Association for Computing Machinery
Advancing Computing as a Science & Profession

Ubiquity, an ACM publication
April, 2012

Bringing Architecture Back to Computing

An Interview with Daniel A. Menascé

by Peter J. Denning, Ubiquity

Editor's Introduction

Over the past 10 or 20 years, the subject of machine organization and system architecture has been deemphasized in favor of the powerful abstractions that support computational thinking. We have grown accustomed to slogans like "computing is bits, not atoms"—suggesting that bits are not physical and the properties of the physical world are less and less important for understanding computation.

Daniel Menascé, a Professor of Computer Science and Senior Associate Dean at George Mason University's Volgenau School of Engineering, is alarmed at this trend. He has extensive experience with designing and deploying networks of distributed services. Menascé claims all computing takes place in the physical world and is constrained by physical limitations of components. He believes without an appreciation for system architecture and the limitations of physical components, future computing people will be unable to design safe, secure, usable, and reliable systems. He calls for all computing professionals to pay attention to the architecture and for educators to restore balance in their teaching of architecture.

Peter J. Denning
Editor

Bringing Architecture Back to Computing

An Interview with Daniel A. Menascé
by Peter J. Denning, Ubiquity

Ubiquity: You have been involved with building computing systems and network services for some time, and teaching your students how to do this well. Please tell us about your background and how you came to these interests.

Daniel Menascé: It started in 1970 with my freshman college introduction to programming course. I loved programming but by today's standards it was an exasperating process. We coded our programs using punched cards, submitted them for processing by a mainframe computer, and then waited about 24 hours for the resulting printout. I learned to be careful about my code, both syntax and logic, because any silly mistake would generate huge delays. It became very important to me to pay attention to design before jumping to implementation.

As a teenager, I had taught myself the principles of radio and television and learned how to fix TVs. So, as an incoming freshman, I was determined to specialize in telecommunications. But that introductory course forever changed me. The world of computers was utterly fascinating. During my college years I started to buy and study by myself all the classic books I could get my hands on. I always wanted to know how things worked. Two books made a big impression on me at the time: *Operating Systems Principles* by Ed Coffman and Peter Denning, and *Compiler Construction for Digital Computers* by David Gries.

Ubiquity: I have heard of those.

DM: While working on my MS degree, I joined a team tasked with developing the operating system and basic software of a new mini-computer. I wrote the assembler and link editor for that machine.

I then moved on to UCLA to pursue my Ph.D. I followed my teenage interests by moving into computer communication networks. I was fascinated with the way queuing theory could predict packet delays and bottlenecks of these networks. For my doctoral dissertation, I studied synchronization in distributed databases. After obtaining my degree in 1978, I joined academia.

In the early '80s, I started a small company to develop an industrial-strength database management system for microcomputers. I still remember fondly the challenges of writing that code for an MS-DOS machine with 64K bytes of main memory. That successful experience taught me how to write programs for environments with limited resources. While today's personal computers are orders of magnitude more powerful than those of decades ago, we find similar situations today when programming sensors or devices with low battery, computing, and memory capabilities.

Through my company, I went on to design and direct the development of information systems for the Brazilian Oil Company and for the Brazilian Telecommunications Company. Later on I was involved in the design and supervision of several Web-based applications to support online education and manage the engineering school at George Mason. These experiences helped to improve my teaching and research.

Ubiquity: You have a particular interest in performance evaluation of systems. What have you done in this area?

DM: Yes, my interest was kindled when I studied queuing theory with Len Kleinrock and Dick Muntz at UCLA. I became very interested in using this theory to study the performance of a variety of computer systems. I developed the first analytic models for concurrency control mechanisms in database systems. Later on, jointly with Virgilio Almeida, I developed several exact and approximate analytic models of multiprogrammed computer systems, which were incorporated into a capacity-planning tool that I developed and commercialized. I then went on to develop predictive analytic performance models for many domains including supercomputer architectures, parallel processing, local area networks, mass storage systems, e-commerce systems, software systems, adaptive systems, security mechanisms, grid-computing, and service-oriented architectures. My books and their accompanying tools describe capacity planning models and teach how to size e-commerce and Web-based systems.

My philosophy has always been performance evaluation should not be an after-thought; it should be part of the design process. The computing systems world is filled with horror stories about large systems that were delivered late, over budget, and could not meet their performance requirements.

Ubiquity: I thought that correctness is the main issue with programming large systems.

DM: It is important, but certainly not the only important issue. When you focus only on programming, you concentrate on making sure that the program's function is the one specified. But in real systems, people are concerned with not only the function being correct, but with how long it takes to get the results. The standard algorithmic measures—such as order n squared for input of size *n*—fail to capture the behavior of real applications in execution.

Ubiquity: Those standard measures are meticulously prepared and rigorously proved. How can you say they fail to predict the behavior of programs in execution?

