Editor: Geoffrey Fox, gcf@indiana.edu

MESSAGE PASSING: FROM PARALLEL COMPUTING TO THE GRID

By Geoffrey Fox

OVER THE PAST DECADES, THE COMPUTATIONAL SCIENCE COMMUNITY HAS DEBATED THE BEST ARCHITECTURE FOR PARALLEL COMPUTING. SOMETIMES IT'S DISTRIBUTED MEMORY, SOMETIMES SINGLE INSTRUCTION, MULTIPLE DATA (synchronous as in Thinking Machines' CM-1 and CM-2). Sometimes it's multiple instruction, multiple data, as in networked computers; or shared memory; or vector nodes; or multithreaded. And at times, it's more or less all of these. The debate recently perked up when the Japanese Earth Simulator supercomputer achieved 40 Tflops using a slightly heretical architecture. The arguments are accompanied by a related discussion as to the appropriate parallel computing model.

Whatever the machine architecture, users would certainly like to just write their software once and see it mapped efficiently onto parallel hardware. However, experience has found there is an almost irreconcilable difference between the way users would like to write their software and the way machines must be instructed to run efficiently. In particular, the natural languages for sequential machines do not easily parallelize. It is interesting that although languages are improving (Fortran, C, C++, Java, Python), writing parallel code has not gotten easier. Most science and engineering simulations are intrinsically parallel (as "nature is parallel" perhaps), but the obvious expression of these problems in today's common languages runs poorly on most parallel machines. In fact, the resulting parallelism is not explicit but a consequence of complex dependencies that are often only discoverable at runtime. Of course, major efforts to build better parallel compilers and runtime systems continue, but it is a difficult battle. This leads to the disappointing conclusion that the user must help the computer in some way or other. Then of course the different architectures suggest different programming models (openMP, HPF, UPC, MPI, ...). However, the conservative user will express the parallelism explicitly by dividing up the defining data domain into parts. The system manages each part as a separate process (the single-program, multiple-data model) that communicates via messages. We usually implement this messaging with MPI today.

This use of message passing in parallel computing is a reasonable decision, because the resultant code probably runs well on all architectures. This choice is not a trivial decision: it requires substantial work over and above that needed in the sequential case.

Messaging in grid and peer-to-peer networks

Now let us consider grid and peer-to-peer networks, which I've discussed in previous columns. Here, we are not given a single, large-scale simulation, the archetypical parallel computing application. Rather, we start with a set of distributed entities—sensors, people, codes, computers, data archives—and the task is to integrate them. In parallel computing, we decompose into parts; in distributed computing, we assemble parts. In some sense, decomposition is surely harder than composition (although Humpty Dumpty would not necessarily agree). In our case, the algorithmic and synchronization issues in parallel computing are technically very hard. For heterogeneous components and their linkages, it is the software engineering that is challenging.

In parallel computing, explicit message passing is a necessary evil. For grid and P2P networks, messaging is the natural universal architecture.

Objects and messaging

Object-based programming models are powerful, and objects naturally use message-based interactions. They have not been particularly helpful for the decomposed parts of parallel applications, because these are not especially natural objects in the system; they are what you get by dividing the problem by the number of processors. On the other hand, the linked parts in a distributed system (the Web, a grid, a P2P network) are usefully thought of as objects; in contrast, they are created for parallel computing by adapting the problem to the machine architecture.

