Supporting Cooperative Software Processes in a Decentralized and Nomadic World

Davide Balzarotti, Carlo Ghezzi, *Fellow, IEEE*, and Mattia Monga

***Abstract*—Recent advances in wireless networks enable decentralized cooperative and nomadic work scenarios where mobile users can interact in performing some tasks without being permanently online. Scenarios where connectivity is transient and the network topology may change dynamically are considered. Connectivity among nodes does not require the support offered by a permanent infrastructure but may rely on ad hoc networking facilities. In this paper, a scenario in which a nomadic group of software engineers cooperate in developing an application is investigated. The proposed solution, however, is not software process specific but holds for other cases where shared documents are developed cooperatively by a number of interacting nomadic partners. Support tools for these groups are normally based on a client–server architecture, which appears to be unsuitable in highly dynamic environments. Peer-to-peer solutions, which do not rely on services provided by centralized servers, look more promising. This paper presents a fully decentralized cooperative infrastructure centered around peer-to-peer versioning system (PeerVerSy), a configuration management tool based on a peer-to-peer architecture, which supports cooperative services even when some of the collaborating nodes are offline. Some preliminary experiences gained from its use in a teaching environment are also discussed.**

***Index Terms*—Computer-supported cooperative work (CSCW), cooperative software development, nomadic computing, versioning.**

I. Introduction

NETWORKED distributed systems are increasingly becoming the main infrastructures over which people cooperate in their daily work. Cooperative work, which traditionally required people to interact in a shared physical space, has been shifting toward being computer supported. This gave strong impetus to research on foundations, methodologies, and technologies for computer-supported cooperative work (CSCW) [1]. Presently, cooperative work is supported on both local and wide-area-based infrastructures [2]–[4].

Cooperation is usually achieved by defining a workflow that defines the coordination of the various actors involved in the work [5], [6]. A workflow engine orchestrates the cooperation among the actors (software tools and components, and humans). A repository of artifacts is also often available in CSCW environments to keep a shared global consolidated view of the system state. Both the workflow engine and the artifact repository are traditionally provided by a CSCW server [7].

The software process is a particular kind of a human-intensive cooperative process that can be supported by a computer-based environment. Software engineers cooperate in this environment by developing parts of an application in different stages (such as requirements specification, design, implementation, and testing), by managing their activities, and by synchronizing their work. A considerable amount of research has been done over the past 20 years to study software processes and the way they may be supported [8]. In particular, starting from the seminal work in [9], research focused on understanding, specifying, validating, implementing, and improving software process. Much work was also directed to provide automated support to software process through process-centered software engineering environments [10].

In this paper, we concentrate on a specific aspect that involves cooperation in the software process, i.e., configuration management. Configuration management is perhaps the activity where automated process support has proved to be most effective [11].

Configuration management is a very critical activity in software development. It is responsible for keeping the development of software artifacts orderly and managed. Not surprisingly, it is considered by process improvement methodologies, like the capabilities maturity model, as one of the key practices that software development organizations should establish in their improvement strategies [12].

The focus of this paper, however, is not configuration management per se. Rather, we wish to investigate how certain assumptions on the distributed infrastructure affect the way cooperative support in managing artifacts is provided. Traditionally, in fact, configuration management tools follow a client–server architecture. The server is responsible for the management of the artifacts. This approach is possible and suitable in all cases where a reliable and permanent network infrastructure is available to connect the participating nodes. In this paper, instead, we investigate how configuration management can be supported through a much looser peer-to-peer infrastructure. In fact, it is quite common that small groups of developers cooperate to build software products by forming highly dynamic virtual communities in which people change frequently their connectivity status, i.e., not always being able to access the network infrastructure. In other words, we have to cope with a network of peers each of which contributes to the overall logical artifact repository with the artifacts it owns, and peers are not always online. Moreover, the network connection is intrinsically intermittent, as in the case of wireless

Manuscript received October 17, 2005; revised May 3, 2006 and July 24, 2006. This paper was recommended by Guest Editor G. Cabri.