DM: The failure comes from a mismatch between the assumptions of the analysis and the way programs use real systems. Algorithmic complexity assumes programs run alone in a large memory sufficient to hold all their code and data. This is quite useful when comparing competing algorithms to solve a given problem. However, real systems are networked and used simultaneously by many users. Your response time depends not only on how many CPU cycles your job needs, but on the amount of congestion your job experiences as it flows though the network and queues up for various resources.

A really simple example is your desktop computer. You think you are the only user. But if you look at your system's process list, you will see dozens of background processes that supply services. The program you run in the foreground competes with all these services. They take CPU time, which is then not available for your program. It gets even more complicated when you realize that your main program and all the service processes are competing for other resources besides CPU, most commonly network ports and disks.

Networks get even more complicated. Your website is used by hundreds or thousands of people at the same time. How do you tell users what their response time will be, when you don't know for sure how many there are and how they are using all the resources in your system? How do you deal with all the randomness?

The designers of these systems have no choice but to make capacity planning a critical ingredient. Google aims to provide a very low response time, around 0.4 seconds, to any query. How do they calculate how many disks and CPUs they need in each data center, and how much bandwidth is needed in each data center and among them? If they didn't do this well, they would have been out of business a long time ago.

Ubiquity: I though issues like that are part of software engineering, not programming.

DM: There is no way to separate them. Every programmer can expect his or her programs to run on a distributed network with hundreds or thousands of competitors for the servers and disks on the network.

Customer load testing is the closest that many programmers get to analyzing how their programs will perform in a network. They build the system and then observe how it performs for real customers. They often release new systems for customer use and, in effect, let the customers do the testing. They then try to resolve bottlenecks that customers complain about. Many startups have gone out of business by following this approach.

This approach is equivalent to a civil engineer building a bridge without a careful analysis of the loads it must sustain. No one would allow a bridge to be put to its first test when the traffic starts to flow. It is beyond me that we do the same for computing systems, when so much of our well-being depends on these systems.

Part of software engineering tradition is to pay attention to "functional" requirements and to downplay "nonfunctional" requirements. Functional requirements specify the outputs that each function of the system is to produce. Nonfunctional requirements include performance, reliability, and security. Success can only come when all the requirements are met—that is, when performance, reliability, and security are treated as part of the function.

I have seen numerous instances of programmers testing their SQL queries against a small test database to satisfy themselves that the correct results are being returned. The same queries that take a few seconds to run in the test database could take hours to execute in a production large database.

Ubiquity: You recently told me you think the current generation of students has lost its appreciations for the workings and limitations of computing architectures. What is the problem you are seeing?

DM: Let me go back to that first course in programming I took as a freshman. We were taught about computer organization and were introduced to a hypothetical computer à la Knuth's MIX computer before learning how to program in FORTRAN. We wrote programs using the assembly language of this hypothetical machine and ran them on its simulator. We did not move on to write programs in FORTRAN until we understood the basics of computer organization, instruction set use, memory allocation, and input-output. It was then easy to understand the role of a compiler to translate a high-level language program into machine executable code.

That understanding also gave us students an appreciation for program efficiency based on the understanding of the limitations of the machine it is supposed to run on.

Knuth showed how complexity measures of program running time were simply the instruction counts of the simple MIX machine. They translated directly into running times of FORTRAN programs.

A little later it fascinated me when the first virtual memory systems were available. As a programmer, I was told I no longer had to worry about the size of memory. The operating system would automatically handle all the movement of data into the main memory for me. I quickly discovered even a simple scientific program could cause excessive paging depending on how it traversed a very large matrix. I had to be very careful about how I traversed matrices if I wanted my programs to work in a virtual memory system.

Ubiquity: You seem to be focusing on a pedagogical approach. You found from your own experience that your students would do better at designing systems when they were steeped in the organizations of the machines and network organizations on which their programs must run. Is this contrary to our standard pedagogy?

DM: Yes, it is. The current generation of students learns Java, Python, C or some other high-level language as their first programming language. There are definite advantages of doing that: These languages provide powerful constructs that allow programmers to build complex programs. On the other hand, by starting at that level, many students are failing to get an appreciation of how a computer really works. I believe students would be better served if they were first exposed to first principles and learned how to program in the assembly language of a simple hypothetical machine.

Ubiquity: So your preferred approach begins with very simple machine models that do not hide how the hardware, CPU, or memory works. You gradually move to higher levels of abstraction, always connecting each new level with the one below it—say through the workings of an assembler or compiler. I've heard some people argue against this approach on the grounds that it takes a long time to work your way up through the levels to get to the point where you deal with the really powerful abstractions, the ones you want to use in professional practice. What do you think of this argument?

DM: This is my preferred model indeed. This paradigm creates computing professionals who understand early on the connections between the abstractions and the computing platforms they are programming for. Therefore, they tend to become more proficient in what they do earlier rather than later.

