SIMULTANEOUS MULTIPLE VERSIONS: THE KEY TO THE WOS

SLIM BEN LAMINE
JOHN PLAICE

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052 Australia
tel: +61.2.9385.6012
fax: +61.2.9385.5995
email: {benamin,plaise}@cse.unsw.edu.au

Abstract
The rapid development and the heterogeneous nature of the Web ensure that it is impossible to develop a complete catalog of all of the resources and services available on the Web. As a result, no single operating system can be used for Web computation, since it will necessarily be incomplete. Furthermore, it is unclear that such an operating system would even be useful, considering the different levels of granularity of service that need to be provided.

The Web Operating System (WOSTM) approach to Web computation envisages a series of versioned servers, possibly offering different services, themselves versioned, that use the same basic protocol for communication amongst themselves, and that are capable of passing on requests for service when appropriate. The WOS is not defined by the actions of any single server but, rather, by the combined actions of the different servers.

Key to the successful implementation of the WOS is the ability for multiple different versions of the WOS to interact in a meaningful manner, and that a request for service be passed on to a server actually capable of managing that request.

Each WOS server will have a version tag, which can be readily translated into a fixed catalog of guaranteed services. In addition, a WOS server might offer additional services, either offered en masse in an additional catalog, or one at a time, on demand.

Once presented in this manner, the problem becomes one of distributed software configuration management. There is no single repository in which all possible versions are stored, and there is no means for recuperating all of the possible versions. In this paper, we show how these problems can be addressed by generalizing standard techniques.
1 INTRODUCTION

With the rapid development of the Internet, and of the various services offered on top of the Web, it is tempting to imagine that we are heading towards an era of widely integrated services through some form of operating system distributed across the world.

However, given the wide variety of imaginable services, and their differing needs, as well as the growth of unimagined services, it turns out to be 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.

In this paper, we discuss the means by which a Web Operating System can be devised by allowing different servers to run different services, while sharing a basic communication protocol among themselves to pass on requests that cannot be handled locally.

We begin in Section 2 by giving an overview of software configuration management, by focusing on version spaces (of software modules) endowed with partial orders. We then show in Section 3 that these ideas can be generalized nicely to multiple servers providing different (versioned) services, using the concept of catalogue. In Section 4, we explain how the search
for a particular version of a particular service is satisfied. Then we show how new servers can be set up, and we conclude with a discussion of future work.

2 ORDERED VERSION SPACES

Software configuration designates the building of a system from its components. This process might be single-layered, in which a complete system is built directly from atomic components, such as source, text or graphics files; or multi-layered, in which case subsystems are successively built up from component subsystems, which have themselves already been configured.

At each level, each system or component—hereafter called a module—can have several versions. We call the set of conceptual (possible) versions—those that can be imagined or “thought of”—of that module its conceptual version space, and the set of physical versions of that module its physical version space.

What distinguishes one version from another are explicit features, which can come about in many different ways. Some will be changes resulting from straight-forward linear development. Others will designate choices or variants, for such concepts as interface languages or host implementations. Given the variety of these features, we choose to call them attributes and we will assume that a version is completely defined by its description, which is the set of attributes of which it is composed.

For example, the description of the initial version of a one-module program for drawing geometric figures might just be \{base\}. Once it has been used, this version might be independently transformed into two other versions, \{base, optimized\} for an optimized version, and \{base, graphic\} for a graphic version using a bitmapped screen. Combining the features of these two versions would require additional work, thereby resulting in description \{base, optimized, graphic, graph_opt\}.

From this example it should be clear that there is typically a causal relation between attributes. In fact, the graphic attribute implies the base attribute, since the graphic attribute is tied to the original version, which has the base attribute. Furthermore, graph_opt implies the graphic, optimized and base attributes. These causal relationships define a partial order, written as follows:

\[
\begin{align*}
\text{base} & \subset \text{optimized} \\
\text{base} & \subset \text{graphic} \\
\text{optimized} & \subset \text{graph_opt} \\
\text{graphic} & \subset \text{graph_opt}
\end{align*}
\]

Once the order on attributes is defined, it is no longer necessary to specify all of the attributes to designate a version. For example, the set \{graph_opt\} unambiguously designates version \{base, optimized, graphic, graph_opt\}. Similarly, \{graphic\} designates version \{base, graphic\}. We will use the term partial description for those sets that unambiguously designate descriptions, even if they are not themselves full descriptions.