D. Balzarotti is with the Department of Computer Science, University of California, Santa Barbara, CA 93106 USA.

C. Ghezzi is with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy.

M. Monga is with the Dipartimento di Informatica e Comunicazione, Università Degli Studi di Milano, 20135 Milano, Italy (e-mail: mattia.monga@unimi.it).

Color versions of Figs. 1–5 are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCA.2006.883165

connections. Peers may dynamically join and leave the virtual ad hoc community, as it typically happens in mobile scenarios. They join it in impromptu meetings, where they exchange data and synchronize their work. Each peer, however, should continue to provide its functionality to the user even when it is in a disconnected stage. The support infrastructure should handle connections and disconnections in a seamless fashion. Coping with intermittent connectivity is perceived as an important feature of modern configuration management systems, as witnessed by the fact that all recently developed products (for example, SubVersion [13] and Monotone [14], see Section III for further details and examples) provide some support to disconnected operations.

This paper originated from MOTION, a research project that was funded by the EU in the 5th Framework Programme [15]. MOTION is an e-work platform that supports collaboration and distributed working methods for cooperative product development and business management. Within MOTION, we developed a middleware platform (PeerWare [16]) supporting peer-to-peer document sharing. We then focused our research on supporting cooperative software processes in mobile and wireless environments. In particular, we investigated how software configuration management can be provided in a peer-to-peer fashion, and we developed the PEER-to-peer VERsioning SYstem (PeerVerSy) tool to support the proposed approach [17]. These ideas were experimented in the Virtual Campus project sponsored by Microsoft Research at Politecnico di Milano [18], [19]. The Virtual Campus project developed an e-learning platform suitable for collaborative work. It gave us the opportunity of experimenting with our PeerVerSy prototype since some collaborative sessions involved small teams of students developing software.

Although the research we report here focuses on decentralized cooperation in software development tasks, our approach has a much wider potential applicability. It applies to scenarios where shared artifacts or documents are developed, manipulated, and modified cooperatively [20] without the permanent global support provided by servers but rather in a peer-to-peer setting.

This paper is organized as follows. In Section II, we describe the requirements a wireless and mobile environment imposes on a cooperative support tool. In Section III, we discuss the state of the art and related work. In Section IV, we describe our distributed algorithm in more detail. The experiments done to evaluate our approach are described in Section V. First, we provide a quantitative assessment based on a simulated performance model. Second, we introduce the results we obtained by experimenting with a prototype tool in a teaching environment. Third, we sketch the current work that focuses on implementing our algorithm as a set of primitives on top of the SubVersion system. Finally, in Section VI, we draw some conclusions.

II. Requirements for the Support Tool

In this section, we develop the requirements for a tool supporting cooperation among nomadic users by focusing on cooperation in software development. As we mentioned earlier, our approach applies to other cases as well, where people cooperate in developing a set of shared documents.

Software development can be viewed as a cooperative process where software engineers exchange and jointly manipulate software artifacts, such as requirements documents, test data, module interfaces, module bodies, etc. It is crucial to manage such artifacts in a flexible and efficient manner. This is exactly the purpose of configuration management tools [11], [21], which constitute the key logical components of a process-centered support environment.

In recent years, software processes became increasingly decentralized and distributed. However, software development environments are still far from supporting these new forms of highly dynamic and distributed virtual workgroups through specific network-based services [22]. In particular (see also Section III), the configuration management tools used by developers to keep their work consistent are still based on a client–server architecture [23], which requires repository servers to be always reachable online. This assumption is too restrictive in many cases because it assumes the network infrastructure to be always available to support check-out and check-in operations even in the frequent case that no concurrent work is done on a particular item.

Configuration management tools support collaboration by regulating the exchange and sharing of software artifacts. In general, for each item, we can identify the role of the "owner," i.e., the individual who has created the artifact or who is responsible for controlling the status of the artifact. Other individuals may want to see, or need to manipulate, artifacts they do not own.

