Message Passing: From Parallel Computing to the Grid

Geoffrey Fox
Indiana University
Computer Science, Informatics and Physics
Community Grid Computing Laboratory,
501 N Morton Suite 224, Bloomington IN 47404
gcf@indiana.edu

Parallel Computing

Over the past decades, the computational science community has debated back and forth the best architecture for parallel computing; sometimes it's distributed memory; sometimes SIMD (synchronous as in CM-1 and CM-2 from Thinking Machines); sometimes its MMD (multiple instruction multiple data as in networked computers); sometimes its shared memory; sometimes vector nodes; sometimes multi-threaded; and sometimes more or less all of the above. This debate has been enlivened recently by the high performance achieved by the 40 teraflop Japanese Earth Simulator supercomputer 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 the parallel hardware. Experience has found there to be an almost irreconcilable difference between the way users would like to write their software and the way machines would like to be instructed to run efficiently. In particular the natural languages for sequential machines do not easily parallelize. It is interesting that even while languages are improving (Fortran, C, C++, Java, Python) it has got no easier to write parallel codes. 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. Of course there is continuing major effort on better parallel compilers and runtime but it is a difficult battle. Expressing most problems in existing languages leads to parallelism which is not explicit but a consequence of complex dependencies which are often only discoverable at runtime. 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, MPI …). However the conservative user will express the parallelism explicitly by dividing up the defining data domain and breaking it up into parts. Each part is managed as a separate process (the SPMD or single program multiple data model) which then communicate via messages. This messaging is usually implemented with MPI today.

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

Messaging in Grids and Peer-to-Peer Networks

Now let us consider the Grid and peer-to-peer (P2P) networks discussed in previous columns. Here we are not given a single large scale simulation – the archetypical parallel computing application, Rather ab initio we start with a set of distributed entities – sensors, people, codes, computers, data archives – and the task is to integrate them together. For parallel computing one is decomposing into parts; for distributed computing we are composing parts together. Actually decomposition is surely harder in some sense than composition although I am not certain Humpty Dumpty would necessarily agree. In our case, the algorithmic and synchronization issues in parallel computing are technically very hard. For composition it is the software engineering that is challenging for heterogeneous components and their linkages.
In parallel computing explicit message passing is a necessary evil. For Grids and P2P networks, messaging is the natural universal architecture. In the next sections we compare the requirements for a messaging service in the two cases.

Objects and Messaging

Object-based programming models are powerful and objects naturally use message based interactions. They have *not* been very helpful for the decomposed parts of parallel applications as 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 (Web, Grid, P2P network) are usefully thought of objects as here the problem creates them; in contrast they are created for parallel computing by adapting the problem to the machine architecture.

Requirements for a Messaging Service

There are common features of messaging for distributed and parallel computing; for instance messages have in each case a source and destination. In P2P networks especially, the destination may be specified indirectly and determined dynamically while the message is en route using properties (published meta-data) of the message matched to subscription interest from potential recipients. Groups of potential recipients are defined in both JXTA (http://www.jxta.org) for P2P and MPI for parallel computing. Collective communication – messages sent by hardware or software multicast – is important in all cases; much of the complexity of MPI is devoted to this. Again one needs to support in both cases, messages containing complex data structures with a mix of information of different types. One must also support various synchronization constraints between sender and receiver; messages must be acknowledged perhaps. These general characteristics are shared across messaging systems.
There are also many differences where perhaps performance is the most important issue. The message passing of parallel computing is fine grain – one must aim at latencies (overhead for zero length messages) of a few microseconds. The bandwidth must also be high and is application dependent and communication needs decrease as the grain (memory) size of each node increases. As a rough goal, one can ask that each process be able to 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 we explained this object structure is natural and useful as it expresses key features of the system. In an earlier article, we stressed that XML was a powerful new approach which expresses objects in a convenient way with a familiar syntax that generalizes HTML. It is not surprising that it is now becoming very popular to use XML for defining the objects and messages of distributed systems. Fig. 1 shows our 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; computer, database, program, user, sensor. The web community has introduced SOAP (http://www.w3.org/TR/2001/WD-soap12-part0-20011217/). which is essentially the XML message format postulated above and "Web services" which are XML specified distributed objects. Web services are "just" computer programs running on one of the computers in our distributed set. Often one would use one of the popular web servers from Apache (http://www.apache.org) to host one or more web services. 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 "old programming model". The messages define the name of the subroutine and its input and if necessary output parameters. This message interface is called WSDL (Web Service Definition Language http://www.w3.org/TR/wsdl) and this standard is an important W3C consortium activity. As an example the simplest Web service could be one that serves up Web pages and this has the URL as input parameter and the page itself as returned value. Web services use by default the same protocol HTTP 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 for a single computer. Processes have many methods and correspondingly web services have many ports. As seen in the peer-to-peer Grid of fig. 2, ports are either user-facing (messages go between user and Web Services) or service-facing where messages are exchanged between different Web services. We will explain Web services and WSDL in more detail in a later article. Using Web services for the Grid requires extensions to WSDL and the resultant OGSA (Open Grid Service Architecture http://www.globus.org/research/papers/ogsa.pdf) is a major effort in the Grid forum (http://www.gridforum.org) at the moment.

