Locators – Dynamic Service Composition and System Evolution

Robert Hirschfeld[*], Jeff Eastman[+], Matthias Wagner[*], Hendrik Berndt[*]

[*]DoCoMo Communications Laboratories GmbH
Landsberger Strasse 308-312, 80687 Munich, Germany

[+]Windward Solutions, Inc.
1081 Valley View Ct., Los Altos, CA 94024, USA

hirschfeld@docomolab-euro.com

Abstract

We present the concept of locators in distributed computing environments. Locators are designed to make service composition transparent to component developers. They have first been implemented in the Aero distributed processing environment to enable seamless dynamic service composition and system evolution. Aero manages complex CORBA-based distributed systems based on the distributed processing environment specification of TINA-C. With Aero's locators, the development and deployment of distributed systems is significantly simplified: Developers only need to specify structural relationships between computational objects to cause the automatic generation of all necessary code to interface with the run-time environment for system composition and re-composition via locators. Locators facilitate post-deployment run-time service evolution.

1 Introduction

In distributed computing environments software services are implemented by service components and made accessible to potential clients via service interfaces. When systems get more complex, services are typically composed recursively out of other services with the one service component acting as the client of the other service components. In these environments, clients and services as well as the domain to be implemented by a service-based application are often physically distributed. Improving system scalability and performance by means of load balancing and replication adds to inherently distributed nature of these service-based systems.

Fundamental challenges arise in building distributed systems [12]. This includes handling of partial failure modes due to failed processes and network partitioning, ensuring stability of relationships between loosely-coupled service components, or mechanisms for introduction and reintroduction of service components. We have designed the Aero [9] distributed processing environment (DPE) to also address these challenges for the development, deployment, and run-time evolution of CORBA-based distributed services [8, 7]. In this paper we describe Aero's locators as an essential and effective means to support dynamic service composition during initial system deployment, system recovery from partial failures, and system evolution after deployment at run-time.

The rest of the paper is organized as follows: Section 2 we give a brief overview of Aero. In section 3 we introduce locators and discuss their usage for system composition and re-composition throughout failure recovery and graceful system evolution. Finally, in section 4 we will summarize and conclude.

2 Aero Overview

The Aero DPE manages complex CORBA-based distributed systems based on TINA-C specifications [2, 3, 4, 5]. TINA-C stands for Telecommunications Information Networking Architecture Consortium, an international research effort to converge telecommunications and distributed object-oriented computing. During system design, software developers employ TINA's object definition language (ODL), an extension to OMG's interface definition language (IDL). In addition to basic IDL properties, ODL allows the specification of trading attributes associated with interface descriptions as well as organizational run-time structures like computational objects and groups [10]. Furthermore, ODL allows computational objects to support multiple interfaces.

```
module Example {
        group GroupA {
                components ObjectA, GroupB;
                contracts InterfaceB, InterfaceD;
        };
        object ObjectA {
                behavior behaviorText "This object does something
useful";
                requires InterfaceD;
                supports InterfaceA, InterfaceB, InterfaceC;
        };
        interface InterfaceA {};
        interface InterfaceB {};
        interface InterfaceC {};
        group GroupB {
                components ObjectB;
                contracts InterfaceD;
        };
        object ObjectB {
                behavior behaviorText "This object does something
useful, too";
                supports InterfaceD, InterfaceE;
        };
        interface InterfaceD {};
        interface InterfaceE {};
};
```

Figure 1: Simple ODL module [4]

Figure 1 depicts an ODL example that will be used throughout the paper to illustrate how locators are derived and used. With ODL, both static and dynamic relationships to be present at run-time are described simultaneously: The static part states that there are interfaces A to E, object A supporting interfaces A to C, object B supporting interfaces D and E, group A managing object A and its subgroup B, and subgroup B managing object B. The more dynamic part covers that group B exports interface D which is supported by object A (interface visibility control along the group hierarchy), and that object A requires another object that supports interface D (in Figure 1, this part describing dynamic relationships is rendered in italics).

Aero's run-time environment which implements TINA DPE concepts [11], uses trading to dynamically compose services at run-time [1]. Trading is based on matching interfaces required by computational objects with interfaces supported by other computational objects. Trading is built around interface types and hierarchical introduction spaces denoted by computational groups. If necessary, additional trading attributes can be utilized to find the most suitable interface-object combination as a match.

