Freeing Cooperation From Servers Tyranny*

Davide Balzarotti†, Carlo Ghezzi‡, and Mattia Monga‡

‡Politecnico di Milano
Dip. di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
I 20133 Milan, Italy

†CEFRIEL
Via Fucini,2
I 20133 Milan, Italy

Abstract. This paper deals with computer supported cooperative work in the context of untethered scenarios typical of mobile environments. The scenario envisions a number of homogeneous peers that are able to provide the same services, disconnect frequently from the net, and perform part of their work while disconnected. The application we choose is Configuration Management (CM), a critical cooperative activity occurring in software development. We discuss an implementation of a configuration management tool in a peer-to-peer setting, evaluate our solution with respect to other systems, and draw conclusions for future development.

1 Motivation

A computer aided cooperative work effort typically deals with in the production of a number of software artifacts. Artifacts are parceled among collaborators. The best application of the principle of division of labor would require that only a worker manipulates each artifact. Unfortunately, this is never true. In general, for each item we can identify the role of an *owner,* i.e., the individual who has created the artifact or who has the duty of carrying out the work on it. However, there are typically other workers who need or want to see or manipulate items that are not under their control, i.e., artifacts they do not own.

A classical solution to coordination of people work relies on the existence of a shared *repository.* Shared artifacts are stored in the repository and if one wants to work on them, one has to *check-out* the needed artifact from the repository. When the work session is over the artifact has to be *checked-in* the repository again. According to this approach the repository becomes the centralized mean of coordination among workers, thus check out and check in operations can be controlled by enforcing agreed policies that ensure consistency of the collaborative work.

In order to meet its requirements the repository has to be accessible by all the workers, thus the traditional architecture is based on a number of servers that provide the "repository service" to the client nodes that are in charge of the work. This architecture relies on two assumptions:

* This work is supported by a grant from Microsoft Research, Cambridge (UK) and COMPAQ.

1. *no off-line cooperation:* check out and check in operations are performed only while a network connection is available;
2. *servers always alive:* repository servers are always available on line when check in and check out operations are needed.

The centralized repository assumption seems to be too restrictive in several current scenarios based on mobility and continuous evolution. In fact, the recent advances in the area of wireless networks and the popularity of powerful mobile computing devices, such as laptops, PDAs, or even mobile phones, is fostering the diffusion of a new form of distributed computing usually called *mobile computing*. In this scenario, where users connect to the network from arbitrary locations and typically they are not permanently connected. In a fully nomadic scenario, no fixed network topology can be assumed. Machine disconnection is not an exceptional case, but the normal way of operating. The pure client-server paradigm, where some machines play the role of service providers for other machines, appears to be unsuitable to enable the required intrinsic dynamism. The main disadvantage of client-server systems is that they are like little "solar systems": the entire application orbits around the main server stars. When servers are not reachable, the entire system is just a dead cold set of asteroids. As far as cooperative work is concerned, this means that the entire service is blocked, until servers arise again to bring new life in the collaborators' work.

Instead we are going to propose an architecture where nodes might freely accomplish their computations *off-line*. Of course, this requires some form of reconciliation with the *on-line* part of the environment when disconnected machines later rejoin the network of nodes. To achieve this goal, we claim that the presence of two classes of computational elements, namely, clients and servers, is a weak point: a service cannot be exploited every time servers are not available. Instead, when the topology of the network environment is not known *a priori, peer-to-peer* settings where all nodes are peers, i.e. they are functionally equivalent and any could provide services to any other [1] may provide the following advantages:

- *absence tolerance:* the absence of a single peer, because of a fault or a voluntary disconnection, can be often compensated by the presence of other peers;
- *bandwidth economy:* network links towards servers are typically the bottlenecks of client-server environments, in particular if the number of nodes is high. In a peer-to-peer setting the network topology can be conceptually considered as a complete graph, and the traffic is more homogeneously distributed on all edges;
- *ease of configuration:* because in theory each peer acts both as a client and as a server, it can customize the services it provides according some commonly accepted protocol, without requiring a centralized supervision;
- *efficient use of resources:* popular resources (data and services) can be easily replicated on several peers. Conversely, unused or obsolete resources could be eliminated by the a decision of the subset of peers which was interested in them.