A classical solution to coordination of people work relies on accessing a shared "repository." Shared artifacts are stored in the repository, and whenever one needs to work on them, he or she has to "check-out" the 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 space of coordination among workers, and check-out/check-in operations can be controlled by enforcing agreement upon policies to ensure consistency of the collaborative work.

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

1) *No offline cooperative work is permitted:* Check-out and check-in operations can be performed only while a network connection is available to support access to servers.
2) *Servers are always alive:* Repository servers are always available online when check-in and check-out operations are needed.

These assumptions are sensible when people work mainly in their office, i.e., by using their workstation that is permanently connected to the network. However, it is fairly common today that mobile programmers work with their laptops, which only have an intermittent network connection. Moreover, they would

like to take advantage of the popular existing wireless facilities, for example, to be able to establish a collaborative session during a meeting without setting up a client–server infrastructure.

In a fully nomadic scenario, no fixed network topology can be assumed. Ad hoc networking solutions may be used to establish communication among nodes that may dynamically join and leave the community. Machine disconnection is not an exceptional case, but the normal way of operating: even by assuming network connections to be reliable, some nodes may deliberately choose to become offline.

The pure client–server paradigm, where some machines play the role of service providers for other machines, cannot support the required intrinsic dynamism of this situation. 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 collaborative universe.

Instead, a nomadic scenario requires an architecture where users may be supported in their own work offline. Users are equipped with personal computational devices called network nodes (the terms "user" and "node" will be used interchangeably in the sequel). When disconnected nodes later rejoin the network, some form of reconciliation with the online part of the environment is necessary. In a decentralized and mobile world, a peer-to-peer setting where all nodes are peers, i.e., they are functionally equivalent and anyone could provide services to any other [24], looks promising since it is able to achieve the following advantages.

- *Absence tolerance:* The absence of a single peer, because of a fault or a voluntary disconnection, can be compensated by the presence of other peers.
- *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 to 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 decision of the subset of peers that was interested in them.
- *Bandwidth economy:* Network links toward servers are typically the bottlenecks of client–server environments, notably in cases where the number of nodes is high. In a peer-to-peer setting, traffic is more homogeneously distributed on all edges, thus enabling a fair load balancing, with respect to the bottlenecks implicit in a client–server architecture.

In this paper, we explore how configuration management can be accomplished in a highly mobile setting where disconnected work frequently occurs. We were interested in an architecture that was able to tolerate the absence (isolation) of any node, possibly without the need of heavy configuration steps. Therefore, we adopted a decentralized peer-to-peer architecture with no centralized servers, where each group member keeps a copy of a set of artifacts on his or her workstation. In particular, each member keeps a copy of the artifacts he or she is responsible for ("owned artifacts") as well as a copy of other artifacts (owned by other peers) he or she has checked out earlier.

Members of the software development team, provided with laptops and some kind of wireless connection, may set 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 artifacts they need also when the owners are not connected to them but rather other peers that possess a copy of the artifact are available. This, of course, requires that other nodes that checked out the artifact provide support for a caching policy. Similarly, check-in should be a transparent operation, which should not require knowledge of who is online when the operation is executed. A check-in is materialized in the workspace of the artifact's machine when it becomes available online. When new versions of configuration items (artifacts) that are under one's control become available online, a notification is submitted to all interested peers to enable them to get an updated copy.

Consider the following scenario. A developer D wants to modify a source file f owned by Z, but Z is currently offline. However, this is not a problem because X, who is available online, has a recent version of f that can be downloaded by D. In the meantime, Z is working (offline) on f, and she checks in a new version f'. When Z reconnects herself with the rest of the system, a notification of the existence of f' is submitted to the other peers. Now, both X and D are going to download the new version of the document from Z in order to keep their local repository up to date.

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 (artifacts) should be accessible also when owners are not connected thanks to suitable caching policies through other peers.
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 online. Since concurrent changes of a configuration item are possible, this may generate conflicts. Conflict resolutions and merge are performed when the owner is online. Offline and online check-in operations are subject 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 are marked as obsolete.

III. Current Tools

Configuration management systems are still mainly based on a client–server architecture where both the application and the repository are stored in the same physical location.

