Problems of computing on the Web

Slim Ben Lamine John Plaice Peter Kropf
Département d'informatique
Université Laval
Sainte-Foy (Québec) Canada G1K 7P4
Email: benlamin@ift.ulaval.ca, {plaice,kropf}@acm.org

1997

KEY WORDS

Web operating system, software configuration, eduction, load balancing, metacomputing.

1 Abstract

We discuss the concept of computing on the Web. We show that the heterogeneous and dynamic nature of the Web makes it impossible to define a fixed set of operating system functions, usable for all services. Rather, we propose that generalized software configuration techniques, based on a demand-driven technique called eduction, can be used to define versions of a Web Operating System (WOS) that can be built in an incremental manner. We illustrate this problem by examining the question of load balancing.

2 Introduction

With the rapid development of new forms and concepts of networked and mobile computing, it is increasingly clear that operating systems must evolve so that all machines in a given network can appear to be controlled by the same operating system. As a result, the world-wide interconnected networks, commonly called the Internet or the Web, could potentially be supported and managed by a giant virtual operating system (Reynolds 1996).

For example, initially the World Wide Web was created to allow one to view remote hypertext pages on one's own machine, thereby facilitating collective work among geographically removed collaborators. Soon after, virtual pages, generated on the fly using tools such as `cgi-bin`, allowed the widespread remote execution of programs. More recently, with languages such as Java and Limbo, it has become possible to download fully executable programs to one's own machine, and then to make them run on that machine. However, there is no general means for taking an arbitrary program and having it run somewhere on the network.

There are several reasons that this last possibility is actually essential. First, with the development of network-centric computing, there will be more and more limited-capacity machines (slower processors, limited memory or storage space, etc.), such as the NC computers proposed by several vendors, that will be forced to use more powerful computers on the network to effect any non-trivial tasks, or simply to download the necessary programs. Second, an arbitrary program on the network might just be incapable of running on the local machine, simply because it is the wrong platform (hardware, local operating system, running applications, etc.).

In the case of a program that requires high performance capacities, both of these points apply. The typical high performance program requires either specialized hardware or clusters of standard hardware, carefully integrated for running computation-intensive applications. The typical

machine on the network will neither have the physical capacity nor the software necessary for running such programs. As a result, a request to run such software can only be satisfied by finding a machine or a subnetwork that can effect the required task.

Implicit in the above discussion is the *heterogeneous* nature of most networks, in particular of the global information infrastructure (Williams 1996). This problem has been partially addressed in work on *metacomputing*, whose aim is to transform a network of computers into a single computer, where the underlying resources are transparent to the user.

However, the Web is more than just a metacomputer, in that there is no complete catalog of all of the resources available on the Web. Moreover, such a catalog is infeasible, because of the highly dynamic and distributed nature of the Web, which is continually integrating rapidly developing technologies.

Therefore, we propose that there is a need for a *Web Operating System* (WOS), which would make available, to all sites on a network, the resources available on that network, or at least a reasonable subset thereof, to effect computations for which local resources are missing. These resources could be of many forms, including processor, memory or storage space, available operating systems or applications, and so on.

In this paper, we present the fundamental problems to be solved to make the WOS possible. We consider the heterogeneous and rapidly evolving nature of the Web to be the key problems to be dealt with, and show how the concepts of software configuration and version control can be applied to these key problems. Since we are focusing on performance issues in this paper, we do not address the questions of security or access rights, although these must of course be handled adequately in such an environment.

Below, we begin by giving an overview of the heterogeneous nature of the Web, presenting requirements for a successful WOS. We follow with an introduction to software configuration and its computational model, *eduction*. We then show that generalizing these concepts will allow us to solve some of the problems of heterogeneity, using as example the question of load balancing across the network. We conclude with a discussion of future work.

3 The heterogeneous Web

If the Web, in the general sense, is the sum of networked computing, it is clear that there are untold numbers of different services being offered, and that many more will be offered in the years to come.