In both of the examples in the previous paragraph, the partial description consisted of the greatest attribute in the complete description. What happens if the greatest attribute is not included, for example, with \{graphic, optimized\}? Intuitively, this means that a version that offers both the graphics and optimized capabilities is requested. This means that
the two must be combined. So the greatest lower bound—with respect to the partial order—of these two attributes is sought. Hence, \{graphic, optimized\} should also designate version \{base, optimized, graphic, graph_opt\}.

Therefore, to create a description from a partial description, all that is required is to compute the downward closure of the least upper bound of the partial description. However, it turns out that this approach will not always work.

For example, consider the attribute space in Figure 1 (a). Set \{a, b, c, e, f, g\} might be considered to be a version, as it has a least upper bound. However, the axioms of information systems insist that every subset of a version should be consistent, i.e. have a least upper bound. But this is not the case, since set \{a, b\}, although bounded above, does not have a least upper bound.

![Figure 1: Factorization of an attribute space](image)

In fact, this example shows the kind of problems that can occur when concurrent engineering is taking place. Attributes e and f each play two roles: first as join of b and c, second as their own additional deltas. But the join of b and c does not exist on a standalone basis, despite the fact that it is required for both e and f. One way to handle this situation is to factor out the common changes to e and f, namely the join of b and c, then to create the necessary changes to reach e and f. The new order would then be the one presented in Figure 1 (b).

An additional condition on attribute spaces ensures that everything works. If every subset bounded above has a least upper bound, then the versions can be constructed automatically. See (Ben Lamine and Plaice 1997) for a full discussion of version spaces as information systems à la Dana Scott.

When multiple modules are being managed, then there is an additional problem to deal with: not all modules vary equally. However, if the partial order approach is used, then a general mechanism called the variant substructure principle can be used. When version V of a structured module is being built, then version V of each of its component modules is requested, and the most relevant approximation of each is returned. See (Plaice and Wedge 1993).

To conclude this section, it is natural for any given module to have a version space, and for versions to be designated by abbreviations that describe the essential attributes that those versions must provide, and that the complete version descriptions be completed automatically. When several modules are requested simultaneously, the simplest way to understand the process is to have each module dealt with separately, and then to combine the results at the end.
3 WOS SERVERS OFFER CATALOGS

Now that we have an general understanding of system configuration management, we can begin to apply these ideas to the WOS.

First, rather than referring to modules, we will refer to services, a generic term that could mean practically anything: access to an operating system, an application, a data package, etc.

A significant difference with the configuration management problem from above is that the WOS is highly distributed, with no central node. As a result, all identifiers must be sourced, i.e. each identifier, be it for a service, a version attribute, or something else, must originate from a particular server. We will therefore write, when necessary, identifiers as $id_w$, where $w$ stands for a particular WOS server; similarly $w(id)$ will give the source for $id$.

Also because of the distribution, there is no way to globally check for coherency of version spaces. This can be done for restricted sets of spaces, but not globally.

To begin, a WOS server is a site, with an address, that offers a number of services. For each of these services, several versions might be available. We suppose that there exists a set of services $S(\exists s)$.

Suppose that $s_w \in S$ is a service originating with server $w$, say with version $v_w$; we can then refer to the pair $(s_w, v_w)$ as a versioned service. Then suppose that version $v_{w'}$ of $s_w$ is provided at server $w'$, and it was built from the previous version. Then $(s_w, v_{w'})$ is also a versioned service, and we write $(s_w, v_w) \rightarrow (s_w, v_{w'})$.

A set $C$ of versioned services $\{(s_i, v_i)\}$ is called a catalog if there exists a finite graph that corresponds to the transitive closure of the dependencies generated from the $(s_i, v_i)$.

