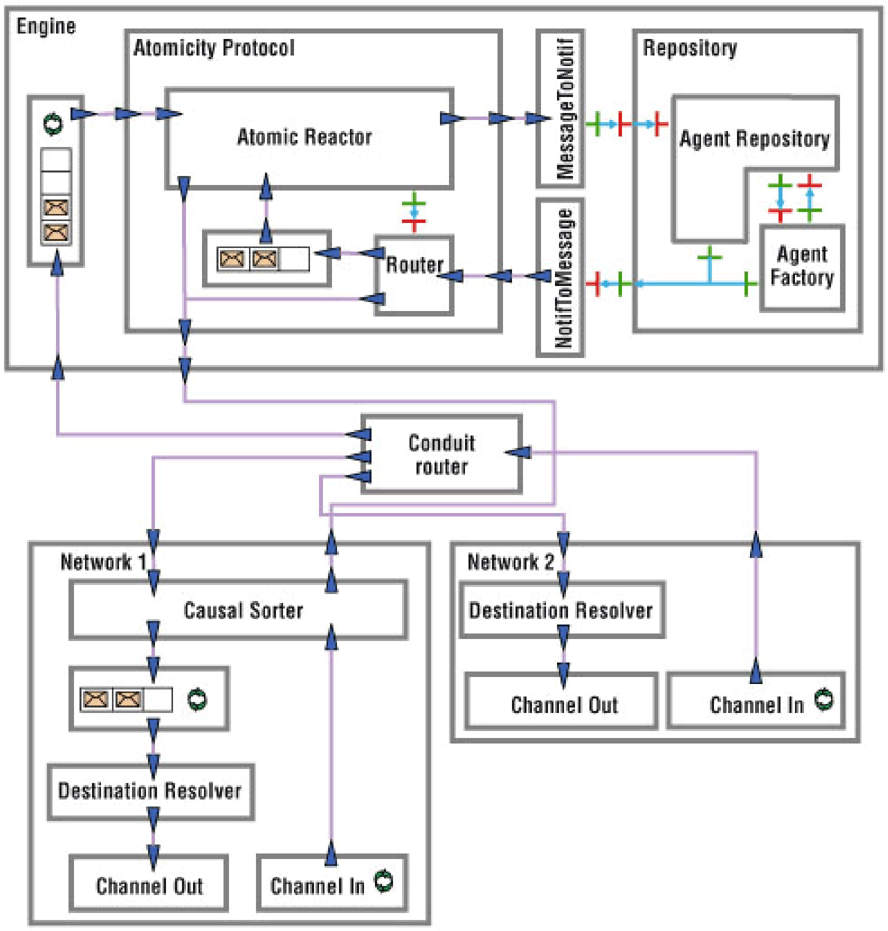


Figure 5. Architecture of an agent server.

Figure 5 shows two typical Networks. Both are composite components encapsulating a Channel In, a Channel Out, and a Destination Resolver component. Destination Resolver is a transformer component that adds the information required by the Channel Out component (IP address and port number). The Network 1 composite contains two more components: the Causal Sorter causally orders messages, and the message queue decouples the workflows of the Engine and the Network. We implemented the Conduit component using a router.

Configurability assessment

A first benefit of the DREAM implementation comes from the ability to easily change nonfunctional properties. For instance, removing causal ordering or removing the atomic protocol ensuring transactional execution of agents is straightforward. Both modifications only require a modification of the ADL description, or they can be applied at runtime. On the other hand, removing these properties from the ScalAgent MOM requires modifying and recompiling its source code.

Another benefit is that changing the number of active components encapsulated in the agent server is easy. The architecture presented in figure 6 involves three active components for an agent server with one Network. We can obtain a monothreaded architecture by removing the message queues encapsulated within the Engine and the Network. The only remaining active component is the Channel In component that listens on a socket.