Should a WOS be developed on a large network, it would have to support at least the following kinds of services:

- "network applications": WWW, email, video, etc.
- "computational applications": number crunching, distributed simulations, etc.
- "transactional applications": banking, electronic commerce, travel reservations, etc.
- "virtual entities": classrooms, companies, etc.
- "knowledge-based applications": data mining, databases, etc.
- "real-time applications": process control, real-time multimedia, etc.

These different services typically require different resources, as well as different strategies to ensure that proper service is offered. For example, for video, an aggressive data-driven strategy of multicasting video frames to the recipients as quickly as possible should be used, thereby reducing the chance that a network brownout will not prevent a video from being viewed. On the other hand, a program interacting with a database would more likely require some form of demand-driven strategy controlled by the user of the application.

Given the wide variety of imaginable services, and their differing needs, as well as the growth of unimagined services, it is clearly *impossible* to devise *the* operating system that is going to

support every service with ease. Rather, general support should be given for (at least) *versioned* resource management, caching techniques, processor scheduling, communication protocols and runtime systems, as well as versioned general policies. In other words, rather than have the operating system provide a fixed set of processor scheduling or caching techniques, the operating system should provide the means for an application to designate the particular techniques that it requires, possibly even providing its own. See (Reynolds 1996) for a similar discussion, referring to recent work in application-specific resource management.

The Web is not just heterogeneous at the conceptual level, i.e. in the offered services, but also at the physical level, i.e. in the hardware and software that is available at different sites. Different technologies are used for all levels of networking, and different machines, from personal computers to workstations to supercomputers of all kinds, can all be found on the Web, each with its own particular setup of operating systems, supported protocols and applications. Each of these programs can, of course, itself be versioned.

In other words, should a particular service be requested, then not only must the conceptual means for that service be found, but also the physical means (subnetwork, processors, disks, etc.) for ensuring that the service can actually run.

Of course, in this discussion, we have only been discussing the situation for a *snapshot* of the Web, i.e. the situation on the Web at any given moment. But the Web is a highly evolving entity. On the short-term level, loads are shifting and vary from one machine to another, from one subnetwork to another, computers and networks go off-line and come back, software upgrades are made, and so on. On a longer-term scale, services change, new services are introduced, and new technologies are integrated and made available.

To aggravate this situation even more, the very basis for the Web and the Internet, namely IP, is itself not static. The current standard, IPv4, is soon to be replaced by IPv6. However, this change will take place over a long period, and the two versions of IP will be used simultaneously (Hinden 1996). So even the network level of the Web is itself heterogeneous.

As a result, versions—of services, hardware and software—are ubiquitous on the Web. As is normal wherever versions appear, there are *revisions*, corresponding to successive stages in a straight-line development process, and *variants*, corresponding to separate branches in the development process, such as for interface language or hardware platform. And, of course, the evolution of an entity can ultimately lead to a completely different entity.

Given this situation, it is not even possible to make a complete specification of the problem of defining a WOS, as the problem will have greatly evolved before any implementation can take place. This is not a problem particular to the Web (Mullery 1996), although the rapid pace of technological development on the Web just aggravates the situation.

Given this daunting version problem, one might be tempted to implement the upcoming WOS in some platform-independent language, such as Java, thereby simplifying the version problem. However, if high performance is required, then careful tuning to (at least) the local environment is imperative, and so native code generation is absolutely essential. Therefore, even if platform-independent code is used, it would have to be compiled into highly efficient native code, possibly through some form of on-demand compilation.

4 Software configuration

The concept of versioning is not new to software development. Bugs must be fixed, demands change, new languages and platforms must be supported, and so on.

Any given software component, be it a full-fledged system or a small documentation file, can come in many different versions. The versions can be *conceptual*, in the sense that up to now no-one has ever actually built that version, or *physical*, in the sense that if one requests that version, it can be supplied immediately. The set of conceptual (resp. physical) versions of a software component is called its *conceptual* (resp. *physical*) *version space*.

Software configuration refers to the building of subsystems or of complete systems from their basic components. If every component only came in a single version, the problem would be quite

simple. The more typical case is that each component has its own version space. What this means is that when the build of a particular version of a system is requested, i.e. when a conceptual version is being transformed into a physical version, then the appropriate version of each component must be selected before the build can be effected.