Fig. 1(a) shows the traditional solution that is used in most conventional configuration management systems, such as the

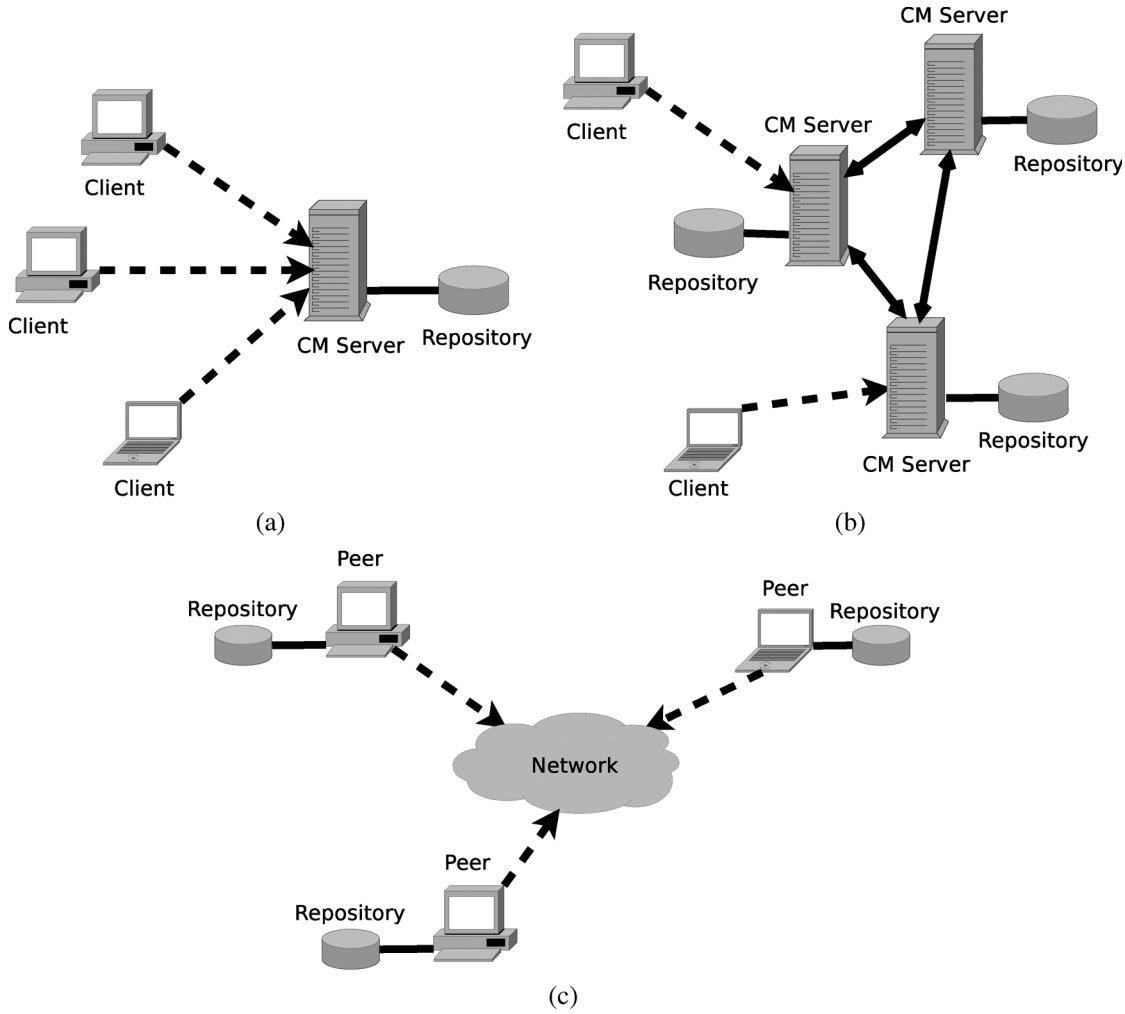

Fig. 1. Configuration management system architectures.

widely used concurrent versions system (CVS) [25]. However, nowadays, software production is becoming a more and more distributed activity where the project teams are physically dispersed over many locations. This has fostered the design of systems that provide distribution of the repository over multiple sites. The solution usually adopted in distributed configuration management systems is shown in Fig. 1(b): the repository is distributed over geographically distinct servers, enabling the distribution of data next to the actors of the software process. However, from the user point of view, not much is changed because, like 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 tools, namely: 1) ClearCase MultiSite and 2) DVS.