Finally, DREAM enables us to build MOMs for embedded devices. We developed an agent server for mobile equipment with limited resources. This kind of equipment presents two characteristics: it might be temporarily disconnected from the Network, and it has limited storage capacity. Figure 6 shows how an agent server can be redesigned to take into account these characteristics.

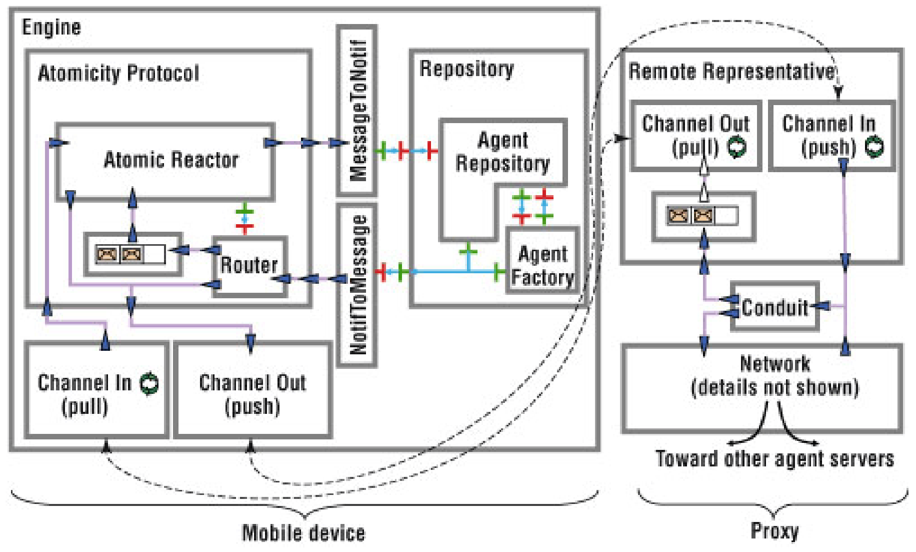


Figure 6. An agent server for mobile devices.

The agent server (on the left side of the figure) contains one composite. This composite is an Engine whose message queue has been replaced by a Channel In component and which encapsulates a Channel Out component to send messages. Messages intended for the mobile device are stored on another device (on the right). This device hosts a component that acts as a proxy for the mobile device's Engine. It's plugged to a Conduit router (like a traditional Engine). It has two functions:

- It receives messages intended for the mobile device and stores them in the queue. The mobile device can then pull these messages.
- It forwards messages that the mobile device sends.

This architecture preserves the MOM functionality while saving memory (the mobile-device part is monothreaded, messages are pulled instead of pushed, and it doesn't have the Causal Sorter and Destination Resolver components). Moreover, it lets the mobile device disconnect, because messages are remotely stored.

Performance comparisons

We compared the efficiency of the same application running on the ScalAgent MOM and on its DREAM implementation. The application involves four agent servers, each hosting one agent. Agents in the application are organized in a virtual ring. One agent initiates rounds, each consisting of forwarding a message originated by the initiator around the ring. We did two series of tests: messages without payload and messages embedding a 1-Kbyte payload. The experiments ran on four 1.8-GHz, 1-Gbyte Bi-Xeon PCs connected by a Gigabit Ethernet adapter, running Linux kernel 2.4.20.

Table 1 shows the average number of rounds per second and the memory footprint. We compared two implementations using DREAM with the ScalAgent implementation.

Table 1. Performance of the DREAM and ScalAgent implementations.