To further complicate matters, the very structure or architecture of a software system may itself be versioned. Some modules may be required for some versions of a system, while other modules will not be required. As a result, one cannot just request the appropriate version of every module, since not all modules will be required in every version. Rather, the main module's structure must be determined, then the appropriate versions of each component are determined, and so on.

In order for automatic software configuration to take place, a process called *eduction* must be used. Eduction, a form of demand-driven lazy evaluation, was first proposed for the execution of dataflow languages (Wadge and Ashcroft 1985). Since then, it has been generalized for many other computer problems (Orgun and Ashcroft 1996).

The eductive process for software configuration is as follows (Plaice and Wadge 1993). A version of a system is requested from a *warehouse*, in this case a software repository. To build that version, appropriate versions of a set of subsystems must be requested. For each subsystem, the versions of its own subsystems must be requested, and this process takes place in turn until finally the versions of atomic components, such as source files, are selected. Compilation and building can take place once all components, as well as their respective versions, have been identified. Once the build is finished, then the finished system can be added to the warehouse, so that if it should be requested in the future, it can be returned immediately rather than being rebuilt.

If the version space of a system can be related to the version spaces of its components, then software configuration can be simplified. However, this is not always the case, and in a situation such as software development on the Web, this would actually be the exception.

Eduction can be generalized for different purposes. In the case of heterogeneous computing, *two* warehouses could be used, one containing available versions of software, and the other available platforms. Requests for a particular version of software would be made to the first warehouse. Rather than to return a single version, the first warehouse would return information about all of the versions that meet the requested query. The *eductive engine* would then ask of the second warehouse which of the versions could actually be run, given the available platforms. Once a match is made, then the appropriate software version could be run on the appropriate platform.

It should be clear from this discussion that eduction can be generalized to use multiple warehouses, distributed geographically and for different purposes, and that multiple eductive engines can also be made to run simultaneously.

5 Building a WOS

We can now see what the kernel of a WOS should look like. It should be a general eductive engine, that would allow interaction with many different warehouses, each offering different versions of services, resource management techniques, applications, platforms, hardware, etc. An eductive engine is a *reactive system* responding to requests or queries from users or other eductive engines, and that fulfills these requests using its warehouses.

The WOS would therefore consist of a collection of eductive engines, as well as many different warehouses, all available from the Web. What changes fundamentally from the previous section is that some warehouses would change very quickly, as they would be used to describe, say, network or workstation load. This means that the warehouses would be time-dependent, hence only an approximation of the actual situation. As such, the proper updating of these warehouses would itself constitute a difficult problem.

The WOS would then work in the following manner. A request would be made by a user to run a particular program or to initiate some service, along with specified data. The program or service and the data might all be located at different sites on the Web, although in practice, most

of these entities would be local. This request, possibly filled out through some kind of form, would then be sent to the closest eductive engine, which might reside on one's own machine.

Upon reception of such a request, the eductive engine would decide whether it is capable of dealing with the request. After all, that engine might simply be overloaded and the request might be of too high a priority to wait. The engine would look in its software warehouses to determine if it actually has the requested program. If not, it might refuse service or pass on the request to one or more other eductive engines, until finally one engine accepts responsibility for the request.

We now suppose that the engine has references to this program in a software warehouse. However, this program might be available in many different variants, and it might still be unclear which of these is the most suitable for the user. In that case, an interactive session could be set up with the user to request more information, thereby narrowing down the set of possible variants. This session could also be run automatically, since a user's local machine might well contain information about a user's preferences (e.g., Arabic user interface, X-Windows, fastest possible perfomance, ignore costs, etc.).

Once a variant has been selected, then the various resource warehouses would have to be consulted, to determine if any of the implementations of that variant could be run locally or elsewhere on the network. Each resource itself might come in different versions, and might itself require other resources, all of which would have to be allocated by the eductive engines.

Once all of the resources become available, then the program could be run. If it were a short run, then that would be all that is required. However, it might occur that this program would run over an extended period, in which case the information that was used might become out of date, and resource allocation would have to be periodically updated. For example, crucial network links might go down while effecting transfers, and these links would have to be recreated.