Rational ClearCase MultiSite [26] is a commercial product that supports parallel software development with automated replication of the 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 to a master replica, and in general, an object can be modified only by touching its master replica. To relax 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. MultiSite does not provide specific support to nomadic users whose online availability varies in an unpredictable fashion. It requires the background infrastructure of interconnected repositories to be accessible by nodes to support any cooperative action.

DVS [27] is a research system that allows one to distribute the configuration management repository over the network, but it does not support the replication of information. Although the absence of replication contrasts with our assumptions, our system and DVS adopt a similar overall architecture, where configuration management features are implemented on top of a middleware layer. In fact, DVS has been implemented on top of network-unified configuration management (NUCM) [28]. NUCM defines a generic distributed repository and provides a policy-neutral interface to realize configuration management systems.

Although these approaches satisfy the requirements of geographically distributed enterprises, they are not suitable for small groups of mobile developers. The next step lies in abandoning the central repository and adopting a purely peer-to-peer distributed architecture, as shown in Fig. 1(c).

BitKeeper [29] adopts a solution that is halfway between the multiple-server architecture and a fully distributed one. The basic idea is that each user can clone a piece of repository in the local machine such that the work may proceed without the necessity of being connected to the main repository. Indeed, each replica is a fully functioning repository, and users can freely check-in and check-out documents also while they are disconnected. BitKeeper includes commands to synchronize the contents of a replica and to propagate changes from one repository to another. Such primitives foster the creation of a hierarchical structure (a tree), where changes propagate upward from the leaves (representing user workspaces) to the root (the main repository), and up-to-date versions of artifacts propagate down in the opposite direction. Although this approach is very flexible, it is not easy to apply in a peer-to-peer setting where there is no fixed hierarchy between nodes.

An interesting tool that overcomes this restriction is Code Co-Op [30]. Code Co-Op is a distributed versioning system that replicates project data on each computer. Thus, check-ins and check-outs are possible also when people are offline. In Code Co-Op, exchange of information among members of the team occurs separately from other sources of control activities. E-mail is used to share change scripts (it is important to note, however, that e-mail is based on a fairly complex protocol that is based on a client–server architecture). During check-in, the file that has been modified locally is compared with some locally available reference version of the same file. In order for other members of the project to be able to interpret the change script, they all have to own the same reference version of the file. This means that, before a given version of a file may become a reference version, there has to be consensus among all the voting members of the project. Indeed, in Code Co-Op, users are divided into two different groups, namely: 1) voting members and 2) observers. The former group participates in the distributed consensus process, and its acknowledgment is required to promote a new reference version. Observers are not involved in such a process. Since reaching consensus could take a long time (especially when a member is offline for a while), Code Co-Op allows users to work on intermediate versions of documents. Of course, this may require a lot of work to resolve conflicts if the intermediate version will be rejected.

Open-source software development is often a special case of a distributed effort. Several large projects like the Linux kernel [31], the Apache web server [32], or the Mozilla web browser [33] involve large geographically dispersed and loosely coupled workgroups [34]. Group members normally know each other only by their e-mail addresses or IRC nickname. These projects mainly rely on server-based coordination tools because they want their collaboration to be Internet wide, and the server-based solution is the more efficient one if a team can provide a server machine (or a set of machines) that is available 24 h per day. However, in some cases, the pure server-based solution was considered to be too rigid: in fact, the Linux development project adopted the BitKeeper tool[1] because small groups of developers working on specific modules required more flexibility in the availability of the repository. As we discussed before, BitKeeper, although a server-based system, introduces some peer-to-peer features because it makes it relatively easy to duplicate subsets of a repository—where independent development could take place—while giving developers tools to keep this copy in sync with the main one.