A WOS server $w$ is said to offer a catalog $C$ of services if for every $(s, v) \in C$, $w$ makes available versioned service $(s, v)$.

For example, there might be a WOS server $gnu$ providing the standard GNU applications running on Unix, while another WOS server $hpc$ might provide support for high-performance computing. For each of these servers, many different versions of each of the applications might be available simultaneously. For example, the server $gnu$ might contain both $gcc_{2.8.1}$ as well as $gcc_{2.7.2.3}$, while server $hpc$ might contain both HPF data and HPF task.

It should also be understood that except in the case where a WOS server $w$ offers no service, then there is no single catalog that defines all of the services offered by $w$. In fact, catalogs form a natural partial order, and we say that $C'$ includes $C'$ if $C \subseteq C'$. This approach is consistent with the idea that a WOS server might actually provide services as needed, rather than preparing them all in advance. In fact, in many interesting cases, the set of possible services might be infinite.

Now, since the WOS is to be used for high-performance applications, catalogs cannot be passed around between servers all of the time. More appropriate would be to pass names, or types, corresponding to catalogs. As with services and versions, the types must originate on a particular server. If $T$ is a type, then catalog$(T)$ will provide the relevant catalog.

A WOS server $w$ is said to be of type $T$ if $w$ offers catalog$(T)$. Clearly, $w$ can be of several different types. Also, since catalogs form a partial order, so do types. We say that $T$ includes $T'$ if catalog$(T)$ includes catalog$(T')$.

Up to now, it has been assumed that the services being offered are independent of each other. However, in general this will not be the case. It may well be that a service $s$ can only be offered if a service $s'$ is also available. For example, GNU’s tar program requires the gzip program
to be available. So we introduce a new kind of dependency, \((s, v) \Rightarrow (s', v')\), which states that any catalog containing \((s, v)\) must also contain \((s', v')\).

4 MANAGING WOS QUERIES

A request for service from the WOS is initiated by a user sending a message of the form \((s, v)\) to a WOS server \(w\). The idea is that if \(w\) can offer this service, it will do so. Otherwise, it will pass on the request to another server that is more likely to be able to handle the request.

Each server \(w\) retains a cache, or a warehouse, describing other servers \(\{w_1, \ldots, w_n\}\) that it knows about. For each server \(w_i\), it also retains a list of catalog types \(\{C_{i1}, \ldots, C_{im_i}\}\) for which it knows that server \(w_i\) is of type \(C_{ij}\). Once again, if \(w\) finds that the service \((s, v)\) is included in one of these \(C_{ij}\), then it will pass on the request to \(w_i\), who will start over.

Now, the words “can offer” are ambiguous. Clearly, if exactly a requested service is available, then it can be offered. On the other hand, there are often subtle differences between versions that are not important in most cases. If anything, there might be even a better version of what has been requested (perhaps faster, or more reliable).

In order for such approximations to take place, more complex orders are required on the version space of a service, that allow one to clearly distinguish between successive revisions, where one supersedes the other and subsumes all of the previous one’s functionality, from the situation of different variants offering quite different functionalities. This area is currently being studied by the authors.

5 CREATING AND MODIFYING WOS SERVERS

Defining a WOS server is essentially defining one version of the WOS. In other words, each catalog that is offered by a server \(w\) can be considered to be a version of the WOS. Effectively, we have a two-level version system, where collectively the versions of the services define an aggregate version (of the WOS).

Once catalogs are available, then new catalogs can be created by adding new functionality, or by applying connectors to existing catalogs. For example, the union and intersection operations can be applied to catalogs, assuming that the dependency constraints can be satisfied.

6 CONCLUSIONS

We have outlined how the concepts of software configuration management can be applied to define the WOS, and what new features are needed: locality of all identifiers, and the notion of version catalog, a sort of aggregate version.

What is still unclear is how the orders in the version spaces of individual components can be used, both for requesting service from the WOS, as well as in defining new and modifying existing catalogs.
7 REFERENCES


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 Senior Lecturer at The University of New South Wales (Sydney, Australia). 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.