Contact Author: Burt Rosenberg, burt@cs.miami.edu
Title: Small Shared Objects
Tracks: brief announcement
Student paper: no
Notes: burt-54421.ps submitted Fri Jan 17 1997

Small Shared Objects

Burton Rosenberg
Assistant Professor of Mathematics
and Computer Science
University of Miami

Introduction

This paper describes an architecture for distributed objects based on shared memory. We argue that a natural synergism exists between objects and shared memory, and exploit this in a shared object layer. This layer decouples the concerns of shared objects with the concerns of object communication, giving an extendible subsystem for shared objects. The resulting architecture allows for a location and communication indifferent context-centered viewpoint on object distribution.

The object paradigm has many components. Most often noted are: a poset class structure; a strong type system whose limitations are remediated by method overloading; a system of data privacy which follows the class structure; the separate name spaces for each object. In addition, an object is a context closure. An object carries with it a self contained context which demonstrates individuality, permanence and self-sufficiency. This is a natural leverage point for the introduction of distributed processing via shared memory. Our small shared objects enclose an element of shared memory, their internal data invariants include a consistency constraint across copies of the shared memory. Our algorithms to enforce consistency are tuned for "small objects", such as integers. Larger objects, such as an unstructured byte arrays or unlimited length strings, are included in our model but not our current implementation.

We have a three component structure: a single underlying communication substrate, separately implementable shared objects, and a *seance manager* which negotiates the names and connections of shared objects across platforms. The actual shared objects are collected into a virtualized device, called the *seance*, to which local implementations, called *manifestations*, can connect.

Shared Memory Algorithm

We describe our shared memory algorithm for a variable shared by two objects, each called a *manifestation.* We will call the manifestations *peers.* Our algorithm simulates the MESI protocol, where a change in the cached copy of a variable invalidates the cached value in the peer. The sharing mechanism is visualized by associating two tokens with the shared variable.

A manifestation seeking to write to the shared variable must possess both tokens. A manifestation requests tokens from the peer via commands *strike, share* and *surrender.* To ask a manifestation to give up one of its two tokens is called a *share,* and the response includes the token and the current value. To ask a manifestation to give up its one remaining token is called a *surrender.* We allow for the case where a manifestation will overwrite without caution a shared variable, and the manifestation is asked to immediately give up the two tokens it holds. This is called a *strike* and the response does not contain the variable's value. Once a manifestation has both tokens, it writes or read-modify-writes the variable.

A manifestation seeking to read the shared variable must possess at least one token. It the manifestation has no tokens, it requests a share from its peer, which necessarily holds both tokens.

A live-lock situation exists where both manifestations simultaneously request a surrender from each other. The situation is handled using a flag *yield.* The two yields, one for each of the two manifestations, are never simultaneously set. In the case of a collision of surrender requests, the manifestation for which the yield is true will return the token, and the other manifestation will reject the surrender request. Manifestations alternate the yielding by setting its yield true when it gets two tokens, and false when it has zero tokens. Initially, an external mechanism must break the symmetry.

The user actions upon a shared variable are *get, set, ready-modify-write* and *sleep-until-change.* To act on a shared variable is to make the request to the manifestation which is local to the requester. That manifestation may have to communicate with the peer to gain tokens and current values. A *busy* flag in the manifestation is set while the object waits for the peer's reply. While busy, a manifestation will generally reject incoming strike, share and surrender requests, with the exception of an incoming surrender when yield is set, as described in the previous paragraph.

Implementation

We have implemented in Java a model, two-component shared memory system using the algorithm described in the previous section. The client connects to a seance server and requests a remote fork of an object specified by class name. The client and the remote object then begin instantiating shared objects, connecting them in the same order to the seance by providing them with a reference to the open communication channel. Shared objects on either end of the channel are identified by the order in which the are added to the seance. On instantiation, the shared objects agree on their initial values, and are endowed with one token. Hence objects which are shared but are used in read-only mode cause no network traffic.

To support the algorithm, the communication channel must allow for symmetric, peer-to-peer communication. The standard client-server discipline, based on TCP/IP sockets using blocking I/O, is inadequate. We finessed two-way communication by starting a thread to listen to the channel and dispatch commands using upcalls to the shared objects. The shared objects must return immediately, since blocking on that upcall thread stalls the seance system. Calls into the channel, e.g. a shared object requesting an exclusive lock on a variable, block the caller's thread until a response is returned across the channel. The listening thread will transform the response into a wakeup for the blocked thread.

Other shared objects are implemented by extending the abstract base class for a shared object, which defines the interface between the object and the communication substrate. The implementor of a shared object must take some care to avoid stalling the channel and maintain thread safety. We concede that programming a shared object is subtle. However, their use is quite simple.

Conclusion

We have described a shared object implementation and a seance as a shared context, composed of shared objects. Our implementation is but a sketch. Larger objects, multiparty sharing, and a formal description of the seance manager are needed. Furthermore, our ad hoc communication channel should be replaced where possible with standard Remote Method Invocations and object serialization. We are currently working towards these goals. The implementation is available via http from mirror.cs.miami.edu.