Monotone [14] manages three different types of storage, namely: 1) a central (and remote) database; 2) a database local to every user; and 3) a working copy of the artifacts. The local database acts as the "switching point" for communication with remote databases by exploiting an interactive protocol for synchronizing data over the network: a "pull" operation copies data from a remote database to a local database; a "push" operation copies data from a local database to a remote database; and a "sync" operation copies data both directions. In each case, only data missing from the destination are copied. The presence of the local database provides some support for offline operations, but pull, push, and sync operations are possible only when the remote server is available. An evolution of the classic CVS is SubVersion [13]. It exploits the same strategy: a local database enables some offline work. SubVersion primitives were also used [35] for building a configuration management tool able to deal with mirrored repositories. These tools are especially suitable for open-source development, where it is often the case that a big part of the code base is developed by some upstream author, and a user customizes and evolves just a small fraction, while he or she wants to keep his or her branch in sync with the upstream part.

Interesting research work was done in the context of distributed file systems. For example, the Coda file system [36] supports disconnected operations by caching remote mounted partitions. However, it is server based: several clients can work on the partitions exported by a central server even when a connection to the server is not available: a reconciliation step is performed when connectivity is reestablished, but Coda does not support collaborative modification or versioning of artifacts. In general, solutions at the file system level provide support on sharing data among partially connected nodes; thus, they could, in principle, be used to build collaborative tools on top of the virtual data structure they provide. As we are going to explain in Section V-B, in our approach, we used a middleware to this end.

IV. Implementation Strategy

To support truly nomadic scenarios (as outlined in Section II), we decided to base our system on a pure peer-to-peer architecture where there are no servers and the whole repository is directly distributed over the users' devices [see Fig. 1(c)].

In designing such a system, we can distinguish two different classes of problems. The first consists of managing in an efficient and automatic way the physical distribution of data among different peers. The second class is instead related to the versioning mechanism itself and includes the problem of document replication, the management of concurrent access, and the conflict resolution.

[1]With a lot of discussion among free-software advocates, since it is licensed with closed-source conditions. A recent change in BitKeeper license has forced the Linux community to develop an open-source clone called GIT.

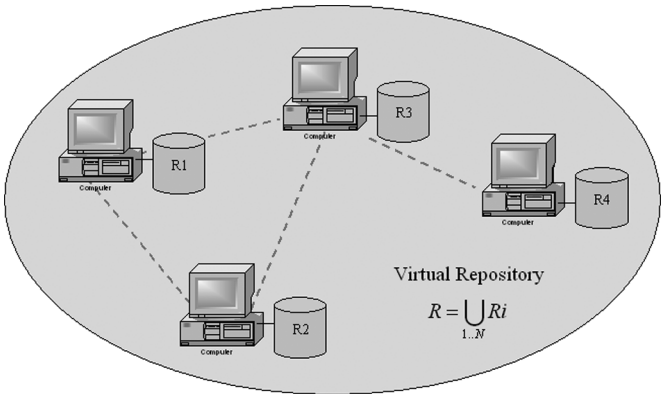

Fig. 2. Distributed repository of configuration items.

Similar to the approach adopted by DVS [27], we decided to delegate the data distribution problem to an underlying middleware layer. In this section, we analyze the versioning problems, while the middleware architecture is going to be briefly introduced in Section V-B.

In a pure peer-to-peer environment, configuration items are distributed among all process participants and stored inside their local repositories R_i, as shown in Fig. 2. The following two different architectural choices are possible:

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

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

Choice (1) yields a system with no replicated information. This solution allows for efficient implementations (see, for example, DVS [23], [27], [28], [37]) and eliminates the risk of producing inconsistent replicated information. However, items can be accessed only if the unique host that provides them is online, and this does not satisfy our requirements about the check-out operations.