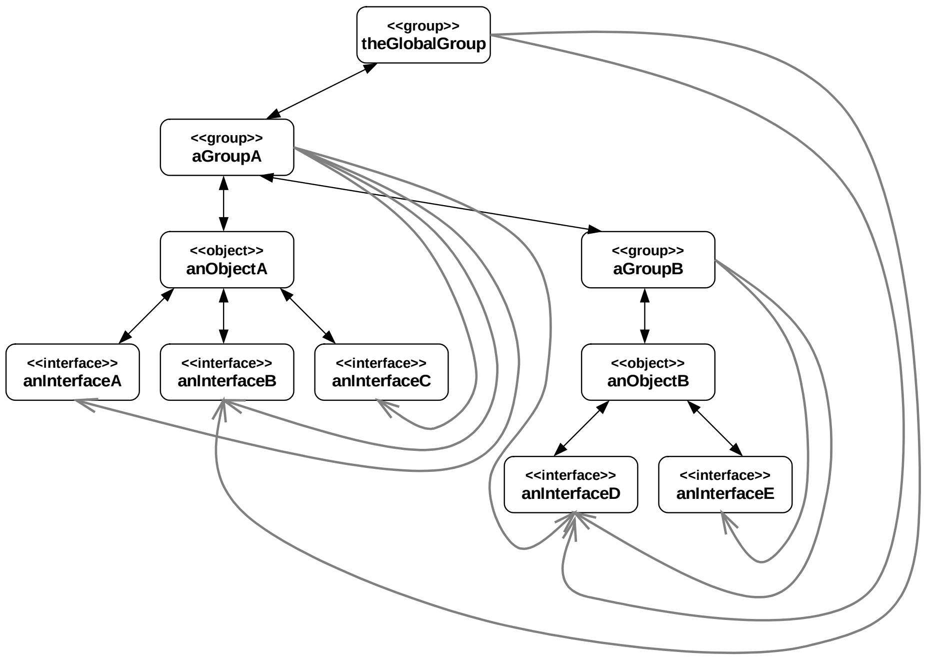

Figure 2: Static and dynamic relationships based on ODL [4]

Figure 2 shows an instantiation of the system described in Figure 1. Solid black arrows represent relationships resulting from the instantiation of the group hierarchy, aggregating groups and subgroups, objects, and interfaces. Solid gray arrows symbolize references to exported interfaces of objects up the group hierarchy. The dotted arrow indicates a traded reference which is completely dynamic and solely depends on the availability of matching interfaces exported and required by active computational objects. All code necessary to instantiate a group hierarchy (our hierarchical introduction space), its computational objects and their interfaces, as well as its incorporation into the trading run-time framework are generated by Aero utilizing Aero's metaobject repository. This repository is a run-time database containing information about deployable and deployed services, which is in part derived from ODL service specifications. Per default, only one computational object specified in a group is instantiated.

However, developers may decided to create multiple instances of the same object in the context of a particular group to allow for redundancy. Redundancy can be achieved furthermore by trading matches out of different branches along the trading hierarchy. Part of the code generated are locators that, in combination with Aero's trading facilities and introduction spaces, free developers from writing code related to initial system composition, as well as system re-composition in response to partial failures and system evolution. In the next section, we describe locators in more detail.

3 Locators

Locators are proxy objects that support transient access to traded interfaces. They are placed in-between client code, written by developers, and CORBA client stubs, generated by Aero (Figure 3). Locators maintain the search criteria used to locate an interface and a reference to the interface matched by the trader. For a client, there is a one-to-one relationship between the locator used and the interface referenced. Methods invoked on the locator are transparently forwarded to the referenced interface (Proxy [6]).

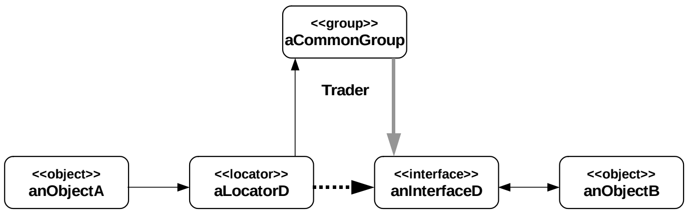

Figure 3: Locators, traders, interfaces

Network partitioning, component failure or system evolution can result in communication errors. Figure 4 illustrates such situation: Object A tries to communicate with object B via interface D. Since object B or interface D are not available anymore, or because both objects were separated as a result of a network partitioning, this attempt to communicate fails and results in an error. Locators can trap such error conditions and take corrective action. Since locators contain all trading information sufficient to obtain references to traded interfaces, they can transparently re-acquire references to new interfaces (via Aero's hierarchy of traders and introduction spaces) that are similar to the ones malfunctioning but operational. State transfer has to be addressed separately, if necessary.