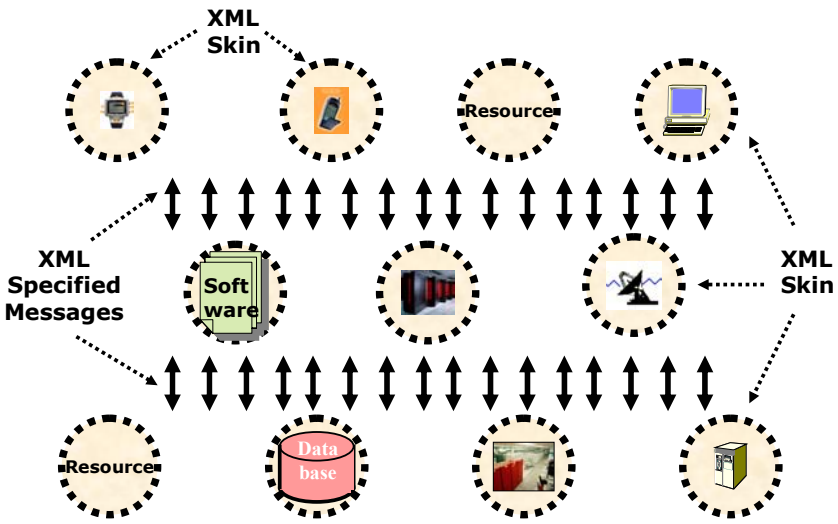

Fig. 1: XML Specified Resources linked by XML Specified Messages

One particularly clever idea in WSDL is the concept that one first defines not methods themselves but their abstract specification. Then there is part of WSDL that ”binds” the abstract specification to a particular implementation. Here one can as mentioned later

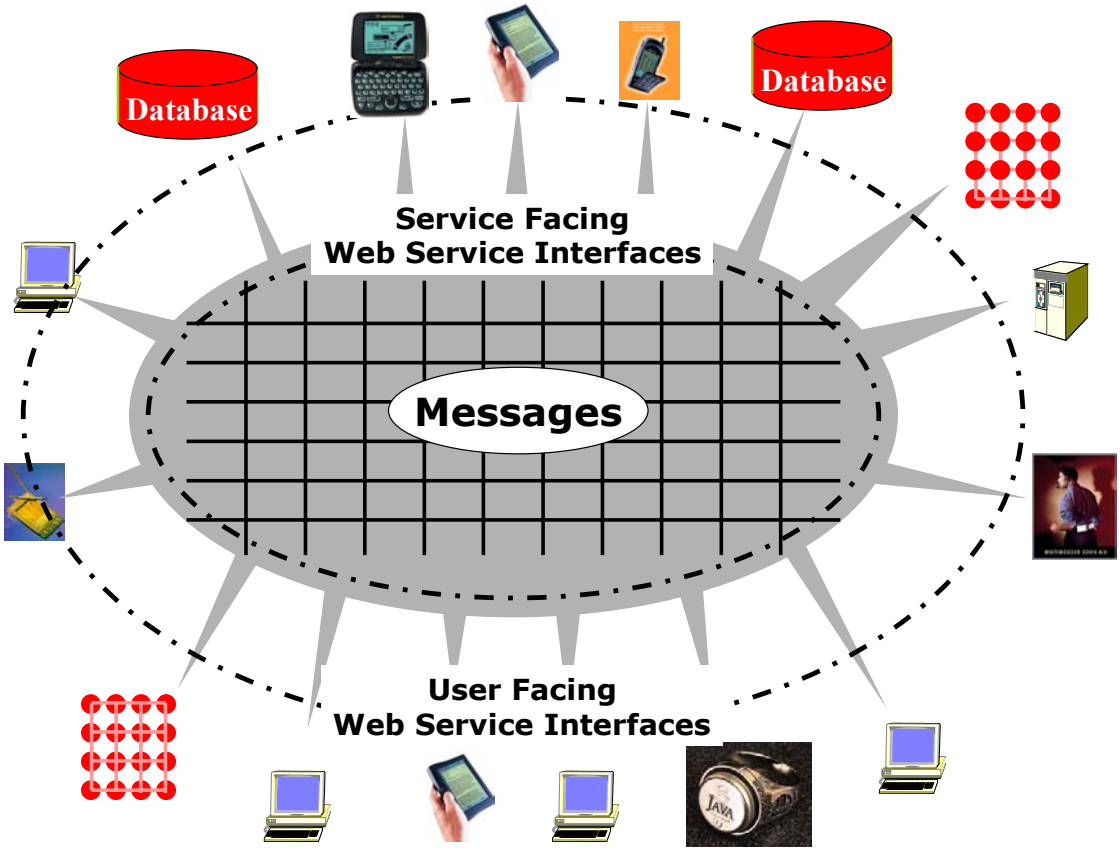

Fig. 2: A Peer-to-Peer Grid constructed from Web Services with both user-facing and service-facing ports to send and receive messages

