Pervasive Programs

Steven P. Reiss

Department of Computer Science
Brown University
Providence, RI 02912
spr@cs.brown.edu, 401-863-7641, FAX: 401-863-6757

1. Introduction

We envision a time in the not too distant future where all applications are handled by a single program that runs over all machines simultaneously. Such a program would be continually evolving, growing, and adapting. It would be the logical extension of today's combination of Internet technology, pervasive computing, grid computing, open source, and increased computational power. Moreover, it would significantly change the way we think about and do programming.

Programs have traditionally been stand-alone entities. From back in the day of card decks, programs were things that had a definite starting time and stopping time. They were run by individual users to accomplish some task and then they were exited. Sharing among users was done in controlled ways, first through files, later through databases. Even long-running programs such as operating systems or database services are thought of as individual entities, running on their own machine (or small set of machines), that happen to provide services to other programs.

New and current technology let us think of programs in a different way. Rather than everyone running their own program, we can think, at the extreme, of there being only one program. This program would run on all computers simultaneously. Users wanting to access information or do some computation would simply provide this program with the appropriate input and get their output. Programmers, instead of developing new programs, would develop new modules that are plugged dynamically into the single running program. New computers, as they come on line, would start running part of this single program, and would be able to take advantage of the existing computing resources while providing the program with additional capabilities.

There are obvious advantages to this view of programming. First, this is a logical extension of the trend to network-based applications and modules. Microsoft's .Net framework, for instance, lets programs invoke network services almost as easily as calling internal routines. Cooperating network services, each with shared state and running on different machines, are a form of a pervasive program. Second, as pervasive computing becomes more prevalent and everything in a house is essentially a computer on a network, viewing the various entities as a single cooperating program provides a framework for controlling and managing the multiplicity of devices. Third, as computers become more sophisticated, users expect to get more out of them. There are only a limited number of 50 million line programs that can be written effectively. If all programmers actually contributed to the capabilities of a single system and each programmer could make use of the efforts done by others, then it would be possible to create the types of systems that users will expect in the future. Fourth, a single program provides a logical means of sharing the resources of large numbers of computers, effectively making grid computing available to all. Finally, interprogram communication right now is one of the more complex parts of system design and building. Addressing large-scale challenges such as controlling automobiles on a computerized roadway requires that large numbers of systems (in this case all the automobiles on or near the roadway) effectively communicate. If we change the way we think of programs so that such communication is inherent rather than an addendum, we should be able to make developing such systems easier.

2. Research Issues

At the same time, there are many technical and social problems that would have to be overcome to making such a view of programming a reality. However, we feel that solutions to these problems are possible.

Any implementation of pervasive programming must support portions of the program becoming unavailable and being replaced while the program is running, as well as new portions of the program being defined dynamically. There have been a number of environments and systems that support dynamic code replacement and others that have had to deal with unreliable networks in distributed systems so, in principle, these problems are tractable. However, doing it on a large scale, in a distributed rather than centralized manner, in a way that cannot afford failure, and where code can come and go dynamically, will require additional research and probably a new framework to manage and track the code base. Dynamic techniques for discovering implementations such as those used by Jini, will also be necessary here.

Pervasive programming is going to require extensions to current languages if not a totally new programming language since it implies a new way of thinking about programs. One approach that seems feasible is to introduce an extended notion of an interface, which we will call an *outerface* for now, as the basis for the new functionality. Outerfaces would be defined as abstract classes with both virtual and static methods. Users would code their portions of the system to implement a particular outerface while at the same time using already existing outerfaces. The underlying system would then be responsible for choosing an appropriate implementation of an outerface to be used by another outerface at execution time. This would be done based on parameters the user might provide, properties of the outerface implementations, analysis of the uses of an outerface, network proximity and availability, as well as other conditions. There are numerous language issues that arise in attempting to make such a notion both practical and implementable. Moreover, the language must provide the hooks to deal with outerface implementations that disappear or are replaced dynamically.

In this view of pervasive programming, outerfaces are the key building block. To support them, there has to be standard ways of defining, finding, and categorizing outerfaces and their implementations. We imagine that this could be done at various levels each of which imply a different level of authority and responsibility. First, we imagine that there would be a well-defined set of standard outerfaces that would be adopted by some sort of standards committee. This would be akin to the methodology currently used in defining extensions to the Java language. Second, we would allow user groups to propose and code to shared outerfaces that might not be in the standard but can otherwise be agreed upon. Finally, it should be possible for individual users to create their own outerfaces and corresponding implementations, keeping the scope private where desired.

Another problem involves identifying the semantics of an outerface. If outerfaces are going to be widely used, each must have well-defined and well-understood semantics. Without a long and drawn out standards process, this is difficult and impractical. Instead, we envision that an outerface would be defined both as a set of methods and a set of test cases. Any implementation proposed for the outerface would have to pass all the test cases in order to be considered valid. Users would then be able to define their own subouterfaces that add additional test cases that might be of particular interest or importance. The underlying system would take care of ensuring that only implementations that pass the test suite are acceptable. This operational definition of semantics seems a practical and feasible approach.

In addition to dealing with the immediate technical issues, a pervasive programming world would have to deal at all levels with security and privacy concerns. These will have to be addressed directly at the language level since they will be properties of outerfaces that implementations will have to conform to and that users will insist on. A capability-based model at the language level could be a basis for a solution in this dimension.

Steven Reiss - Bio

Steven Reiss is Professor of Computer Science and Associate Chairman of the Computer Science Department at Brown University. He has been a member of the Brown faculty since 1977. He received his A.B. from Dartmouth College in 1972 and his Ph.D. from Yale University in 1977. He was named an IBM Research Scholar from 1987-1989. He is the author of numerous papers in journals and proceedings and has served on a variety of conference program committees for SIGPLAN, SIGSOFT, IEEE, and USENIX. He has written several software systems that have been widely distributed outside of Brown.

Dr. Reiss's research interests and expertise lie in the area of programming environments and software visualization. His previous programming environments include PECAN, GARDEN, FIELD, and DESERT. FIELD pioneered the concept of message-based (control) integration and illustrated a variety of program visualizations. It uses a central message server and selective broadcasting to combine existing UNIX programming tools into a common environment, and provides graphical visualizations of program structure, dynamics and performance. The concepts used here have been copied in HP's Softbench and Sun's Tooltalk and were licensed for DEC's FUSE environment. Moreover, they form the foundation for more advanced integration methods such as those incorporated into CORBA and OLE. The most recent environment, DESERT, combines an inexpensive data integration mechanism with a common editor and a collection of design and programming tools.

Current research being undertaken by Dr. Reiss includes work in the area of the visualization and analysis of the dynamics of complex software systems, software environments that let code and design evolve consistently and in parallel, web-based information retrieval, and developing software for a wireless classroom.