Ubiquity: Is there enough time in a curriculum to teach students about the organization of the machine? There is so much to know about algorithms. Are we out of balance?

DM: Yes, I think we are out of balance. Students typically take a computer organization course after they had a couple of object-oriented programming courses. In the programming course, they deal with concepts like locks on objects and methods to avoid race conditions. But since they have not yet encountered machine organization, they cannot appreciate these features of the language. They get confused in trying to use them, and make many programming errors they do not understand.

This is why I advocate teaching rudimentary computer organization *before* teaching students how to program in a high-level language. We can place an advanced computer organization course afterwards. Learning about algorithms and their complexity is still very important, but it's so much more meaningful if you understand the limitations of the machine.

As to your question, how to fit everything in the curriculum? I think it can be done with proper planning.

Ubiquity: You mentioned the need for system designers to understand the organization of their distributed networks if they are going to be able to design for performance requirements. Just how much do they really need to know to do this well? Do they have to understand caches, interrupt systems, interprocess communication, sockets, and the like? Or will something at a higher level do the job?

DM: They need to understand enough to know how each of these components places a limit on the system. Designing for performance requires being able to map the behavior of a computer system into a predictive performance model. Because modeling can be done at multiple levels of detail, we usually start with a simple model and add details as necessary to get good agreement between the model and the real system. We have to do this as the system is being designed. As we come to know more details about the system being designed, we add more details to the model. We would include low-level details when they are highly influential on performance; for example, when we want to evaluate the effect of a large cache on a storage subsystem.

Ubiquity: Are there any other architectural aspects that have an impact on a system performance?

DM: Yes, the software architecture! Software engineers think of software system as a collection of interconnected software components. These components interact with one another through software connectors. We use architectural description languages to describe the structural and behavioral views of a software system. The performance of a software system is even more complex than the network hardware. It depends not only on the software architecture, but also on how the components are instantiated on physical resources such as service providers in a service-oriented network.

Ubiquity: You suggested we are having trouble meeting reliability and security requirements of systems because of the same syndrome, which I am now thinking is lack of appreciation for the architecture on which programs are running. How is the lack of appreciation for architecture contributing to these problems? How serious are the problems?

DM: The situation with reliability is very similar to that of performance. Reliability models can be used to estimate the reliability of a computer system as a function of the reliability of its components and their interactions. Security is much more difficult to quantify and assess. But again, a good understanding of a system's architecture and of the role security mechanisms play is essential. There is a growing movement to teach people how to write secure programs (as a practice)— to do that they need to understand the architecture features that make things secure or insecure. It is also important to note that there is interplay among reliability, security, and performance. A reliable design may be based on data replication across networked servers and may require the use of two-phase commit protocol for updates, which increases response time. Similarly, stronger cryptographic algorithms tend to demand more CPU cycles, which may degrade a system's performance.

Ubiquity: What hope do we have for resolving these problems? What do you recommend?

DM: Integrated education. We need to make performance, reliability, and security considerations an integral part of what we teach. In other words, database, computer architecture, operating systems, software engineering, and computer networking courses, should all discuss these issues in the context of these disciplines.

Here is a pragmatic example of the importance of integrated education. In watching cyber security exercises, I see the best defenders are the ones who really know the low level details of the system and the network, and not just the high-level program structures. Those who know only the high-level structures can't defend worth a whit. That's not the kind of person I want defending my critical infrastructures.

Ubiquity: In their preliminary draft of "Curriculum 2013," the joint ACM/IEEE committee has included a new knowledge area for security. They recommend an introduction to security so students get to know the problems and terminology. After that they recommend weaving security considerations into each other course as appropriate. Is this the approach you advocate?

DM: Definitely.

Ubiquity: Why should computing professionals be concerned with these problems?

DM: Because many are typically involved in the design of systems that are essential to the livelihood of people in the planet. These computer systems are ubiquitous. They control our cars, airplanes, air traffic, the flow of trains, our communications infrastructure, the electric grid, smart buildings, hospitals, the financial system, our defense systems, and much more. These systems are generally very complex and include very large numbers of subsystems. A designer of these systems needs to adopt the architectural perspective to understand how all the subsystems work together to accomplish a mission in timely, reliable, and secure fashion.

Some computing professionals are involved in designing the small building blocks used in larger systems while other professionals integrate the smaller building blocks into larger systems. They all need to work together. They can do this most effectively when they all share the same mental picture of the architecture model of the system they are building. Each of them then knows how his or her piece fits with the whole. That is a leadership issue for managers of systems projects.

Let us not get lost in a world of abstractions and forget that computing happens in the real world.

About the Author

Peter J. Denning is the editor-in-chief of *Ubiquity* and a past president of ACM (1980-82).

Currently he is a distinguished professor, chair of the Computer Science Department, and director of the Cebrowski Institute at the Naval Postgraduate School in Monterey, CA.

DOI: 10.1145/2188284.2188285