In our system, we adopted choice (2) because replication enables cooperation also when some nodes are not available online. However, this is achieved at the expense of making the conflict resolution among different versions of configuration items more complex. To address this issue, we adopted a strategy similar to the one used in the management of the distributed database of the domain name system (DNS) [38], in which 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 from 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, which knows the correct answer.

The philosophy of our solution is also similar to the one adopted by Code Co-Op. However, as we discussed in Section III, its approach is based on e-mail (or other forms of file exchange such as floppy disks or FTP connections) to share scripts. This requires users to agree on a protocol and to set up the proper infrastructure. Our system aims at being more flexible: in particular, we want to be able to share items also if the sender does not know the IP address of all the possible recipients. The main advantage of the approach adopted by Code Co-Op is that its way of exchanging messages is more scalable in WAN settings than a peer-to-peer middleware. However, on a smaller scale, our solution requires far less infrastructure.

Moreover, the algorithm adopted by Co-Op is quite restrictive because for each new version of an artifact it requires a global consensus from all the participants. To mitigate the problem, Co-Op introduced the idea of voting members and observers, as we saw. Our algorithm has no such restriction because the process of promoting a new version involves only two peers, namely: 1) the author of the version and 2) the current authority of the document.

In our approach, in fact, each peer is the "authority" for a set of items, and the copy of an artifact owned by the authority is the "master copy." Peers can keep a local copy (i.e., a "replica") of the documents for which they are not the authority to allow users to work on them even when the authority is not reachable or when the peer is disconnected from the network. In fact, a user can perform both check-in and check-out operations from the local copies of a document. The only difference between the master copy and a replica is that a check-in of a new version becomes definitive and available for all users only when the authority authorizes the changes and updates the master copy.

Initially, the authority role for a document is assigned to the peer that introduces the first version of the document. The binding between authorities and peers, however, can be changed dynamically in order to improve the overall performance of the system. For instance, consider a simple scenario where the node X is the authority for the document d, but node Y was responsible for the last ten check-ins. In this case, Y is clearly the node where most of the work on the document is performed, and thus it may be reasonable to promote it to become the new authority of d.

Whenever a peer joins the online community of peers, a reconciliation step is performed. More specifically, when the node X connects, for each item i for which X is the authority, X notifies all interested peers if a newer version of i is available (the same happens when a new checked-in copy is accepted by the authority). In this case, all the peers that own a replica of i should contact the authority to update their local copies.

We can now examine what happens when a user tries to check-out or check-in a document. Suppose that peer X issues a check-out for a document d whose authoritative peer is $A\ (\neq X)$. Two cases may occur.

1) d is present in X's local repository. In such a case, the check-out operation delivers a copy of d from the local repository.
2) d is not present in X's local repository. In such a case, a network search is issued to retrieve a valid copy. If no copy is found, the check-out operation fails. Otherwise, the system creates a new local replica of the document and then checks-out a copy of the artifact from the local repository like in the previous case.

Suppose that a peer X issues a check-in request for a new version of a document whose authoritative peer is A ($\neq X$). We distinguish two cases.

1) A is reachable by X. A check-in proposal is notified to A, which can either reject it (if in the meanwhile a newer version was released) or accept the proposal to become the new "official" version of the document.
2) A is not reachable by X. A check-in proposal is stored in the local repository hosted by X, and it will then automatically notify A when the node becomes available. At that point, A can reject the proposal or commit to making it persistent in its local repository as a new master copy. If the item owned by A is newer then the one proposed by X, a conflict arises, and X's proposal will be refused. In this case, it is X's responsibility to resolve the conflict and submit a new version.

A solution based on this strategy offers several advantages.

