Design Patterns for Safe Reflective Middleware

Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{seguti,nalini}@ics.uci.edu

Abstract

Flexible, scalable and customizable middleware can be used as enabling technology for next generation systems, where a wide range of services and activities must execute concurrently, non-disruptively and share resources. In order to avoid resource conflicts, deadlocks, inconsistencies and incorrect execution semantics, the underlying resource management system (middleware) must ensure that the concurrent system activities execute and compose in the correct manner. However, characterizing the semantics of shared resources and specifying what correctness of the overall system means is a difficult task. Patterns have proven to be useful to build flexible and modular frameworks by capturing the common structure and collaboration among their participants. Hence, they can be used to deal with semantical issues that guarantee correct composition of system resource activities in DRE systems, where issues of correctness and composition can be quite subtle, and complex.

1 Introduction

Middleware is the fundamental infrastructure that enables distributed computing. Flexible middleware frameworks incorporate the notion of reflection in order to provide the desired level of configurability and openness in a controlled manner. While such distributed middleware enables the modular connection of software components to manage the resources of an open distributed system, it must constraint the global behavior of the distributed system to ensure safety while providing for dynamic reconfiguration [9]. In DRE (Distributed Real-time and Embedded) systems, issues of correctness and composition can be quite subtle and complex. Interactions within and across components must be considered, the semantics of shared, distributed resources must be cleared spelled out and new notions of correctness of the overall system need to be developed.

On the other side, patterns and languages patterns have proven to be useful to build flexible and efficient software. The basic concept of a pattern is to capture the static and dynamic structure and collaboration among key participants

within a framework. Hence, patterns can be viewed as abstract descriptions of frameworks that facilitate widespread reuse of software architecture. When related patterns are woven together, they form a language that provides a process for the orderly resolution of software development problems[3]. In general, pattern languages help to alleviate the continual rediscovery and reinvention of software concepts and components by conveying a family of related solutions to standard software development problems [2]. Recently, pattern languages have been applied to develop dynamically configurable ORB middleware [4].

In this paper, we describe how we developed a framework to reason about composition of middleware services, based on the actor model of object distributed computation [6], and how we use design patterns to provide safe composition of middleware services.

2 The Two Level Meta Architectural Model

Since the actor model of computation [6] incorporates the notion of encapsulation and interaction only via message passing, it offers a clear, flexible and simple semantic approach to describe DRE systems based on incoming communications. The system is modeled as a group of self contained and independent autonomous objects, called actors, which communicate via asynchronous (buffered) message passing. On receiving a communication, an actor processes the message in a manner determined by its current behavior. As a result, the actor may: (1) Create new actors, (2) Change its behavior and (3) Send messages to itself or to other (existing) actors. Since mail addresses may be communicated in messages the configuration of the communication is dynamic and the activation order (one message activates another if the latter is sent during the processing of the former) determines communication patterns.

The two level actor machine (TLAM) model refines the actor model to specify, compose and reason about resource management services in open distributed systems[9]. In the TLAM model, a system is composed of two kinds of actors, base actors and meta actors, distributed over a network of processing nodes. Base actors carry out application level computation, while meta actors are part of the run-time system, which manages system resources and controls the run-time behavior of the base level. Meta actors communicate with each other via message passing as do base actors, and they may also examine and modify the state of the base actors located on the same node. Base level actors and messages have associated run-time annotations that can be set and read by meta actors. The annotations are invisible to base level computation. Actions which result in a change of base level state, are called events and meta actors may react to them if they occur on their node.

A TLAM configuration has a set of base and meta level actors and a set of undelivered messages. The actors are distributed over the TLAM nodes. Each actor has a unique name (actor identifier) and the configuration associates a current state to each actor name. The undelivered messages are distributed over the network (some are traveling along communication links and others are

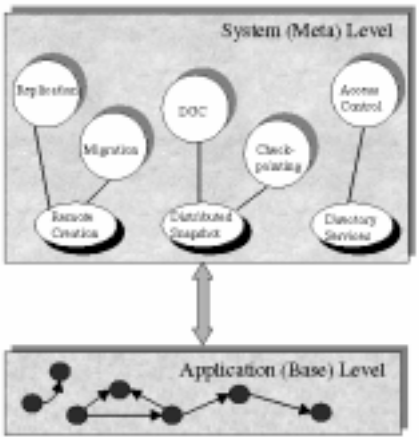

Figure 1: Classification of core services

held in node buffers).

To ensure non-interference and manage the complexity of reasoning about components of the DRE system, we identified key system services where non-trivial interactions between the application and system occur, i.e. base-meta interactions. We refer to these key services as *core services*. Core services are then used in specifying and implementing more complex activities as purely meta-level interactions. The development of suitable non-interference requirements allows us to reason about composition of multiple system services; these services have constraints that must be obeyed to maintain composability. We use commonly observed patterns in distributed system services to extract implementable abstractions and identify three meta-level core activities (See Fig 1):

- **Remote Creation:** - Recreation of services/data at a remote site. Remote creation can be used as the basis for designing algorithms for activities such as migration, replication and load balancing.
- **Distributed Snapshot:** Capturing information at multiple nodes/sites used as a basis for distributed garbage collection
- **Directory Services:** Interactions with a global repository. Directory services can be used to provide access control and implement group communication protocols.

