INTRODUCTION AND OBJECTIVES

This chapter resumes our survey of application-level fault-tolerance protocols considering approaches based on aspect-oriented programming.

Aspect-compliant programming languages allow a source code to be regarded as a pliable web that the designer can weave so as to specialize or optimize towards a certain goal without having to recode it. This useful property keeps concerns separated, bounds complexity, and enhances maintainability. Aspect programs may be used for different objectives, including non-functional properties such as dependability. To date, it is not known whether aspect-orientation will actually provide satisfactory solutions for fault-tolerance in the application layer. Some researchers believe this is not the case (Kienzle & Guerraou, 2002)—at least for some fault-tolerance paradigm. Some preliminary studies have been carried out (for instance in (Lippert & Videira Lopes, 2000)), but no definitive word has been said on the matter. It is our belief that, at least for some paradigms, aspects may reveal themselves as invaluable tools to engineer the application-level of fault-tolerance services. For this reason their approach is described in this chapter.
FAULT-TOLERANT PROTOCOLS THROUGH ASPECT ORIENTATION

General Ideas

Aspect-oriented programming (AOP) (Kiczales et al., 1997) is a programming methodology and a structuring technique that explicitly addresses, at application level, the problem of the best code structure to express different, possibly conflicting design goals such as high performance, optimal memory usage, and dependability.

Indeed, when coding a non-functional service within an application—for instance an application-level error handling protocol—using either a procedural or an object-oriented programming language, one is required to decompose the original goal, in this case a certain degree of dependability, into a multiplicity of fragments scattered among a number of procedures or objects.

This happens because those programming languages only provide abstraction and composition mechanisms to cleanly support the functional concerns. In other words, specific non-functional goals, such as high performance, cannot be easily captured into a single unit of functionality among those offered by a procedural or object-oriented language, and must be fragmented and intruded into the available units of functionality. As already observed, this code intrusion is detrimental to maintainability and portability of both functional and non-functional services (the latter called “aspects” in aspect-oriented terminology). Such aspects tend to crosscut the system’s class and module structure rather than staying, well localized, within one of these units of functionality, e.g., a class. This increases the complexity of the resulting systems.

The main idea of aspect-oriented programming is to use:

1. A “conventional” programming language (that is, a procedural, object-oriented, or functional programming language) to code the basic functionality. The resulting program is called a component program. The basic functional units of the component program are called components.
2. A so-called aspect-oriented language to implement given aspects by defining specific interconnections (“aspect programs” in aspect-oriented lingo) among the components in order to address various systemic concerns.
3. An aspect weaver, that takes as input both the aspect and the component programs and produces with those (“weaves”) an output program (“tangled code”) that addresses specific aspects.

The weaver first generates a data flow graph from the component program. In this graph, nodes represent components, and edges represent data flowing from
Context and Trust Based Adaptive Security for Accessing Services in Ubiquitous Computing Environment