It should be clear from this discussion that ultimately, the WOS will offer incredibly powerful kinds of services. However, it would be inappropriate to attempt to implement "everything at once". So doing would guarantee failure, in that the very notion of "everything" would have so changed by the time any implementation had taken place that it would be irrelevant. More to the point, the WOS can be built in an incremental manner, beginning by offering basic heterogeneous support for local area networks, sending tasks to machines with the appropriate resources. As time goes by, additional services can be added, without rendering obsolete already running eductive engines.

6 Example: Load balancing

Being able to use the global network for parallel execution of a program is clearly promising. For this idea to be realistic, mechanisms are required to distribute the work, collect the results and coordinate the participating processes or agents. If these mechanisms are effective, the potential performance of the components (processor performance, network bandwidth, etc.) must be taken into account, as must the dynamically changing system load.

To minimize execution time of a distributed or parallel application, the work load must be balanced across processors as well as possible. There already exist decent techniques for statically and dynamically balancing the work load. However, most of the literature on load balancing (e.g., Cybenko 1989; Kumar *et al.* 1994; Mansour and Fox 1992; Scheurer *et al.* 1995; Xu *et al.* 1995) makes assumptions that are simply unrealistic on a global network.

First, much of the literature assumes that the network topology is uniform. More importantly, it is assumed that the topology is static, which clearly cannot be the case for a large network of heterogeneous computers, since lines may be cut, certain sections of the network may become saturated, or some machines might be unreachable at certain times.

Second, it is assumed that a fixed amount of useful work is to be done, and that the total amount of work to be effected corresponds to the sum of the useful work and the overhead. There are clearly situations where it costs less for certain computations to be effected in a redundant manner, in order to avoid unnecessary communication (data distribution vs. data replication).

Third, it is assumed that all messages are of the same size. Some of the more recent literature has addressed this problem (Kumar *et al.* 1994).

Next, it is assumed that all processors are equivalent. The typical laboratory—and, of course, the network—consists of machines that have been acquired over a period of time, and are of different capacities, in many senses. It might be simpler to have a homogeneous environment, but few sites can afford the cost of completely retooling their laboratories as technology progresses. Rather, they partially retool as time advances.

Finally, it is assumed that all processors are running simultaneously. Making this assumption means that the failure of a single component can bring down a large series of computations, clearly not a useful result. Fault tolerance should always be built into a metacomputer. The question then becomes, what kind of fault tolerance is required?

It is precisely to deal with such situations that the WOS is so important. Rather than to build *the* load balancing scheme that will be used in *all* situations, a family of load balancing schemes can be devised. Different schemes could be application specific or resource specific, or a combination thereof, depending on need.

Additional information acquired from the program could also be considered. This information could be provided by the user, the compiler or the statistical results culled from previous runs. Then future runs of the same program could take place more efficiently.

This approach would thus allow the implementation of a family of adaptive and dynamic fault-tolerant load management systems, all integrated into the WOS.

7 Conclusions

A major goal of the eductive engine is to provide to a user the most effective hardware and software environment. In particular, this means providing the fastest possible application execution, respecting user price/performance, as well as response time requirements. Therefore, the eductive engine must be able to provide resources as quickly as possible from the point of view of response time, as well as the "best" resources regarding performance requirements.

For example, in a medical environment, for a diagnostic request such as doing a simple comparison of a patient's Computed Tomography (CT) scan with a CT database, followed by the consultation of an expert system, a rapid response is fundamental, more so than the fastest execution, for which the eductive engine might require more time than the desired response time.

On the other hand, where the highest performance requirements are needed, as in the case where an application might take hours or days of computation time, such as in some particle simulation or finite element analysis problems, it would certainly be worthwhile to let the eductive engine search for the best (fastest) environment to run in.

It is clear that these tradeoff issues can become quite complex, depending both on the applications and the factors that are considered to be relevant, as well as on the number and specificity of the different kinds of resources that might be available on the Web.

