LISKOV83
Liskov, Barbara, and Scheifler, Robert; Guardians and Actions: Linguistic Support for Robust, Distributed Programs; ACM TOPLAS 35, 3 (July 1983) pp. 381-404.

Distributed programs are programs in which modules reside and execute at communicating, but (typically) geographically distinct, locations. The authors present an overview of an integrated programming language and system, called ARGUS, that was designed to organize and maintain distributed programs effectively.

ARGUS has two main concepts: *guardians* and *actions*. The ARGUS programming language is built upon the CLU language. An ARGUS distributed program is composed of a group of guardians, each of which encapsulates and controls access to one or more resources (e.g., databases or devices). Guardians provide a mechanism for reliable and internally concurrent modularity. They maintain local control over their local data; no other guardian may access or manipulate the data directly. The guardian provides access to the data through *handler* calls, but the actual access is performed inside the guardian. The guardian is to guard its data in three ways: by synchronizing concurrent access to the data, by requiring that the caller of a handler have the authorization needed to do the access, and by making enough of the data stable so that the guardian as a whole can survive crashes without loss of information. After the crash of a guardian's node, the language support system recreates the guardian with the stable objects from stable storage.

While guardians are the unit of modularity, *actions* are the means by which distributed computation takes place. Actions provide *atomic computations* which are both indivisible and recoverable: they either fully succeed (*commit*) or fully fail (*abort*). When an action aborts, the effect is as if the action had never begun: all modified objects are restored to their previous states. When an action commits, all modified objects take on their new states. A top-level action starts at some guardian. This action can perform a distributed computation by making handler calls to other guardians; those handler calls can make calls to still more guardians; and so on. Since the entire computation is an atomic action, it is guaranteed that the computation is based on a consistent distributed state and that, when the computation finishes, the state is still consistent (assuming in both cases that user programs are correct). Actions may be hierarchically structured and nested to cope with failures and provide concurrency within an action. An action may contain any number of subactions, some of which may be performed sequentially, some concurrently. The authors believe that the single most important application of nested actions is in masking communication failures.

In the authors' model, distributed programs run on nodes connected by a communications network. Each node consists of one or more processors of various kinds, one or more levels of memory, and any number and type of external devices. While neither the network nor the nodes need be reliable, it is assumed that all failures can be detected. The model concentrates on a class of applications concerned with the manipulation of long-lived, on-line data (e.g., airline reservations system or banking system), where real-time constraints are not severe, but reliable, available, distributed data is of prime importance. Such an application domain imposes a number of requirements: (1) continuous service of the system as a whole in the face of node and network failures; (2) dynamic logical and physical reconfiguration; (3) organizational autonomy of control of nodes according to efficiency, political, and sociological considerations; (4) explicit programmer control over the distribution of modules in the system, with changes in the location of modules having limited, localized effects on the actual code; (5) taking advantage of potential concurrency in an application to increase efficiency and decrease response time; (6) maintaining the consistency of the distributed data.

The authors found the last requirement the hardest to meet, because consistency involves both coordination of concurrent activities while avoiding interference and the masking of hardware failures. Atomicity is provided to support consistency in a "modular, reasonably automatic way" (the notion of atomicity has been used extensively in database applications).

Guardians and handlers are an abstraction of the underlying hardware of a distributed system. A guardian is a logical node of the system, and interguardian communication via handlers is an abstraction of the physical network. The most important difference between the logical system and the physical system is reliability, in that the stable state is never lost (to some very high probability) and the at-most-once semantics of handler calls (using remote procedure calls) ensure that the calls either succeed completely or have no effect.

Since synchronization of access to shared objects and recovery are expensive to implement, only special objects called *atomic objects* have these properties. Atomic objects are encapsulated within *atomic abstract data types*, which have operations just like normal data types, except that the operations provide indivisibility and recoverability for the calling actions. A guardian definition implements a special kind of abstract data type whose operations are handlers. *Creators* may be invoked to create new guardian instances dynamically. Guardians may be parameterized. The definition also includes the specification of stable and volatile variables, a recovery section to recreate the guardian's volatile state after a crash (e.g., creating a database index), and a background section, through which periodic or continuous tasks in the guardian may be performed.

ARGUS provides separate compilation of modules with complete type checking at compile time. Guardians and handlers can be used as arguments in local procedure calls and in handler calls. Compile-time checking does not rule out dynamic reconfiguration; a distributed catalog registers guardians and handlers for binding. The authors provide a somewhat cryptic example of a simple mail system to show how the ARGUS language features could be used.

The authors argue that ARGUS differs from other languages that address concurrent or distributed programs. While those languages provide "modules that bear a superficial resemblance to guardians" and some form of intermodule communication based on message passing, the modules have no internal concurrency and contain no provision for data consistency or resiliency, completely ignoring the problem of hardware failures. Arguably, ARGUS supports well consistency, service, distribution, concurrency, and extensibility. Two areas are not, however, well supported: protection and scheduling. There is no way within the language to express constraints as to where and when guardians may be created or whether guardian calls to handlers are legal. As well, there is no direct support for scheduling incoming calls and providing handler call priorities across an entire node. Only a preliminary, centralized implementation of the language had been completed when this paper was published, although a "real, distributed implementation" was underway. No information on the efficiency of the existing implementation was given, and the authors do not discuss approaches to supporting guardians and actions. The paper suffers from not having enough implementation information, although presumably the excellent reference list covers this problem.

The authors argue that, regardless of advances in hardware which may render software compensation for hardware failures obsolete, atomic actions are necessary and are a natural model for a large class of applications.

This paper provides a new and interesting approach to the problem of supporting in a programming language robust, distributed programming. While the concepts are interesting and worthy of examination, one awaits the results of the experimental validation of the guardian and action approach in a true distributed environment.