MOM	No. of rounds/sec.		Memory footprint (Kbytes)
	0-Kbyte messages	1-Kbyte messages	
ScalAgent	325	255	4 × 1447
DREAM (nonreconfigurable)	329	260	4 × 1580
DREAM (reconfigurable)	318	250	4 × 1587

The first implementation using DREAM isn't dynamically reconfigurable. As you can see, the number of rounds per second is slightly better (approximately 1.2 to 2 percent) than in the ScalAgent implementation. The small improvement comes from the message serialization, which is more efficient in DREAM . Concerning the memory footprint, the DREAM implementation requires 9 percent more memory, which can be explained by the runtime structure Fractal needs (approximately 70 Kbytes) and the fact that each component has several controller objects. This memory overhead isn't significant for a standard PC. The second implementation is dynamically reconfigurable; in particular, each composite component supports a lifecycle controller and a content controller. This implementation is slower than the ScalAgent one (approximately 2 to 2.2 percent) and only requires 7 Kbytes more than the nonreconfigurable DREAM implementation. Moreover, note that DREAM performances are proportionally better with 1-Kbyte messages than with empty ones. This happens because fewer messages are handled (more time is spent in message transmissions), thus limiting the impact of interceptors.

Table 2 reports on experiments we did to assess the impact of the concurrency level on the ScalAgent MOM's performance. We compared three architectures built using DREAM that differ by the number of active components they involve. In the two-thread architecture, we removed the message queue encapsulated in the Network; in the monothreaded architecture, we removed both active message queues (Engine and Network). We found that reducing the number of active components improves the number of rounds (3 to 5 percent for the two-thread architecture, 7 to 12 percent for the monothreaded architecture). We explain this by the fact that agents are organized in a virtual ring, so each agent server processes only one message at a time. As a consequence, only one thread is necessary.

Table 2. Impact of the concurrency level.

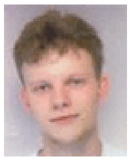
MOM (DREAM)	No. of rounds/sec.		Memory footprint (Kbytes)
	0-Kbyte messages	1-Kbyte messages	
3 threads	329	260	4 × 1580
2 threads	346	268	4 × 1516
1 thread	370	279	4 × 1452

Conclusion

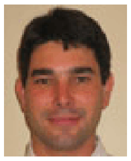
Our implementations gained in flexibility and configurability without significant loss of performance. The high configurability of DREAM can yield significant performance improvements by adapting a middleware architecture to its environment and application load. In our future work, we plan to refine the DREAM component library and to further substantiate our claim that DREAM architectures can successfully span the publish-subscribe design space. We'll do this by implementing different forms of publish-subscribe systems and experimenting with different scalability trade-offs. We also plan to apply the DREAM framework to the construction of highly dynamic notification systems for large-scale distributed system monitoring and supervision. DREAM is freely available under a LesserGeneralPublicLicense at http://dream.objectweb.org.

References

1. G. Banavar et al., "A Case for Message Oriented Middleware,"*13th Int'l Symp. Distributed Computing* (DISC 99), LNCS 1693, Springer, 1999, pp. 1—18.
2. Microsoft, "Microsoft Message Queuing (MSMQ) Center,"2005; www.microsoft.com/msmq.
3. R. Strom et al., "Gryphon: An Information Flow Based Approach to Message Brokering,"1998; http://researchweb.watson.ibm.com/distributedmessaging/papers/ext-abstract.pdf.
4. F. Kon et al., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB,"*Proc. Int'l Conf. Distributed Systems Platforms* (Middleware 00), LNCS 1795, Springer, 2000, pp. 121—143.
5. G. Blair et al., "The Design of a Resource-Aware Reflective Middleware Architecture,"*Proc. 2nd Int'l Conf.Meta-Level Architectures and Reflection* (Reflection 99), Springer, 1999, pp. 115—134.
6. J. Loyall et al., "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," http://doi.ieeecomputersociety.org/10.1109/ICDSC.2001.918993, *Proc. 21st Int'l Conf. Distributed Computing Systems* (ICDCS 01), IEEE CS Press, 2001, pp. 625—634.
7. F. Kon et al., "The Case for Reflective Middleware,"*Comm. ACM,* vol. 45, no. 6, 2002, pp. 33—38.
8. N. Bhatti et al., "Coyote: A System for Constructing Fine-Grain Configurable Communication Services,"*ACM Trans. Computer Systems,* vol. 16, no. 4, 1998, pp. 321—366.
9. E. Bruneton et al., "An Open Component Model and Its Support in Java,"*Proc. Int'l Symp. Component-Based Software Eng.* (CBSE 04), Springer, 2004, pp. 7—22.
10. N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," http://doi.ieeecomputersociety.org/10.1109/32.825767, *IEEE Trans. Software Eng.,* vol. 26, no. 1, 2000, pp. 70—93.
11. D. Rémy, "Type Inference for Records in a Natural Extension of ML,"*Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design,* C.A. Gunter and J.C. Mitchell, eds., MIT Press, 1993, pp. 67—95.
12. D. Rémy, "Typing Record Concatenation for Free," Proc. 19th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, ACM Press, 1992, pp. 166—176.
13. ObjectWeb, "JORAM: Java Open Reliable Asynchronous Messaging," 2005; www.objectweb.org/joram.
14. L. Bellissard et al., "An Agent Platform for Reliable Asynchronous Distributed Programming," http://doi.ieeecomputersociety.org/10.1109/RELDIS.1999.805107, *Proc. 18th Int'l Symp. Reliable Distributed Systems* (SRDS 99), IEEE CS Press, 1999, p. 294.



