Guest Editorial

Eugene Spafford Purdue University

"Distributed systems (loosely-coupled) are simply multiprocessors with greater latency than parallel (tightly-coupled) systems." Consider that statement carefully before you read further. How strongly do you agree or disagree?

During the last Symposium on Experiences with Distributed and Multiprocessor Systems (March 1991), I presented that notion to a number of participants. Most thought about it a moment and then agreed. Since that time, I've proposed it to a number of other computing professionals, and many concur that it represents distributed systems as they know them.

It is, however, false.

I believe that the reasons some researchers view it as true underlie why there has been more progress made in parallel computing than in distributed computing over the past few years.

Certainly, there is some truth in the statement. Distributed systems present some problems of task partitioning, multiprocessor synchronization, appropriate choice of data structures, communication, and user interface that are similar to parallel systems—except for latency issues. They are not simple problems: for instance, knowing how to partition a job into appropriate-sized and sequenced pieces is a difficult task, whether those parts are to be run on processors sharing the same bus and memory, or whether they are to be run on processors connected by T3 links spanning a continent.

However, the two kinds of computing systems are most definitely not the same in practice. There are many practical issues that make distributed systems more difficult to administer and use—whether they are distributed throughout a building or across continents.

For instance, consider the problems of authentication. Tightly-coupled parallel systems are always housed in the same room, if not within the same cabinet, and the processors usually do not have completely autonomous modes of operation. Distributed systems, however, often have components residing far away from their peers; physical security and proximity cannot be used to verify identity and authenticity of messages in distributed systems.

As another example, consider fault tolerance. Keeping single copies of critical data may cause single-point failures to cripple hundreds of machines, but keeping multiple copies consistent is difficult to do efficiently. On a tightly-coupled system, failure of one component can often be compensated for without undue difficulty, or may result in a complete, observable failure of the system. In a loosely-coupled distributed system, a component failure, such as a network partition, may not be distinguishable as a failure. The result may be inconsistent data, cascading failures, and eventual chaos.

Consider tasks of simple system administration. On a tightly-coupled system, installing a new software release or adding a new user account requires modifying files at a single location. In a distributed environment, it potentially involves updating all the locations where the system connects—and perhaps doing so synchronously and with extended privilege. Of course, it is possible to devise a protocol for managing administration and distribution tasks from a few centrally-located sites (e.g., NFS and NIS), but this creates yet a new potential avenue for compromise of system security as well as introducing single points of failure.

There are other problems, too, but they usually become apparent once you begin to think about how you would actually administer and use a system on a day-to-day basis—assuming you have some experience to guide your imagination. It requires something other than a simplified theoretical view where machines never fail, users are trustworthy, and software never changes.

The low-level technical tasks of distributed programming and efficient communication are difficult, but even if they are solved it does not mean that the resulting systems will be usable. To achieve the potential benefits of distributed processing—greater computational power, failure isolation, load balancing, specialized resources, etc.—will require devising answers to some seemingly more mundane but critical questions.

These are not new observations. In the late 1970s, Philip Enslow at Georgia Tech described something he called "Fully Distributed Processing Systems" (cf. "What is a 'Distributed' Processing System?," *IEEE Computer*, 11(1), Jan. 1978, 13-21). He defined these as systems that had multiple computers with distinct capabilities, no shared data (memory), and autonomous individual control. If we understand that "control" includes administrative control as well as low-level device and processor control, we have a definition that is still interesting—and unmet by any existing system, although some have attempted extremes of a pair of these properties.

I too see a major challenge to researchers in distributed systems is to design their systems explicitly—from the very beginning—to near extremes of all three properties. Adding network layers to single-system operating systems, as many efforts have done in the recent past, seems unlikely to produce the kind of system many of us will find trustworthy, easy to use, and easy to administer. Neither does simply structuring a system around micro-kernels, threads, or objects automatically achieve such a goal.

I find it interesting to note the number of bright and informed people who think that solving the challenge of multicomputer systems requires only the solution of communication and memory questions. Perhaps that is why we continue to have so many machines with multiple processors in the same box on the same bus, and why we continue to have such problems managing systems that are geographically separated (e.g., workstations on LANs). This is why experiences with multiprocessor systems are so important—they illustrate areas of difficulty that conceptual models and designs might never consider.

In this special issue of *Computing Systems* we present the revisions or extended versions of five papers drawn from SEDMS II, held in Atlanta in March of 1991. As with all the papers at a SEDMS symposium, each of these represents real experiences with innovative approaches to solving problems in multiprocessor systems. These articles, however, are not merely reprints of SEDMS presentations. The authors involved rewrote their papers, and the papers underwent additional review; not every article invited for this issue was accepted. The editors, authors, reviewers, and I all hope you find the results of our efforts rewarding and enlightening.

My special thanks to the referees who read papers submitted for this special issue, and whose thoughtful comments aided both the au-

thors and me in our efforts: William Appelbe, Bharat Bhargava, Kenneth Birman, Anita Borg, Roger C. Camp, Roy H. Campbell, David Cheriton, David L. Cohn, Raphael A. Finkel, Rob Fowler, James Griffioen, J. Robert Horgan, Christopher Kent, Yousef Khalidi, John T. Korb, Richard LeBlanc, George Leach, Darrell D. E. Long, James E. Lumpp, Charles McDowell, Trevor N. Mudge, Bodhi Mukherjee, John Nicol, Michael O'Dell, David Patterson, Sharon Perl, David V. Pitts, Daniel A. Reed, Michael L. Scott, Satish K. Tripathi, and Jonathan Walpole.