These advantages are available at the cost of the loss of the centralized control. Nonetheless, it is possible and convenient to build middleware components that provide primitives aimed at supplying a common framework where coordination and cooperation of peers is facilitated, and the changing network topology is hidden. In this paper we assume the existence of such a middleware. In particular (see Section 3.1),we base our work on PeerWare, a middleware suitable for peer-to-peer settings developed at Politecnico di Milano [2, 3].

This paper is organized as follows: in Section 2 we describe the requirements for what we want from our support tool, in Section 3 we describe the architectural structure of the tool, and finally, in Section 5 we draw some conclusions.

2 Requirements for the support tool

As an example of computer supported cooperative work we focus on software development. Thanks to the increasing availability of distributed computational infrastructures, software production is often dispersed among geographically distant locations, and software processes become necessarily network aware cooperative processes. Software development environments, however, are still far from supporting these new forms of virtual workgroups through specific network-aware services [4]. In particular, consider the case of configuration management (CM) tools. Software processes are typically supported by CM tools [5] that help developers to keep consistent their work, and, despite of dynamic nature of software teams, these are typically client-server applications [6]. As stated, the availability of a repository machine is critical, because without it no check-out or check-in operations are permitted. This is too restrictive because it assumes the availability of the network infrastructure even in the frequent case that no concurrent work is done on a particular item. It is perfectly reasonable and desirable that one could check in a file which is under his/her control if no other developers want to manipulate it. Similarly, check out operations can be performed also when the latest version of an artifact is available somewhere, not necessarily on repository servers, but for example on the local file system because no one has checked in a newer version. However, these operations, while performed in unthetered or even off-line mode, should be fully compliant with the cooperative system policies. Thus, an off-line check in should version the artifact and, for instance, satisfy a given set of requirements.

The main point is that in a highly mobile settings, disconnected work is no more an exceptional case. A software development team, provided with laptops and some kind of wireless connection, sets up impromptu meetings during which they can, for example, correct bugs on the fly and merge the patches in the baseline. Developers may wish to check out the modules they need also when the owners are not connected to them. This, of course, requires the system to provide support for some caching policy. Similarly, check in should be a transparent operation, which should not require knowledge of who is on-line when the operation is executed. A check in request should be executed asynchronously when the owner of the item becomes available on-line. Finally, when new versions of

configuration items that are under one's control become available on-line, a notification should be submitted to all interested peers, to enable them to keep an updated view of the system.

Consider the following scenario: a developer D wants to modify a source file f owned by Z, but Z is currently off-line. However, this is not a problem, because X, who is available on-line, has a recent version of f that can be downloaded by D. In the meanwhile, Z is working (off-line) on f and she checks in a new version f' of it. When Z reconnects herself with the rest of the system, a notification of the existence of f' is submitted to the peers. If X now asks to check out f, the new version f' is downloaded, since the previously locally cached copy is no more valid. If D decides to check in his modified version of f, a conflict arises, which may be solved by a manual merge of the two independently developed modifications of the same module.

Summing up our requirements, the configuration management tool should provide the following features:

1. check-out: every work session starts with this operation. Configuration items should be accessible also when owners are not connected, thanks to suitable caching policies;
2. check-in: every work session ends with this operation. It should be possible to check in items at any moment. However, the actual check-in is physically carried out only when the owner is available. Since concurrent changes of a configuration item are possible, this may generate conflicts. Conflict resolutions, which may imply some manual merge, is performed when the owner is on-line. Off-line and on-line check in operations are subjected to the same policy rules.
3. change notification: when a peer joins the network, it notifies the changes made to its own items since it last left off to all interested peers. In this way, any cached copies kept by such peers become invalid.

3 An architectural view

In order to implement the requirements described in Section 2, the configuration items repository must be distributed among all process participants, as showed in Figure 1. In a process with n participants, the global repository R is composed by the union of the local repositories R_i

$$R = \bigcup_{i=1}^{n} R_i$$

Two different architectural choices are feasible:

$$\bigcap_{i=1}^{n} R_i = \emptyset \tag{1}$$

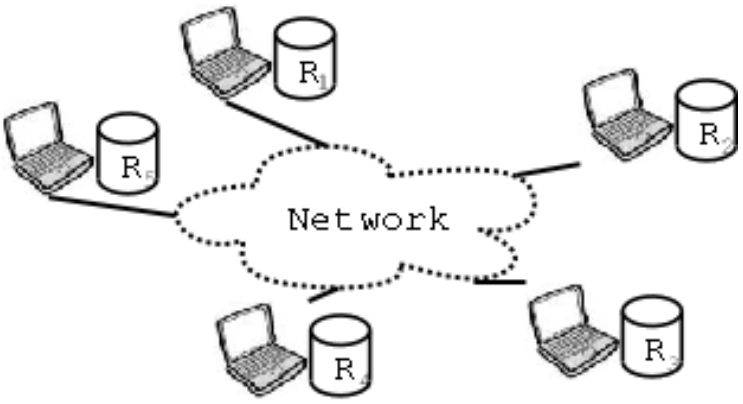


Fig. 1. Distributed repository of configuration items

$$\bigcap_{i=1}^{n} R_i \neq \emptyset \tag{2}$$

The choice (1) gives a system with no replicated information. This solution allows efficient implementations (see for example DVS [7–9, 6]) and does not introduce the risk of getting inconsistent replicated information. However, items can be accessed only if the unique host that provides them is on-line, and this does not satisfy our requirements about check-out.

In our system we preferred the choice (2) because by replicating information, it enables cooperation also when some nodes are not available on-line. However, more machinery to compose conflicts among different versions of configuration items is needed. In order to settle conflicts we adopt a strategy similar to the one used in the management of the distributed database of the Domain Name System [10], in which the data regarding associations between IP numbers and host names are replicated on several DNS servers. Each DNS server records some associations known with certainty (*authoritative* associations) and some others simply as remembered form previous accesses (*cached* associations). Whenever a DNS server gets a request for a host for which it cannot give an authoritative answer or that is not contained in its cache, it queries the network, possibly ending up asking the authoritative server, who knows the correct answer.

In our system each peer is *authoritative* for the configuration items it owns, and its copy of such items is the "master" copy. Every check-in of a new version becomes definitive only if it is authorized by the authoritative peer. If a peer X wants to check-in a document whose authoritative peer is A ($\neq X$) two cases may occur:

1. A is reachable by X: a check-in proposal is notified to A. A can reject the proposal or commit to making it persistent in its local part of the repository as a new master copy;
2. A is disconnected from X: a check-in proposal is recorded in the local part of the repository hosted by X. When A becomes available, the proposal is notified to it. A can reject the proposal or commit to making it persistent in its local repository as a new master copy. If A has an item newer than the one proposed by X, a conflict arises. Similarly, other concurrently pending check-in requests generate conflicts. Conflicts must be managed by merging the various change requests, and then issuing a new check-in proposal.

When a peer X wants to check-out a document d whose authoritative peer is A ($\neq X$) two cases may occur:

1. d is present in X's the local repository and the copy is *valid,* that is, no newer versions were notified to X. The check-out operation boils down to getting a copy of d;
2. d is not present in the local part of repository under control of X: a network search is issued to retrieve a valid copy. If no valid copy is found, the check-out operation fails. Notice, however, that it may also happen that an invalid copy is found, but the authoritative peer for the searched item is off-line. This may happen when the authoritative peer gets on-line, notifies all interested peer that a new version is available for a given item, and then immediately gets disconnected from the network. In such a case, the cached versions of the item become invalid, but at the same time the most recent version of the item is unaccessible. We decided that, in this case, the check-out operation retries the invalid copy.

Finally, when a peer enters the community of peers, a reconciliation step is performed. More specifically, when X gets connected, for each item i for which X is the authority, X notifies all interested peers if a newer version of i is made available. In such a case, the locally cached copies of peers that are not authoritative for the item become *invalid.*

In the next Section we sketch the middleware we used to implement the operations we described here to support distributed configuration management.

3.1 The underlying middleware

PeerWare [2, 3] provides the abstraction of a *global virtual data structure* (GVDS), built out of the local data structures contributed by each peer. PeerWare takes care of reconfiguring dynamically the view of the global data structure as perceived by a given user, according to the connectivity state. The data structure managed by PeerWare is organized as a graph composed of nodes and documents, collectively referred to as items. Nodes are essentially containers of items, and are meant to be used to structure and classify the documents managed through the middleware.