Matthieu Leclercq is a research engineer at INRIA. His research interests include component-based programming, architectures and designs of message-oriented middleware, and resource management in J2EE clusters. He received his MSc in computer science from Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble—Grenoble Institute of Technology (INPG). Contact him at INRIA, 655 avenue de l'Europe, F-38334 Saint-Ismier Cedex, France; matthieu.leclercq@inrialpes.fr.



Vivien Quéma is a PhD student at INRIA, where he is doing research on component-based middleware, architecture description languages, and Grid resource monitoring. He received his MSc in computer science from Grenoble Institute of Technology (INPG), where he is also a PhD candidate. Contact him at INRIA, 655 avenue de l'Europe, F-38334 Saint-Ismier Cedex, France; vivien.quema@inrialpes.fr.

Jean-Bernard Stefani is research director at INRIA. His research interests are process calculi, component-based systems design, and distributed algorithms. He received his degree from Ecole Polytechnique. Contact him at INRIA, 655 avenue de l'Europe, F-38334 Saint-Ismier Cedex, France; jean-bernard.stefani@inrialpes.fr.

Cite this article: Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, "DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware," *IEEE Distributed Systems Online*, vol. 6, no. 9, 2005.

Related Work in Communication Middleware

Several research areas relate to DREAM .

Reflective adaptable middleware

The past 10 years have seen a lot of activity related to building reflective middleware, as exemplified by the OpenORB,[1] DynamicTAO,[2] QuO,[3] and Hadas[4] projects. DREAM (*d*ynamic *re*flective *a*synchronous *m*iddleware differs in several main characteristics:

- DREAM is using and extending the Fractal component model, an original component model for Java. Unlike component models used in other reflective middleware, Fractal doesn't impose predifined reflective capabilities. On the contrary, it lets middleware developers design arbitrary metaobject protocols.
- DREAM and its component library target the construction of asynchronous middleware services. To our knowledge, other reflective middleware focuses on synchronous interactions.
- DREAM provides a set of tools that enable the configuration and deployment of asynchronous middleware built using the component library. In particular, DREAM provides a type-checking tool that defines a rich type system, enabling developers to detect incorrect architectures.
- DREAM provides resource management functions, in particular for activities. The only middleware we know of that has integrated such functionalities is OpenORB. Its activity model is similar to but less flexible than ours. QuO provides interesting means to enforce quality of service in an ORB. QuO's main contribution is the definition of languages that let QoS requirements be expressed. It enforces QoS mainly through interceptors, which developers could implement with Fractal interceptors.