Our example considers several changes at run-time that can be described as unanticipated, amongst them component and communication failures, as well as system migration and evolution. Note that in Figure 4 and Figure 5 there are two instances of object B supporting interface D and E. Multiple instances can exists as replicas to allow for redundancy where partial failures would cause a fail-over to a functioning component supporting the required service interface(s).

While partial failures are usually caused by broken system parts, they can be triggered on purpose. This method is utilized for system migration as follows: New component implementations supporting the expected interfaces are

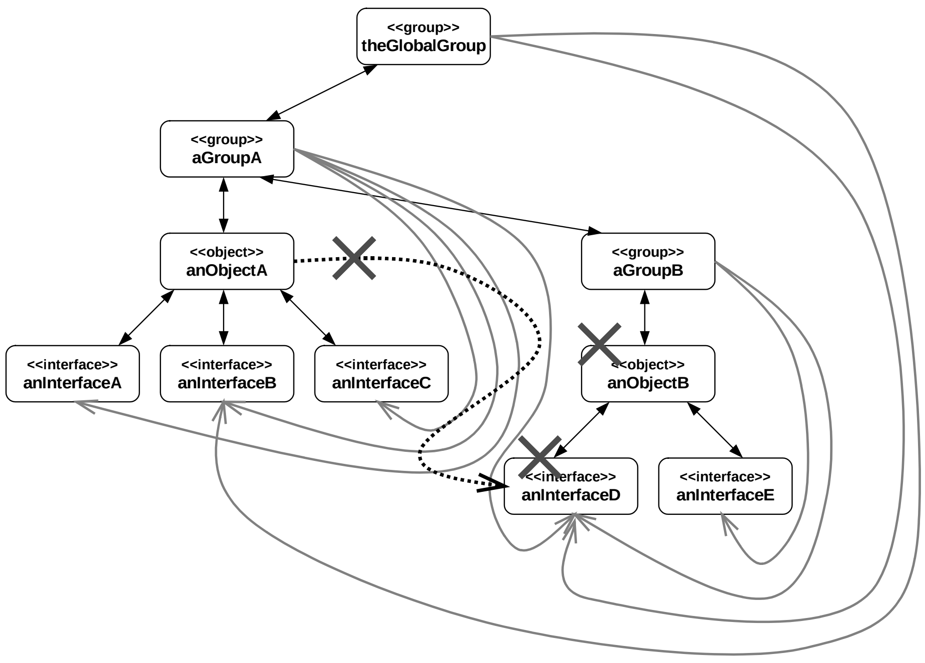

Figure 4: Invalid reference to interface D due to partial failures

instantiated in parallel to the ones to be replaced. All processes hosting old components are terminated and with that cause the signaling of partial-failure mode in their respective clients. Such signaling is trapped by our locators that in response try to re-acquire new references to new interfaces of operational components and will find our newly instantiated components based on our new implementation (Figure 5).

Locators are designed to make service composition transparent to component developers. Developers are provided with information about what a service component has to offer (via its provided service interfaces), and what other services this component can depend on (via its required service interfaces). Developers need to implement the former part. But they do not have to be concerned about neither the implementation nor the access to the latter part which is taken care of by locators. However, if necessary, developers are allowed to customize locator behavior to meet special conditions, for example by providing additional values for trading attributes.

The use of locators can minimize or even avoid code dedicated to system composition via trading, error detection and resulting corrective actions including system re-composition.