For example, in some of the above medical examples, we might wish to be able to effect dynamic changes, adding processors, database engines, increasing network bandwidth, dynamically replacing the eductive engine, and so on. There is no limit to the possible improvements that one could make with this sort of application: medical science can always use more powerful tools, be it for diagnostics, computer-assisted surgery, or fundamental research.

The Web is an incredibly heterogeneous and dynamic environment. Therefore, as the WOS develops, successful eductive engines will do most of their computation through *negotiation* (with other eductive engines as well as with warehouses) and through *approximation* (since we can never have a completely accurate picture of the current status of the Web).

The above picture might appear to be unsolvable, since everything moves. Nevertheless, in this sea of versions, we *can* find a firm place to stand on. It is the precisely the evolving Web that offers us the unique opportunity of incrementally building these eductive engines, adding functionality as needed and as further provided by technological advances.

Furthermore, version spaces *can* be structured so that they can be used by a wide variety of applications (Plaice and Wadge 1993). In fact, much of our current research deals with the interoperability of version spaces of different entities, which will be a key issue for the successful development of the WOS.

References

Cybenko, G. 1989. Dynamic load balancing for distributed memory multiprocessors. *Parallel and Distributed Computing*, 7:279–301.

Hinden, R.. 1996. IP Next Generation Overview. *Communications of the ACM*, 39(6):61–71.

Kumar, V., Grama, A.Y., and Rao, V.N. 1994. Scalable load balancing techniques for parallel computers. *Parallel and Distributed Computing*, 22:60–79.

Mansour, N. and Fox. G. 1992. A comparison of load balancing algorithms for parallel computations. In *Proceedings HPCS'92*. CRIM, Montréal.

Mullery, G. 1996. The perfect requirement myth. *Requirements Engineering Journal*, 1:132–134.

Orgun, M.A. and Ashcroft, E.A., eds. 1996. *Intensional Programming I*. World-Scientific.

Plaice, J. and Wadge, W.W. 1993. A new approach to version control. *IEEE Transactions of Software Engineering*, 19(3):268–276.

Roberts, F. 1996. Evolving an operating system for the Web. *IEEE Computer*, 29(9).

Scheurer, C.; Scheurer, H.; and Kropf, P.G. 1995. A process migration and load management tool. In *Proceedings HPCS '95*. CRIM, Montréal.

Wadge, W.W. and Ashcroft, E.A. 1985. *Lucid, the Dataflow Programming Language*. Academic Press.

Williams, D.O. 1996. Review of wide-area aspects of high performance networking. *Conference of the Federation of European Simulation Societies*. Elsevier.

Xu, C.Z.; Lau, F.C.M.; Monien, B.; and Lüling, R. 1995. Nearest-neighbor algorithms for load balancing in parallel computers. *Concurrency, Practice and Experience*, 7:707–736.

Biographies

Slim Ben Lamine received an engineering degree from the École Nationale des Sciences d'Informatique (Tunis, Tunisia), and the M.Sc. degree from Université Laval (Québec, Canada), both in computer science. He is currently a Ph.D. candidate at Laval. His research interests are in configuration management, version control, information systems (Scott domains), in particular as they relate to the dynamic aspects of the Web.

John Plaice received the B.Math degree from the University of Waterloo (Ontario, Canada), and the DEA (master's) and Ph.D. degrees from the Institut National Polytechnique de Grenoble (France). He is currently an Associate Professor at Université Laval. His research interests cover all areas of computer science in which change and time play a role. He has recently published papers on real-time programming, dataflow programming, intensional programming, version control, software configuration, particle simulation and multilingual typography, as well as on the fact that computer science is an experimental science.

Peter Kropf received the M.Sc. and Ph.D. degrees from University of Bern (Switzerland). Subsequently, he was a research scientist at the University of Bern for several years. Since 1994, he has been an Assistant Professor at Université Laval. He has been carrying out research in parallel computing for over ten years, particularly in the field of mapping and load balancing.

Current projects especially include the efficient use of parallel computing in real world applications (radiotherapy, biomechanics, radio-reconnaissance, etc.). His research interests include parallel computing tools, networking and simulation.