Improving Rapid Application Development Environments Through Coordination

Nick Carriero, David Gelernter, and Martin Schultz

Department of Computer Science
Yale University
New Haven, CT 06520

and

Scientific Computing Associates Inc.

Rapid application development environments (RADs) have emerged as one of the major success stories of research in programming languages and systems. Systems such as Mathematica, Matlab, Perl, Python, R, S+, and Tcl/TK have become the software development environment of choice for researchers across a wide variety of disciplines. These environments have become increasingly sophisticated, reaching the point in some cases of providing a full-fledged virtual OS experience. That is, once a user enters the RAD, there is little reason to leave.

Performance, however, is one all-too-common reason why a user might do just that. Adding a coordination facility to a RAD can help address performance issues in a variety of ways:

- RADs traditionally offer an extension mechanism that can be used to address performance limitations. A coordination facility can be used to simplify, enhance, and augment this mechanism.
- While extension mechanisms of this kind might lead to a level of performance comparable to compiled code, a coordination facility opens the door to parallel execution, offering a level of performance far-beyond compiled, sequential code.

In this paper, we discuss the design and implementation of such a coordination facility. The application of such a facility to specific RADs such as python, R, and Matlab will be discussed. In addition, we will show how such a coordination facility can be used to development high level distributed/parallel applications.

It is important to note that users of RADs have already recognized the value of high-level problem solving tools and are therefore unlikely to find low-level coordination tools attractive. Nor will low-level coordination tools be a good match to the flexibility and conceptual richness of the hosting RAD environments. A high level "rapid coordination development environment” (RCD) provides a suitable complement to a RAD.

The ideal RCD would support coordination at a level matching the semantic richness of the base RAD. The RCD’s base data types would be consonant with the base data types of the RAD. It would offer a conceptual framework that inspires the construction of flexible coordination "harnesses.” Thus, a RAD+RCD could address performance concerns, but the whole would be greater than the sum of its parts, allowing for the application of RAD systems to new problem areas. We discuss the design and Implementation of a RCD based on a tuplespace (virtual shared memory) approach. Such an RCD is an excellent fit to these RCD requirements.

A tuplespace-based RCD’s object store interface easily accommodates "native" base types, and permits coordination harnesses to be built from distributed data structures. Distributed data structures are variants of traditional data structures that admit of concurrent manipulation. As such, they represent a natural, and powerful, generalization of the basic paradigm of sequential RAD coding (and indeed, virtually all programming) -- the algorithms as an orderly sequence of manipulations of data structures -- to concurrent, distributed and parallel settings.

In addition, the uncoupled nature of a tuplespace approach encourages a flexible and dynamic style of ensemble building that is well suited to the general spirit of RADs, which are often used to implement components in "work flow" graphs. tuplespaces are uncoupled in both space and time. Data may be stored into a tuplespace by one process and later consumed by another process whose lifetimes do not overlap. Processes that produce and consume data objects need have no topological ("spatial") relationship. Once a tuplespace host is established, the format and content of the data objects stored implicitly defines the means of reference. No process ids, names of component hosts, machine addressing schemes, or the like have to be agreed upon in advance or kept up-to-date at runtime

Key benefits of RCDs:

1) One of the RAD’s most attractive features is the vastly accelerated software development and refinement cycle. When performance demands lead to abandonment of the RAD, the development and refinement cycle slows to a crawl, and in some cases applications become, effectively, living fossils. By enabling RADs to be performance effective in a wider domain, RCDs may enable some problems to live entirely within the RAD environment, while postponing the fossilization stage of others enabling the realization of substantial additional benefits from the RAD cycle. This benefit applies both in the algorithmic complexity and problem size dimensions.

2) RADs encourage efficient exploration of the solution space for a given problem. An RCD will expand the solution space that can be efficiently explored.

3) By providing a common RCD to complement RADs, the boundaries between RADs may fall away, allowing complementary RAD environments to be integrated to solve complex problems, drawing on the unique strengths of each.