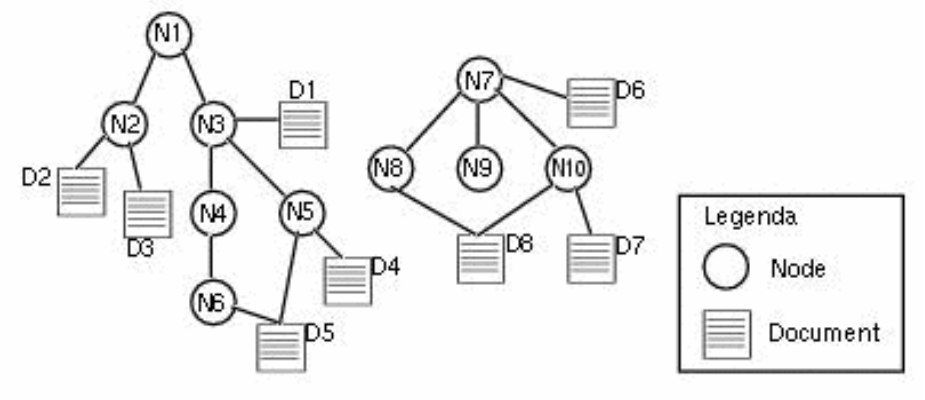


Fig. 2. An example of the PeerWare data structure managed by a peer

This means that nodes are structured in a forest of trees, with distinct roots, which most likely represent different perspectives on the documents contained into the data structure. For instance, one could have an "GNU/Linux projects" tree, a "Latex papers" tree, and so on. Within this graph, each node is linked to at most one parent node and may contain different children nodes (see for example Figure 2). Conversely, stand-alone documents are forbidden; documents are linked to at least one parent node and do not have children. Hence, a document may be contained in multiple nodes. As for labels, two nodes may have the same label, as long as they are not both roots and are not directly contained into the same node.

At any time, the local data structures held by the peers connected to PeerWare are made available to the other peers as part of the global virtual data structure managed (GVDS) by PeerWare. This GVDS has the same structure of the local data structure and its content is obtained by "superimposing" all the local data structures belonging to the peers currently connected, as shown in Figure 3.

Changes in connectivity among peers determine changes in the content of the global data structure constituting the GVDS, as new local data structures may become available or disappear. Nevertheless, the reconfiguration taking place behind the scenes is completely hidden to the peers accessing the GVDS, which need only to be aware of the fact that its content and structure is allowed to change over time.

There is a clear distinction between operations performed on the PeerWare local data structure and on the whole GVDS. While hiding this difference would provide an elegant uniformity to the model, it may also hide the fundamental difference between local and remote effects of the operations [11]. In particular the operations for creating or destroying a node ($createNode(node, parent)$, $removeNode(node)$), for inserting or removing a document ($placeIn(node, document)$, $removeFrom(node, document)$), and for publishing an event occurred on an item