1521-9615/02/$17.00 © 2002 IEEE

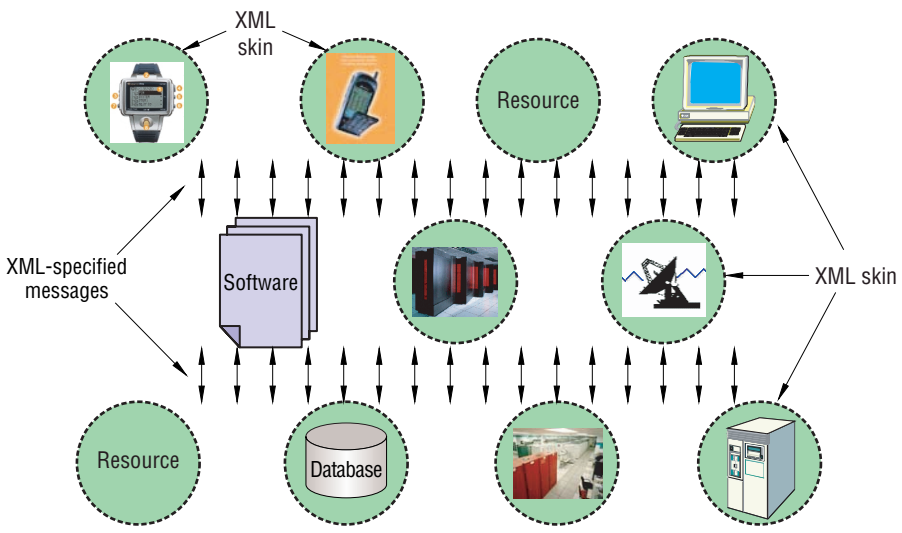

Figure 1. XML-specified resources linked by XML-specified messages.

Requirements for a messaging service

Messaging for distributed and parallel computing share some common features. For instance, in each case, messages have a source and a destination. In P2P networks especially, the destination can be specified indirectly and determined dynamically while the message is en route using the message's properties (published metadata) matched to subscription interest from potential recipients. Groups of potential recipients are defined in both JXTA (www.jxta.org) for P2P and in MPI for parallel computing. Collective communication—messages sent by hardware or software multicast—is important in all cases; much of MPI's complexity is devoted to this. Again, in both cases, we must support messages containing complex data structures with a mix of different types of information. We must also support various synchronization constraints between sender and receiver; perhaps messages should be acknowledged. Messaging systems share these general characteristics.

There are also many differences between distributed and parallel computing, so perhaps performance is the most important issue. The message passing of parallel computing is fine-grain; latencies (overhead for zero-length messages) should be only a few microseconds. The bandwidth must also be high and is application dependent, and communication needs decrease as each node's grain (memory) size increases. As a rough goal, it would be good if each process could receive or send one word in the time it takes to do a "few" (around 10) floating-point operations. MPI is trying to do something quite simple extremely fast.

Now consider message passing for a distributed system. Here, we have elegant objects exchanging messages that are themselves objects. As I mentioned, this object structure is natural and useful, because it expresses key system features. In my May/June 2002 column, I stressed that XML is a powerful new approach that expresses objects in a convenient way with a familiar syntax that generalizes HTML. It is not surprising that using XML to define distributed systems' objects and messages is now becoming popular. Figure 1 shows my simple view of a distributed system—a grid or P2P network—as a set of XML-specified resources linked by a set of XML-specified messages. Again, a resource is any entity with an electronic signature, such as a computer, database, program, user, or sensor. The Web community has introduced SOAP (www.w3.org/TR/2001/WD-soap12-part0-20011217), which is essentially the XML message format described earlier plus Web services, which are XML-specified distributed objects. Web services are "just" computer programs running on one of the computers in a distributed set. Using one of Apache's popular Web servers (www.apache.org) to host one or more Web services is a common approach. In this simple model, Web services send and receive messages on so-called ports; each port is roughly equivalent to a subroutine or method call in the traditional programming model. The messages define the subroutine's name and its input and (if necessary) output parameters. This message interface is called WSDL (Web Service Definition Language, www.w3.org/TR/wsdl), an important W3C standard.

As an example, the simplest Web service could be one that serves up Web pages, with the URL as input parameter and the page itself as returned value. By default, Web services use the same HTTP protocol as this simple case but use the rich XML syntax to specify a more complex input and output. The Web service is the unit of distributed computing in the same way that processes and threads are the unit for a single computer. Processes have many methods; correspondingly, Web services have many ports. As seen in Figure 2's P2P grid, ports are either user-facing (messages go between user and Web services) or service-facing (where messages are exchanged between different Web services). Using

