Service Lifecycle in a Distributed Computing Environment

J. Eastman, I. Fuller, R. Hirschfeld
Windward Solutions, Inc.
395 W. El Camino Real, 2nd floor
Sunnyvale, CA 94087

Abstract - **This paper presents an environment for building, deploying and managing distributed applications that is based upon a subset of the Distributed Processing Environment specification of the Telecommunications Information Network Architecture Consortium.**

I. INTRODUCTION

Most of today's CORBA-based distributed systems may be characterized as multiple language implementations, running as multiple interdependent processes on multiple hardware platforms within multiple vendor environments [1]. This complexity is one of the main reasons that organizations have difficulties in organizing and deploying such systems.

This paper presents an environment that helps to deal with these complexities. It will show how to organize, package, map, deploy, monitor and evolve a CORBA-based system using a subset of the Distributed Processing Environment specification of the Telecommunications Information Network Architecture Consortium (TINA-C) [2, 3, 4].

II. THE ENVIRONMENT

The environment that implements the concepts discussed in this contribution is called Aero [6, 7, 8]. Aero includes two main parts – the Distributed System Schema repository (DSS), and a Distributed Processing runtime Environment (DPE).

The DSS contains meta-information about several aspects of a distributed application:

- its logical organization,
- its physical organization,
- its target hardware environment,
- its service deployment status.

This meta-representation enables a system administrator to deploy, monitor and control the application services and environment at runtime. Input to the DSS may be specified in OMG IDL, ODMG ODL, and TINA ODL definition languages, in addition to a number of supplied graphical user interfaces. The DSS is used to manage the specifications and to produce implementation skeletons in Java, Smalltalk, and C++ according to standard language bindings [5]. One advantage of the DSS meta-model is that new languages may be supported in future without disturbing the contents of the repository.

The DPE consists of programming language specific frameworks and operating system specific components that allow deployment and management of applications represented within the DSS. These include language-specific implementations of interface, object, group, and capsule artifacts that are installed with each application component. The DPE has a daemon process that runs on each system hardware node. This process communicates with the DSS and acts as its agent on the target platform.

The DSS and the DPE work together in a network to manage deployment, monitoring and migration of these applications. The Aero environment supports the following capabilities:

- Representation of system/service organization,
- Generation of implementation skeletons and customized frameworks,
- Modeling of platform and network organization,
- Installation, activation, and shut down of service components,
- Monitoring of operational status,
- Capsule and service restart for failure recovery,
- Graceful system evolution.

A. *Representing System and Service Organization*

The Aero DSS contains meta-information that reflects both the logical and the physical organization of distributed applications. Its Interface Repository is used to maintain information about the interfaces, objects, and groups that have been defined for applications. This information is available at runtime to provide insight into the logical structure of the application system.

Aero's Service Repository contains additional meta-information that represents the physical packaging decisions that have been made for deploying applications. Services are represented by a single root group construct (the service group), and that group may contain sub-groups. Each group may be packaged into a different deployment unit, and thus execute in a separate process at run-time. Services are defined in terms of sets of such deployment packages. Currently a deployment package may be written in the Java, Smalltalk, or C++ programming languages.

B. *Generation of Implementation Code*

Once packaging decisions have been made, the Interface Repository is used to generate implementation code in the desired programming language(s). Each supported language includes runtime code frameworks that provide the basis of application-specific group, object, and interface implementa-

0-7803-5785-X/$10.00 © 1998 IEEE

tions. The skeletons that are generated customize these entities as defined by the logical structure of the application.

Completed deployment packages are prepared by integrating the generated code with implementations provided by developers. These packages and any associated files are then entered into the Service Repository so that they may be automatically deployed. Service packages are characterized by their host platform, operating system, networks and other required properties of the deployment host. By compiling equivalent packages for different host environments, the selection of appropriate packages for different types of deployment platforms may be automated.

C. *Modeling the Deployment Environment*

Aero's Node Repository contains a set of references to daemon processes that reside on each target node. Information is maintained by each such process to allow the correct package to be selected for installation during deployment. Each daemon is responsible for managing local packages for its installed components, for spawning capsule processes under the direction of the DSS, and for subsequent monitoring and restart of those capsule processes.

D. *Service Mappings and Deployment*

In order to deploy a service thus modeled, each of its component packages must be mapped to an execution node. The Aero Mapping Repository has a user interface that allows this information to be specified separately for each instance of each service to be deployed. Once all required packages have been so mapped, the Mapping Repository can automate the installation, startup, instantiation, and shutdown of services with no further human intervention.

Service instantiation involves messaging among the service's capsules to cause the creation and registration of groups, objects and interfaces within the group trading hierarchy. Aero's generated groups utilize the trading attributes of defined interfaces to register them within their local group. Contracts declared in the group definitions then determine how far up the trading hierarchy each traded interface reference is propagated. Clients desiring an interface of a particular type submit requests to their local group (representing a local trading access point) and these requests propagate upward until they are satisfied at the appropriate level in the group/trader hierarchy.

E. *Service Monitoring and Restart*

Once the packages that comprise a service have been installed and activated by the Mapping Repository, each node daemon can be configured to automatically poll the capsule processes to detect failures at the network, process or application levels. Detection of capsule failure can then result in the automatic restart of the package.

If the service has already been instantiated, then groups, objects and interfaces of related components that are running in other capsules may hold object references into the failed capsule. These references are encapsulated by Aero Locators that trap distribution-related errors and contain trading information sufficient to re-acquire new references to traded interfaces. Similarly, Aero group implementations are designed to purge references to interfaces that become unavailable due to capsule failures.

Once a capsule has been restarted after a failure, new entities are instantiated within it and other capsules have an opportunity to take additional corrective actions. The result is a service group structure that spans multiple process capsules and heals itself after capsule failures.

F. *Service Evolution*

Aero's Mapping Repository is fully dynamic, allowing new packages to be mapped and deployed without shutting down the whole service. This capability can be used to add redundant groups, to move capsules from host to host, and to otherwise alter the deployment characteristics of the service. Aero's full reflective knowledge of the deployed service is essential for tracking the interdependencies between deployed components.

Graceful evolution of a deployed service in the face of change utilizes many of the same mechanisms. New versions of packages need to be mapped and installed, new capsules need to be activated and the trading structure needs to be adjusted to make the new object implementations available. Once new groups have been instantiated and the group structure has been adjusted to include them, the old capsules can simply be terminated. The client's error recovery mechanisms will correctly reacquire new traded interfaces and minimal interruption will be perceived.

REFERENCES

[1] OMG: "The Common Object Request Broker: Architecture and Specification. Revision 2.0," *OMG*, July 1995, Updated July 1996.

[2] TINA-C:"TINA DPE Architecture," *TINA-C Document*, Version 2.0b0, November 1997.

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

[4] L.A. de la Fuente, T. Walles, "Management Architecture," *TINA-C*. December 1994.

[5] J. Eastman, R. Hirschfeld, "Meta-Object based System Generation" *STJA'97 Proceedings*. Erfurt 1997.

[6] J. Eastman, R. Hirschfeld, "A Trading-Based Component Environment" *STJA'98 Proceedings*. Erfurt 1998.

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

[8] http://www.windwardsolutions.com/Aero.