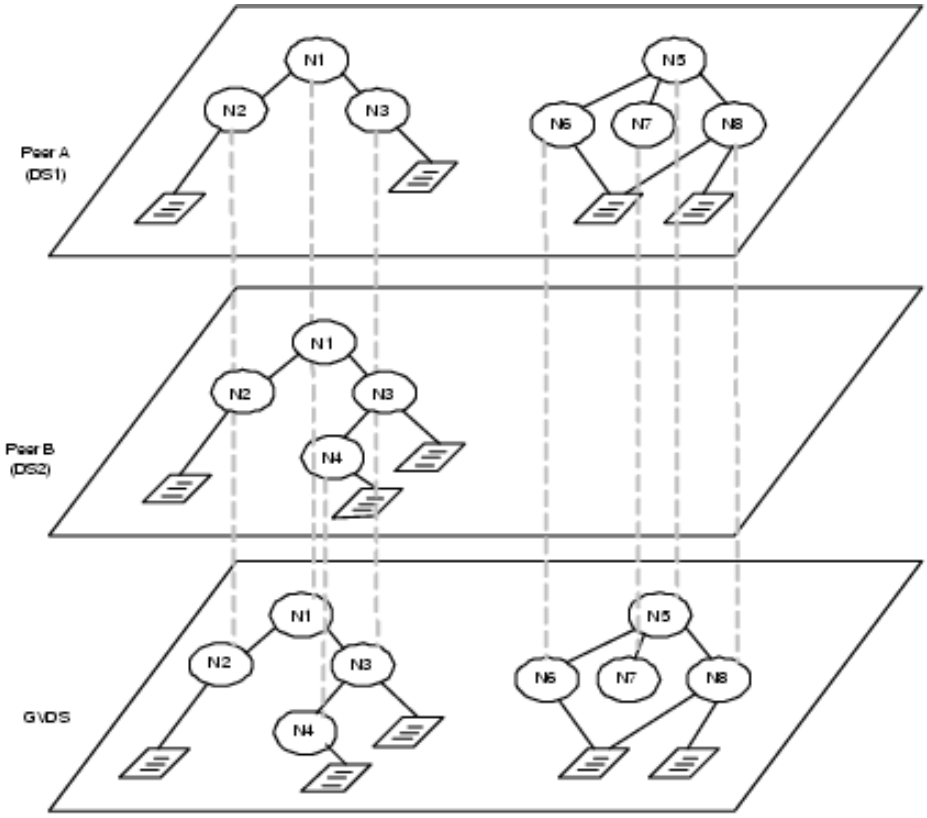


Fig. 3. An example of the global virtual data structure managed by PeerWare

($publish(event, item)$) are defined only on the local data structure. PeerWare provides three operations that can be performed both on the local and the global data structures:

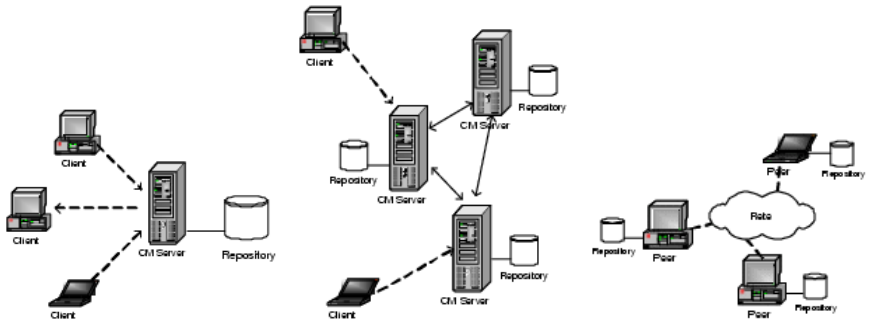
1. $I = execute(F_N, F_D, a)$. An action a is performed on all documents – contained in nodes whose name matches the filter F_N – that match the filter F_D. The matching set of documents I, affected by a is returned to the caller.
2. $subscribe(F_N, F_D, F_E, a)$. Allows a peer to subscribe to the occurrence of an event matching the event filter F_E and being published within the projection of the data structure identified by the filters F_N and F_D. When the event occurs the action a is executed locally to the caller.
3. $I = execSubscribe(F_N, F_D, F_E, a_e, a_s)$. Executes an arbitrary action a_e on the projection of the data structure identified by F_N and F_D, similarly to *execute*. Also, in the same atomic step, it subscribes for events that match F_E, and occur within the same projection, by specifying the action a_s that must be executed locally to the caller, when one of such events occurs.

The semantics of a global operation can be regarded as being equivalent to a distributed execution of the corresponding operation on the local data structures of the peers currently connected. While as far as concerns local operations atomicity can be assumed, this is an impractical assumption in a distributed setting. Hence, the global operations do not provide any guarantee about global atomicity, and they guarantee only that the execution of the corresponding operations on each local data structure is correctly serialized (i.e., it is executed atomically on each local data structure).

The operations provided by PeerWare together with a publish/subscribe engine on which PeerWare itself relies on (the distributed event dispatcher JEDI, see [12]) build the framework needed to implement the configuration management operations described in Section 3. In particular, by using PeerWare we can abstract from the actual network topology and perform actions on *on-line* items.

4 Related work

Despite the recent evolution of Software Configuration Management systems, most CM systems are still founded on a centralized architecture where both the application and the repository are stored in the same physical location.
Figure 4(a) shows the traditional Client-Server architecture that is used in most conventional CM systems (CVS [13] is probably the best known). Nowadays, software production is becoming a more and more distributed activity where the project teams are physically dispersed over a great number of locations. This has fostered the design of systems that provide the distribution of the repository over multiple sites. The solution usually adopted in the distributed CM systems is shown in Figure 4(b): the repository is broken up among geographically distinct servers, allowing the distribution of the data next to the actors of the software process. However, from the user point of view not much is changed because, like



(a) (b) (c)

Fig. 4. CM system architectures

in the previous architecture, they must connect to a server to perform either a check-in or a check-out operation. As examples of systems built over this architecture we consider two different products : ClearCase Multisite and DVS.

Rational ClearCase Multisite [14] is a commercial product that supports parallel software development with automated replication of project database. With Multisite, each location has a copy (*replica*) of the repository and, at any time, a site can propagate the changes made in its particular replica to other sites. Nevertheless, each object is assigned a master replica and in general an object can be modified only at its master replica. To avoid this restriction Multisite uses branches. Each branch can have a different master and since the branches of an element are independent, changes made in different sites do not conflict. The concept of master is like our concept of authority; however, the access policy provided by Multisite is too restrictive for our scenario.