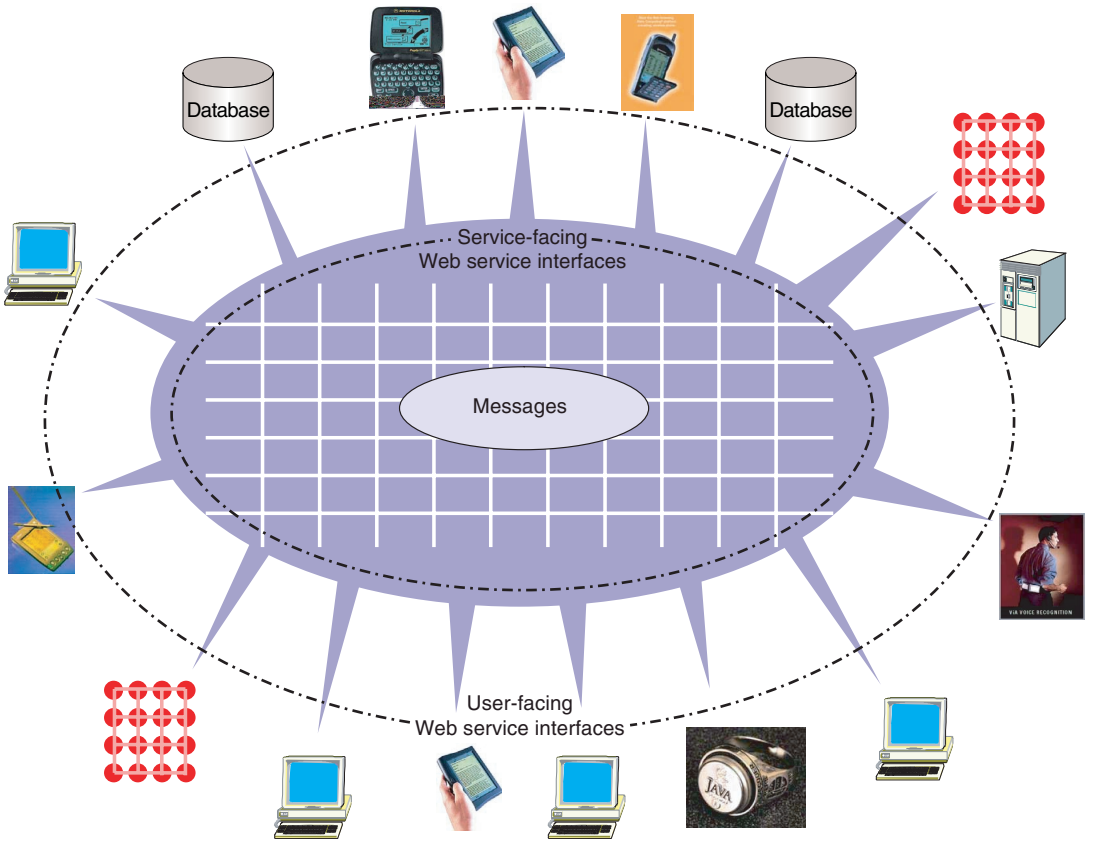

Figure 2. A peer-to-peer grid constructed from Web services with both user-facing and service-facing ports to send and receive messages.

Web services for a grid requires extensions to WSDL. The resultant Open Grid Service Architecture (www.globus.org/research/papers/ogsa.pdf) is a major effort in the Global Grid Forum (www.gridforum.org) at the moment. (I will explain Web services and WSDL in more detail in a future column.)

One particularly clever idea in WSDL is the concept that you first define not methods themselves but their abstract specifications. Part of WSDL "binds" the abstract specification to a particular implementation. You can choose to bind the message transport not to the default HTTP but to a different and perhaps higher-performance protocol. For instance, if you had ports linking Web services on the same computer, you could in principle bind them to direct subroutine calls. This concept has interesting implications for building systems defined largely in XML at the level of both data structure and methods. We can imagine some nifty new branch of compilation that automatically converts XML calls on high-performance ports and generates the best possible implementation.

Performance of grid messaging systems

The latency of grid messaging systems differs from that for MPI. It can take 10 to 100 milliseconds for data to travel between two geographically distributed grid nodes; in fact, the transit time becomes seconds if you must communicate between the nodes via a geosynchronous satellite. Thus, a grid is often not a good environment for traditional parallel computing. Grids do not deal with the fine-grain synchronization needed in parallel computing, which requires a few-microsecond MPI latency. Moreover, you can use different messaging strategies with a grid compared to parallel computing. In particular, you might be able to afford to invoke an XML parser and high-level processing for messaging. Note that interspersing a filter in a message stream—a Web service or Corba broker perhaps—increases a message's transit time by some 1 to 3 milliseconds (a small price compared to typical Internet transit times). This allows us to consider building grid messaging systems with substantially higher functionality than traditional parallel computing systems. The maximum acceptable latency is application dependent. If you are doing relatively tightly synchronized computations among multiple grid nodes, and if overlapping communications and computations hide the high latency, you must control the latency and reduce it as much as possible. On the other extreme, if the com-

putations are largely independent or pipelined, you only need to ensure that message latency is small compared to total execution time on each node. Another estimate comes from cases with users in the loop receiving messages. Here, a typical scale is 30 milliseconds, the time for a single frame of videoconferencing or a high-quality streaming movie. This 30-msec scale is not really a limit on the latency but on its variation. In most cases, a more or less constant offset is possible.