3 Applying Design Patterns to Ensure Safe Composition

In order to develop a pattern language suitable to ensure safe composition of core services, we extend the *Active Object pattern* [10] by allowing to queue other active objects (*i.e.* actors), which may actually execute the method following the *command pattern* [5]. Thus, a given actor can handle many request without actually knowing what those requests do. Then we define the following patterns to describe the core services:

- **Remote Creation Factory (RCF):** This pattern provides a single component that creates actors in a distributed fashion (*i.e.* a specific node other than the node from which the creation is being initiated). By encapsulating the process of actor creation (local or remote) in the *RCF*, we can state requirements that ensure safe and correct composition of other resource management activities that use actor creation as a basis. For example, the *RCF* may further refined to encapsulate and consolidate replication, migration and load balancing mechanism that can be reconfigured dynamically through the use of the *Strategy pattern* [5]. By default, there is no acknowledgement from the *RCF*, but if the requester needs to know tha the request has been met, or the names of some of the newly created actors, then the addresses of the newly created actors can be returned in a message sent by the recently created actors.

- **Abstract Distributed Snapshot (ADS):** This pattern allows us to generalize the snapshot mechanisms used to capture the global state of the system, such as actor information, number of messages in transit or being processed, task queue sizes and reachability, in a distributed fashion. Since reachability characterizes the potential for communication of one actor with another, it forms the basis for resource management activities such as distributed garbage collection and checkpointing and helps to make run-time decisions that lead to efficient and safe management of a DRE system. As state information is accessible explicitly only in nodes, the *ADS* pattern ensures that node state information in channels are recorded at some node in the system without any interference at the application or system level, so application level computation and system services proceed concurrently with the snapshot in progress thereby preserving application and system semantics.

- **Directory:** This pattern allow us to generalize the mechanisms for identifying and locating actors as coordination and interaction patterns that can be used to define flexible and customizable naming schemes and security mechanisms. It also may be used to provide resource discovery and implement group communication protocols.

We use these patterns to provide a library of core services that allows a wide variety of resource management mechanism and policies to be composed,

attached and detached dynamically without requiring that the representation of one mechanism or policy knows about the others.

For Example, a remote creation request is given by a triple (α,ad,ν). This is interpreted as a request to create a base level fragment consisting of the actors created and messages sent when ad is executed on node, ν. The fragment is independent of the node and configuration up to choice of new actor addresses. The actor α is specified as the creator/sender of actors and messages in the newly created fragment. Assuming that every node in the system has an instance of a *RCF* and a Node Manager, which takes care of inter-node communication, the remote creation is done in two phases:

1. *Handshake:* The local *RCF* requests its node manager (caller) to negotiate the remote creation request. Then, both node managers (caller and callee) check security and resource availability.

2. *Local Creation:* Once the handshake has been successfully accomplished, the remote *RCF* locally creates the actor (or actors) requested.

Besides to allows us to consolidate actor creation (locally or remotly), the *RCF* pattern give us the flexibility to create a set of actors within the same request and it opens the possibility to change dynamically between different compiler transformations to enhance the performance of remote creation in a per node basis. By encapsulating the interactions between the base and meta level actors within the *RCF*, services using it are assured of correct base-meta interactions without interference from other system activities. *e.g.* the migration service allows for actor relocation for easier access, availability and load balancing. Since relocation can be seen as a state transfer, which is a specialization of a remote creation, we built the migration service using the *RCF* pattern as follows: A migration request is interpreted as a request to move the computation carried out by a local actor α to the node ν. In order to state explicitly the invariants that need to be maintained, we classify the migration process into 3 phases

1. *Initialization:* It determines a safe point in which the actor computation should be suspended, it specifies the current actor state and creates a *surrogate actor*, which will receive the new actor address.

2. *Remote Creation:* Using the *RCF*, a remote creation request is issued to the node ν with the actor state as a parameter. Then the *RCF* will create the actor and substitute the initial state with the current actor state. Here we use the *Decorator pattern* [5] to extend the responsibilities if the *RCF* and be able to send the state of the actor as a parameter in the remote creation request.

3. *Rerouting:* It establishes transparent access to the migrated actor by providing a special actor (*forwarder*) that redirects all the incoming messages from the previous to the current actor location.

4 Concluding Remarks

In this paper, we described a meta-architectural framework that help us to gain understanding of the semantic issues involved in the composition and customization of middleware services, and how a pattern language can provide safe composition of these services by constraining their collaboration and provide a new layer of abstraction that allow services to reconfigure themselves in a dynamic fashion. In general, the dynamic nature of DRE applications under varying network conditions and request traffic imply that the underlying middleware framework must be dynamic and customizable. We believe that composable and safe middleware frameworks that implement cleanly defined meta architectures enable customization of applications and system services; this will provide a foundation for the evolution of large scale distributed computing.

References

[1] Chandy K.M. and Lamport L. Distributed Snapshots: Determining Global States of a Distributed System. In *ACM Transactions on Computer Systems*, 1985.

[2] Douglas C. Schmidt. Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM*, 38, 1995.

[3] Douglas C. Schmidt. Applying Patterns and Frameworks to Develop Object-Oriented Communication Software. *Handbook of Programming Languages*, 1, 1997.

[4] Douglas C. Schmidt and Chris Cleeland. Applying a Pattern Language to Develop Extensible ORB Middleware. *IEEE Communication Magazine*, 37(4), 1999.

[5] Erich Gamma, Richard Helm, Ralp Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] Gul Agha. *Actors: A model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[7] Nalini Venkatasubramanian. *An Adaptive Resource Management Architecture for Global Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1998. Department of Computer Science.

[8] Nalini Venkatasubramanian and Carolyn Talcott. Integration of Resource Management Activities in Distributed Systems. Technical Report Computer Science Department, Stanford University, 1995.

[9] Nalini Venkatasubramanian, Mayur Deshpande, Shivjit Mohapatra, Sebastian Gutierrez-Nolasco and Jehan Wickramasuriya. Design and Implementation of a Composable Reflective Middleware Framework. In *International Conference on Distributed Computer Systems (ICDCS-21)*, 2001.

[10] R.G. Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Pattern Languages of Program Design*, 1996.