Implementing Shared Data Objects Above the Kernel

Jeff Chase, Sabine Habert, and Hank Levy
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Commonly-cited goals for kernel-supported object abstractions are: (1) a uniform view of system resources, and (2) a common foundation for sharing and/or storage of application data. It is tempting to guarantee a uniform view of all system entities by defining a single object mechanism encompassing both data and resources; capability-based addressing is the archetype for this approach. This idea is flawed because data and resource sharing involve separate categories of objects with fundamentally different purposes, properties, and implementation needs.

Resource objects are abstractions for protected entities such as devices, files, processes, and services. *Data objects* are units of data sharing between applications; they are fine-grained entities whose meaning is determined at the language level. The essential difference is that language-level encapsulation (by name scope rules and strong type checking) is sufficient for data objects, but resource objects require individual access controls and encapsulation by hardware-enforced addressing boundaries. Any common implementation will necessarily be inappropriate for one category or the other: fine-grained access control and hardware encapsulation are excessively costly for data objects, and reliance on language protection for resource objects assumes a single-language environment, with no malicious interference.

To support shared objects efficiently, we must separate the implementations of data objects and resource objects. Data objects should be implemented using lightweight mechanisms in the language processor and runtime system, layered above conventional kernel abstractions of untyped storage and communication. They need not be implemented in the kernel simply because they are used as a basis for sharing; if the right model is supported, the application writer does not care who supports it. However, the kernel must support resource objects directly in some way (e.g., protected ports), because they require absolute protection that only the kernel can provide. The programmer's uniform view of all objects in the system can be preserved by "sugaring over" the implementation differences at the programming language level.

There are three fundamental reasons for rejecting kernel-supported object abstractions as a basis for data sharing:

- **Performance.** Implementing data objects above the kernel reduces the need to interact with the kernel.
- **Flexibility.** Placing data object support in the kernel limits the system to a particular set of management policies. This is too restrictive for data objects, which can benefit from a variety of caching and migration policies.
- **Language-Independence.** Some aspects of data object implementation are language-specific, or are simplified by knowledge of language semantics. Placing support in the kernel

limits opportunities to benefit from this knowledge, unless the system is strongly coupled to a single programming language.

Shared data objects should be implemented using the techniques of persistent object stores and object-oriented databases. A group of related data objects can be clustered within some *container*, such as a file or a mappable memory segment backed by a paging server. From the point of view of the kernel, the container is an untyped storage resource; its internal structure is known only to the clients that bind to it. Once a client is bound to the storage area, the data objects it contains are installed in the client's context and accessed directly with ordinary procedure calls, relying on the programming language to enforce encapsulation and type-safe usage.

Overhead is reduced by treating the container as a unit to amortize the cost of expensive operations. For example, access checks are made only at bind time; once a client has been granted access, all objects in the container are visible to it. The clustering is also useful to decentralize name assignment; if object names are allocated in groups to containers, the names for objects created within a container can then be assigned independently by the client. Smaller clustering units within the container may be useful to reduce the overhead of I/O or synchronization.

Allowing multiple user-level implementations of shared data objects can compromise the programmer's uniform view of objects. It can also complicate interoperability, since the different flavors of objects might not share the same name space or even a common set of invocation behaviors. Our position is that *uniformity is a language issue.* A uniform object model does not imply a uniform underlying implementation; it can be achieved at the language level with the help of additional runtime support, despite diverse implementations.