Now consider the bandwidth required for grid messaging. This situation is rather different: large amounts of information might need to be transferred between grid nodes, and you will need the highest performance the network allows. In particular, numbers often must be transferred in efficient binary form (say, 64 bits each) and not in some ridiculous XML syntax such as `<number>3.14159</number>`—with 24 characters requiring more bandwidth and substantial processing overhead. There is a simple but important strategy here; note that in Figure 1, I specified the messages in XML. This let me implement the messages in a different fashion, which could be the very highest performance protocol. As explained earlier, this is called binding the ports to a particular protocol in the Web service WSDL specification. So what do we have left if we throw away XML for the implementation? We certainly have a human-readable interoperable interface specification, but there is more—which I can illustrate with audio–video conferencing, which is straightforward to implement as a Web service. Here A/V sessions require some tricky setup process where the clients interested in participating join and negotiate the session details. This part of the process has no significant performance issues and can be implemented with XML-based messages. The actual audio and video traffic does have performance demands; here, you can use existing fast protocols such as RTP (Real Time Protocol). This is quite general; many applications consist of many control messages that can be implemented in basic Web service fashion, and just part of the messaging needs good performance. So, you end up with control ports running basic WSDL and high-performance ports bound to a different protocol.

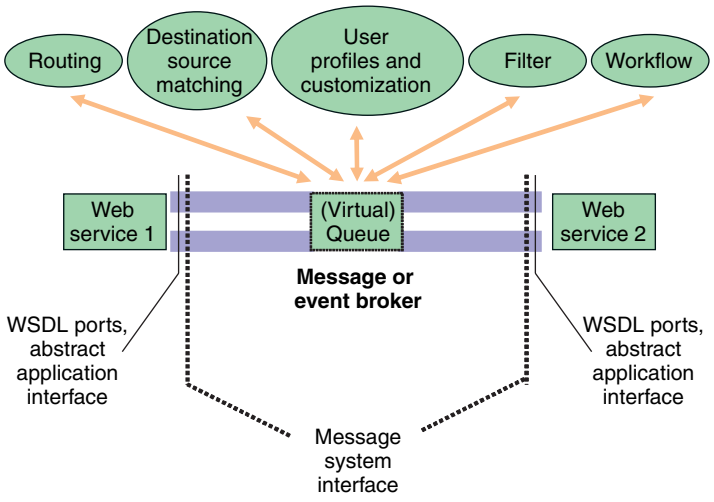

Figure 3. A messaging system for Web services.

Messaging services

Just as a standard MPI was good for parallel computing, so the different requirements of grid and P2P systems could lead to a new family of message-passing systems (see Figure 3). Such systems could handle several capabilities at the message layer largely independent of applications. These include

- Network quality of service (defined by the application)
- Secure transmission
- Collaboration
- Filtering channels to special clients such as PDAs or those on a slow network
- Efficient collective (multicast) messaging with rich matching between those sending and those interested in receiving information
- Tunneling through firewalls
- Allowing flexible delivery schedules linking synchronous and asynchronous schedules

These details are still at the research stage, but I expect more attention to be paid to messaging systems as we build large, distributed networks needed both in e-science (see this column in the July/August 2002 issue) and commercial service-based systems. The motivations for grid messaging systems will be even greater than those for parallel computing. You can find more information on my work in this area at http://grids.ucs.indiana.edu/ptliupages. Also, Shrideep Pallickara in the Community Grids Laboratory at Indiana (www.naradabrokering.org) has developed a message system for Web resources designed according to the principles I have sketched. It has been compared with typical commercial messaging systems, such as the Java Message Service, and that in P2P networks (www.jxta.org). I invite you to write to me or to the magazine (cise@computer.org) to report on work elsewhere.

Geoffrey Fox is director of the Community Grids Lab at Indiana University. He has a PhD in theoretical physics from Cambridge University. Contact him at gcf@indiana.edu; www.communitygrids.iu.edu/IC2.html.