Requirements for the Igor Program Database

Robert A. MacLachlan (ram@cs.cmu.edu)
Carnegie Mellon University

1 Introduction

The Igor project is a new project at CMU to develop an innovative programming environment that supports programmers using multiple networked workstations by integrating the implementation of an OO language (Dylan) with with a programmer interface that provides find-grained revision control, configuration management, exploratory browsing and whole-program compilation.

The Igor environment is oriented toward supporting evolutionary refinement, reuse and maintenance of programs. During development, coding, design and documentation can proceed concurrently, with the environment maintaining the association between code, documentation, tests, human and compiler annotations, etc. Interactive semantically-based browsing and searching replace "grep" or cross-reference listing.

In order to support this "hypercode" model of software development, we plan to make the program permanently reside in a persistent object database. Some special locality properties of configuration management and typical work patterns can be exploited to minimize the cost of maintaining consistency in the distributed system.

Although our main concern is to meet the programming environment's need for responsive persistent object management, the environment is to be written in Dylan, so we will probably end up implementing a fairly general persistent extension to Dylan. The resulting system should be usable for many design databases other than for software. I suspect that once we get used to persistent objects, we'll use them for everything.

2 Architecture

project A *project* can be equated with the central database which contains the current checked-in configurations of the all the code, documentation and everything else that we want to know about a particular system. In addition, all of the previous configurations are also recorded for future reference.

workspace The *workspace* is a persistent version of the database representing a given programmer's current stage of development. Objects can be freely modified, but the project database is only updated explicitly. Once changes have been flushed, the workspace can be destroyed. Workspace creation must be fairly fast; presumably a copy-on-write strategy would be used.

target process A *target process* is an actual process that can be run, debugged, etc. A target process is initialized from a workspace, probably by starting with a runable version of the the project with any subsequent modifications added in.

3 Database Requirements

— Object references must be efficiently stored in the database for cross-referencing and dependency recording.

— Must be reliable, support distributed authenticated access, and be reasonably scalable.

— It should be possible to condense a project into a *release* which can be exported to other sites and referenced copy-on-write by other projects.

— Good integration with Dylan; some (probably explicitly specified) Dylan objects would be persistent, and this would be the primary way of accessing the database. Schema evolution would be nice.

— Some GC capability; at a minimum, pointers in the database must be locatable for offline GC.

— Versioned, supporting compaction or automatic regeneration of old old versions.

— Must have quick interactive response time for simple read requests, and good bandwidth for batch read/write or prefetching. Grouping hints would be nice.

4 Problem Properties

— The database doesn't change all that rapidly, and the writes that do happen (due to compilation or check-in) can be explicitly grouped.

— Since access is read-intensive and fairly predictable (at least to the application), prefetching is likely to be effective. For example, the compiler can prefetch symbols that it referenced on the previous compilation of a given procedure, and browsers can prefetch adjacent objects or entire persistent collections.

— With typical work patterns, actual write contention is low, since people usually aren't modifying the same object simultaneously. Contention is reduced compared to a file-oriented system, since the granularity is finer (e.g. a single procedure.)

— If necessary, concurrent writes can be resolved by a human merge.

5 Implementation Ideas

It looks like we're going to have to implement our own storage manager, since existing object database and distributed object systems are either too inefficient, too inflexible, not yet available, or (often) all three. We will probably try to find and integrate a uniprocessor persistent store manager, an IPC package, and authentication and name services.

A central server will hold the project database, and each programmer's workstation will run a server to manage his active workspaces. Workspace servers will cache copies of objects. These copies can be decached via callbacks on modification, but it may also be possible to explicitly maintain consistency via a snapshot–diverge–merge style of configuration management.

The various client applications (the compiler, etc.) will also cache objects in their address spaces as Dylan objects. These client copies will be garbage collected when unreferenced. Saving of updates is done in software; the programmer can choose various strategies on a per-class basis. Some objects might want to automatically be added to an update queue when a slot is modified, while others might require explicit flushing.

6 Distribution

The two main efficiency concerns with the implementation strategy are:

— If doing reads usually required round-trip IPCs with either server, data-structure traversal would to too slow.

— The central server could prevent scaling by becoming a bottleneck.

To minimize read overhead, we exploit temporal and spatial locality using caching and prefetching on both the client–workspace and workspace–project channels.

The hope is that the number of central writes and the need to propagate those writes to other workspaces will be small enough so that the central server won't be a bottleneck. The project server has little information about activity in workspaces: where to find each workspace, what dirty project objects it owns, and which modification callbacks it has registered. A time-indexed change log allows workspaces to update their caches with a single query.

Several programmers can work on the same problem while communicating in real-time as long as they are sharing the same workspace; we can afford to have workspace servers tightly synchronized with a few clients.

7 To GC or not to GC

It is an open question whether this distributed persistent store should (or can) be garbage collected. My current inclination is to allow a choice on a per-class basis of explicit deallocation or reference counting, combined with an off-line garbage collector/database fixer. Management of reference-counted objects can be made somewhat automatic by using finalization in the per-process garbage collector to decrement reference counts.

There won't be a great deal of persistent garbage creation in the project database, since its purpose is a permanent record of all modifications. The database associated with a workspace may have a greater need for GC, since objects representing the state of the user interface (open windows, etc.) may be created and destroyed at a moderately high rate.