4 Summary

In this paper we have illustrated the concept of locators in distributed computing environments. Locators have first been implemented in the Aero DPE to facilitate seamless dynamic service composition and system evolution. Aero has

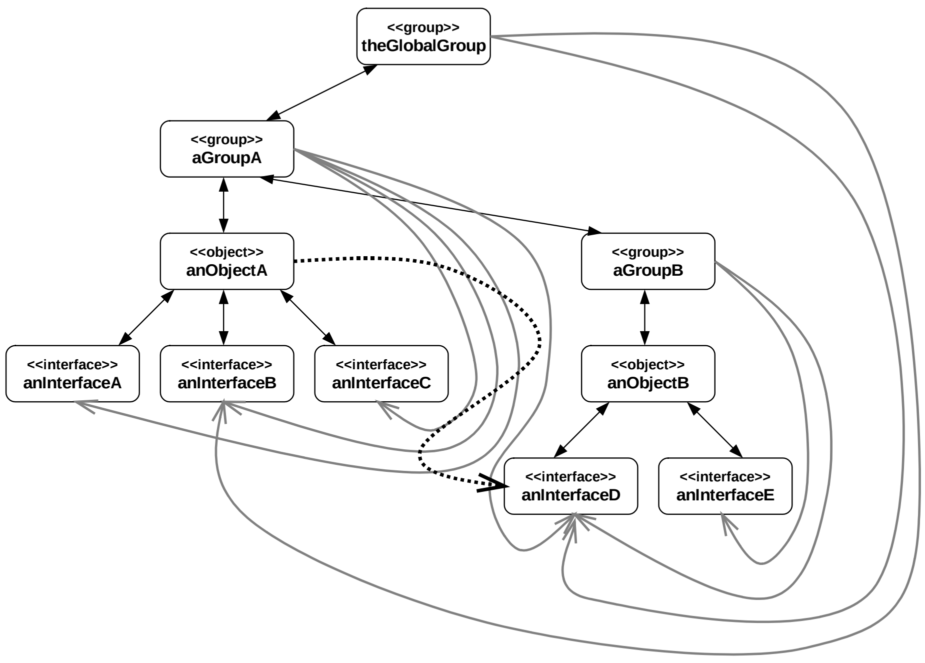

Figure 5: acquired reference of object A to another interface D

been used to deploy large-scale telecommunications applications based on the TINA service architecture. There, Aero's locators proved to be essential and effective during system activation, recovery, and post-deployment evolution at run-time.

During development, programmers only need to specify structural relationships between computational objects using ODL. Aero then generates all code necessary to interface these objects with the run-time environment. Locators are part or the code generated. Locators support transparent service composition. They are placed between client code and client stubs to remote objects, containing all information required to locate and connect to remote objects that provide interfaces required by the computational object locators are associated with. As the main benefit for developers, there is no need to explicitly implement connectivity- and composition-related system parts, but to take advantage of partial system specification and code generation. For a computational object requiring other services to fulfill its responsibilities, locators assure operational connections to other computational objects which provide the required interfaces. Locators assist in the initial composition of the system as well as recomposition addressing communication errors due to partial failures and system migration and evolution.

In Aero, locators are used for system evolution via the provisioning of adapted services, the instigation of partial failure modes, and as a consequence of that, the automatic failover to the evolved parts of the system.

References

[1] D. Bäumer and D. Riehle. Product Trader. In *In Pattern Languages of Program Design III.* Addison-Wesley, 1998.

[2] H. Berndt, editor. *The TINA Book: Cooperative Solution for a Competitive World.* Prentice Hall, 1999.

[3] J. Eastman and R. Hirschfeld. Meta-Object based System Generation. In *Proceedings of STJA'97*, Erfurt, Germany, 1997.

[4] J. Eastman and R. Hirschfeld. A Trading-Based Component Environment. In *Proceedings of STJA'98*, Erfurt, Germany, 1998.

[5] J. Eastman and R. Hirschfeld. Repository-Based Deployment of CORBA Applications. In *Proceedings of COMDEX Enterprise'98, TelecomIT Forum*, 1998.

[6] E. Gamma, R. Helm, R. Johnson, , and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[7] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1996.

[8] F. Leong, S. P. Mylavarabhata, T. Nguyen, and F. Quemada. Distributed Processing Environment: A Platform for Distributed Telecommunications Applications. *Hewlett-Packard Journal*, October 1996.

[9] Windward Solutions. Aero website. http://www.windwardsolutions.com/Aero.

[10] TINA-C. TINA Object Definition Language. TINA-C Document, Version 2.3, July 1996.

[11] TINA-C. TINA DPE Architecture. TINA-C Document, Version 2.0b, November 1997.

[12] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, November 1994.