choose to bind the message transport not to the default HTTP but to a different and perhaps higher performance protocol. For instance if one had ports linking Web services on the same computer, then these could in principle be bound 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. Further one can imagine some nifty new branch of compilation which automatically converted XML calls on high performance ports and generated the best possible implementation.

Performance of Grid Messaging Systems

Now let us discuss the performance of the Grid messaging system. The latency is very different from that for MPI as it can take 10-100 milliseconds for data to travel between two geographically distributed Grid nodes; in fact the transit time becomes seconds if one must communicate between the nodes via a geosynchronous satellite. One deduction from this is that the Grid is often not a good environment for traditional parallel computing. Grids are not dealing with the fine grain synchronization needed in parallel computing that requires the few microsecond MPI latency. For us here, another more

interesting deduction is that very different messaging strategies can be used in Grid compared to parallel computing. In particular we can perhaps afford to invoke an XML parser for the message and in general invoke high level processing of the message. Here we note that interspersing a filter in a message stream – a Web service or CORBA broker perhaps – increases the transit time of a message by some 1-3 milliseconds; small compared to typical Internet transit times. This allows us to consider building Grid messaging systems which substantially higher functionality than traditional parallel computing systems. The maximum acceptable latency is application dependent. Perhaps one is doing relatively tightly synchronized computations among multiple Grid nodes; the high latency is perhaps hidden by overlapping communication and computation. Here one needs tight control over the latency and reduce it as much as possible. On the other extreme, if the computations are largely independent or pipelined, one only needs 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 video conferencing or a high quality streaming movie. This 30 ms. scale is not really a limit on the latency but in its variation. In most cases, a more or less constant offset is possible

Now consider, the bandwidth required for Grid messaging. Here the situation is rather different for there are cases where large amounts of information need to be transferred between Grid nodes and one needs the highest performance allowed by the Network. In particular numbers often need to be transferred in efficient binary form (say 64 bits each) and not in some ridiculous XML syntax <number>3.14159</number> with 24 characters requiring more bandwidth and substantial processing overhead. There is a simple but important strategy here and now we note that in fig. 1, I proclaimed that the messages were specified in XML. This was to allow me to implement the messages in a different fashion which could be the very highest performance protocol. As explained above, this is termed 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 we can illustrate by audio-video conferencing, which is straight-forward to implement as a Web service. Here A/V sessions require some tricky set-up 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 and here one can use existing fast protocols such as RTP. This is quite general; many applications consist of many control messages, which can be implemented in basic Web service fashion and just part of the messaging needs good performance. Thus one ends up with control ports running basic WSDL with possible high performance ports bound to a different protocol.

Messaging Services

Shrideep Pallickara in the Community Grids Laboratory at Indiana has developed (http://www.naradabrokering.org) a message system for Web resources designed according to the principles sketched above. It has been compared with typical commercial messaging systems (JMS or the Java Message Service) and that in P2P networks (http://www.jxta.org). It seems that 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. One can identify several capabilities that can be handled 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, and 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 earlier article) and commercial Service-based systems. We see the motivation for messaging systems for the Grid to 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

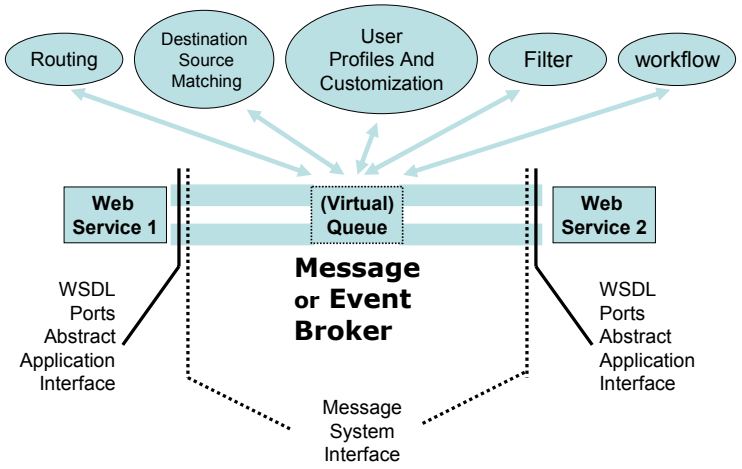

Fig 3: A Messaging system for Web services