Communication subsystems

Researchers have also worked on component-based communication subsystems. Several frameworks have been designed: Click,[5] Coyote/Cactus,[6] OSKit/Knit,[7,8] and APPIA.[9] The proposed component models are limited. They don't allow the dynamic manipulation of composite components, nor fully dynamic reconfiguration, available in Fractal. They provide mainly for static configuration. In contrast to DREAM , for instance, you can change the concurrency structure of a Click or Coyote protocol at runtime only if you've explicitly programmed it in the protocol implementation.

Asynchronous middleware

Several MOMs have been developed in the past 10 years: Astrolabe,[10] Gryphon,[11] MSMQ,[12] Siena,[13] and SonicMQ.[14] Research has primarily focused on supporting various nonfunctional properties, as exemplified by the work of Philippe Laumay and colleagues.[15] MOM configurability has been less emphasized. The only work relating to constructing configurable MOMs we know of is the work discussing the reflective features of Gryphon, a content-based asynchronous middleware.[11] However, to our knowledge, these haven't been implemented and come short of providing the level of reconfigurability that DREAM provides.

References

1. G. Blair et al., "The Design and Implementation of Open ORB v2,"*IEEE Distributed Systems Online,* vol. 2, no. 6, 2001.
2. F. Kon et al., "Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB,*Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms* (Middleware 00), LNCS 1795, Springer, 2000, pp. 121–143.
3. J. Loyall et al., "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," http://doi.ieeecomputersociety.org/10.1109/ICDSC.2001.918993, *Proc. 21st Int'l Conf. Distributed Computing Systems* (ICDCS 01), IEEE CS Press, 2001, pp. 625–634.
4. I. Ben-Shaul, O. Holder and B. Lavva, "Dynamic Adaptation and Deployment of Distributed Components in Hadas," http://doi.ieeecomputersociety.org/10.1109/32.950315, *IEEE Trans. Software Eng.,* vol. 27, no. 9, 2001, pp. 769–787.
5. E. Kohler et al., "The Click Modular Router," *ACM Trans. Computer Systems,* vol. 18, no. 3, 2000, pp. 263–297.
6. N. Bhatti et al., "Coyote: A System for Constructing Fine-Grain Configurable Communication Services,"*ACM Trans. Computer Systems,* vol. 16, no. 4, 1998, pp. 321–366.
7. B. Ford et al., "The Flux OSKit: A Substrate for Kernel and Language Research,"*Proc. ACM Int'l Symp. Operating Systems Principles* (SOSP 97), ACM Press, 1997, pp. 38–51.
8. A. Reid et al., "Knit: Component Composition for Systems Software,"*Proc. 4th Symp. Operating Systems Design and Implementation* (OSDI), Usenix Assoc., 2000, pp. 347–360.

9. H. Miranda, A. Pinto and L. Rodrigues, "Appia, a Flexible Protocol Kernel Supporting Multiple Coordinated Channels" http://doi.ieeecomputersociety.org/10.1109/ICDSC.2001.919005, (poster session), *Proc. 21st Int'l Conf. Distributed Computing Systems* (ICDCS 01), IEEE CS Press, 2001, pp. 707—710.
10. R. van Renesse, K. Birman and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining,"*ACM Trans. Computer Systems,* vol. 21, no. 2, 2003, pp. 164—206.
11. R. Strom et al., "Gryphon: An Information Flow Based Approach to Message Brokering,"*Proc. 9th Int'l Symp. Software Reliability Eng.* (ISSRE 98), IEEE CS Press, 1998.
12. Microsoft, "Microsoft Message Queuing (MSMQ) Center,"2005; www.microsoft.com/msmq.
13. A. Carzaniga, D. Rosenblum and A. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service,"*ACM Trans. Computer Systems,* vol. 19, no. 3, 2001, pp. 332—383.
14. Sonic Software, "SonicMQ,"2005; www.sonicsoftware.com/products/documentation/docs/mq_application_program.pdf.
15. P. Laumay et al., "Preserving Causality in a Scalable Message-Oriented Middleware,"*Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms* (Middleware 01), Springer, 2001, pp. 311—328.