DVS [7] is a simple research system that allows one to distribute the CM repository over the network, but it does not allow the replication of the information. Even though the absence of replication contrasts with our assumptions, it is interesting to make an architectural comparison with DVS because it also makes a clear distinction between the CM application and the underlying middleware. In fact, DVS has been implemented on top of NUCM [8] (Network-Unified Configuration Management), whereas our system is built on top of PeerWare. NUCM defines a generic distributed repository and provides a policy-neutral interface to realize CM systems. PeerWare is a general-purpose peer-to-peer middleware, whereas NUCM is focused on the CM system creation and provides ad hoc functionalities.

To make the comparison clear, our solution is sketched in Figure 4(c). It is a pure peer-to-peer architecture where there are no servers and the whole repository is directly distributed over the users' devices. We claim that such an

architecture is a more suitable solution for evolvable scenarios where topology is highly dynamic.

5 Conclusions and future work

In this paper we have discussed how to build a tool supporting cooperation to a networked team, without relying on the existence of centralized repository servers. We do not want to restrict the use of the system to the scenarios where repository servers are always available on line. Instead, we propose the use of a suitable peer-to-peer middleware which provides the abstraction of a global virtual data structure. This is a data structure parceled among all peers, but it can be searched and modified transparently, without knowing the actual network topology.

The use of this abstraction enabled us to design a configuration management tool which is especially oriented to supporting scenarios in which users' connectivity to the network can change dynamically. This is achieved by exploiting the global virtual data structure as the artifact repository. Our solution overcomes the intrinsic problems of client-server architectures, which are clearly not suitable for scenarios where the absence of a host is not an exceptional case, but rather the normal case. Along the same line, repositories based on distributed file systems expose the system to failures when a server is unavailable.

Our solution is based on caching copies, and then making them available for use even if the hosts that own them are disconnected. The outcome is a genuine peer-to-peer architecture, where any on-line machine can in principle replicate the unavailable resources. We pay, of course, for this advantage in terms of a harder coordination effort.

The approach we described in this paper is currently being implemented as part of our current efforts in the provision of a suite of software process support tools well suited for educational environments in which students are equipped with mobile devices.

References

1. A. Oram, ed., *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*. O'Reilly & Associates, first ed., Mar. 2001.
2. G. Cugola and G. P. Picco, "Peerware: Core middleware support for peer-to-peer and mobile systems." submitted for publication, 2001.
3. F. Bardelli and M. Cesarini, "Peerware: un middleware per applicazioni mobili e peer-to-peer," Master's thesis, Politecnico di Milano, 2001.
4. J. Estublier, "Software configuration management: A road map," in *The Future of Software Engineering* (A. Finkelstein, ed.), ACM Press, May 2000.
5. E. H. Bersoff, "Elements of software configuration management," *Software Engineering*, vol. 10, no. 1, pp. 79–87, 1984.
6. A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. L. Wolf, "A testbed for configuration management policy programming," *Transaction on Software Engineering*, vol. 28, pp. 79–99, Jan. 2002.

7. A. Carzaniga, "Design and implementation of a distributed versioning system," tech. rep., Politecnico di Milano, Oct. 1998.
8. A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. L. Wolf, "A reusable, distributed repository for configuration management policy programming," tech. rep., University of Colorado, Boulder CO 80309 USA, Oct. 1998.
9. A. van der Hoek, D. Heimbigner, and A. L. Wolf, "A generic, peer-to-peer repository for distributed configuration managemenet," in *18th International Conference on Software Engineering*, (Berlin - Heidelberg - New York), p. 308, Springer, Mar. 1996.
10. P. Mockapetris, "Rfc 1035 (standard: Std 13) domain names–implementation and specification," tech. rep., Internet Engineering Task Force, November 1987.
11. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Mobile Object Systems*, vol. 1222 of *Lecture Notes in Computer Science*, pp. 49–64, Springer-Verlag, Berlin, 1997.
12. G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an event-based infrastructure to develop complex distributed systems," in *ICSE98 proceedings*, (Kyoto (Japan)), April 1998.
13. "Concurrent versions system." `http://www.cvshome.org/`.
14. Rational Software Corporation, Maguire Road Lexington, Massachusetts 02421, *ClearCase MultiSite Manual (release 4.0 or later)*, 2000.