- *Usability:* The master copy of each document is kept by the peer where the document is mostly used. This is very useful because when a user works on a document whose master copy is kept in its local repository, he can work like if he was the only developer, without the need of continuously connecting to the network to check if a new version of the document has been released by another peer.
- *Flexibility:* Each peer can control the amount of replicated data, from the simple case where nothing is replicated to the case where the whole repository is replicated on all peers. Thus, users can choose which documents have to be replicated on the ground of their needs.
- *Configurability:* The algorithm responsible for the distribution of authority roles can be easily changed, thus allowing users to choose their preferred policy. The authority of a document can be assigned both statically or dynamically. A static binding can be used, for example, when a user want to keep control on the versioning of some particularly important documents. Also, the client–server approach can be obtained as a special case, just assigning the authority of all the artifacts to a computer always connected to the network. Nevertheless, a more flexible solution consists of using some algorithm to dynamically move authority from one peer to another, trying to keep the master copy of each artifact on the node where it is likely to occur the next check-in operation (for instance, one may set a policy that assigns the authority of a document to the user responsible for the majority of the past five versions).

V. Approach Evaluation

Assessing the practical validity of a distributed cooperative tool is not an easy task because it requires a throughout analysis of many different aspects. To go beyond a mere proof-of-concept experiment, we decided to split the evaluation of our solution into three different phases.

In the first phase, we focused on analyzing the algorithm itself without taking into account any network or implementation detail. For this purpose, we designed an analytical model based on colored Petri nets to simulate the performance of our approach (in terms of conflict and check-in rates) and compare them with the reference values obtained with a traditional client–server architecture.

In the second phase of the evaluation, we implemented our algorithm in a prototype application named PeerVerSy. We then used this tool in a controlled environment, set up in the context of the Virtual Campus project. The objective, here, was twofold, namely: 1) to try whether it was possible to actually implement our algorithm in a working tool and 2) to test on the field if a totally distributed architecture can present some real advantage in a realistic experiment.

Finally, the last step would consist of testing our approach in a real-world experiment. Unfortunately, it is especially hard to design and implement experimental assessments in realistic industrial environments. Moreover, we could not use PeerVerSy in industrial settings because of the obvious limitation of a prototype application (such as the total lack of security and the inadequate reliability of the document repository). Thus, we decided to reimplement the PeerVerSy algorithm as a collection of scripts working on top of an existing versioning system.

In our evaluation, we focused on the algorithm, thus neglecting any performance assessment. In fact, bandwidth and latency depend basically on the middleware we used to dispatch data. The results of our evaluation efforts are reported hereafter.

A. Evaluation With a Model-Based Analysis

Intuitively, our protocol provides some advantages over a server-based one. Imagine, for example, the case in which two nodes are isolated from the rest of the group: even if they cannot connect to a central server machine, they could connect with each other. With our approach, if one of the two is the authority of document D, every configuration management operation is still available. If, instead, both peers are nonauthoritative with respect to D, their replicas can be used to perform check-outs, and check-ins can be cached, waiting for the next opportunity to finalize the operation when the authority becomes available. In this way, people collaboration may be oblivious about actual network topology. Of course, when the number of writers is high, we expect an increase in the number of conflicts. In fact, in any server-based protocol, the server works as a shared point of coordination, and if one can assume its permanent availability, it may be used to ensure that no one works (i.e., modifies artifacts) concurrently with another. However, "optimistic" protocols [39] such as the one normally used by CVS [25] do not force any locking policy, and conflicts are still possible. Therefore, in order to reduce conflicts, workers are advised to do an update before starting their work, if such an update is possible (i.e., they are able to reach the server).

Since the number of nodes, their working profile, and their connectivity profile intuitively impact on the performance of a server-based or a peer-to-peer protocol, we were interested in a comparative assessment of the two architectures. Bellettini *et al.* [40] describe an analytical model of both the PeerVerSy and the CVS protocol with stochastic well-formed nets. Interested readers are invited to refer to [40] and [41] for details about the model and complete results. The model abstracts away the implementation details in order to focus

on the check-in protocol. Moreover, network properties as bandwidth and partial connectivity are not considered: peers are considered to be online (able to communicate with others) or offline (able to manage only local data). The analysis (performed by using the GreatSPN tool [42]) took into account the following parameters:

1) The number of workers (n_p). The model was analyzed with teams up to 12 members because of two concomitant reasons. First, our cooperative approach targets small groups of peers. Second, even so, the analysis requires several days of CPU time on a GNU/Linux machine, equipped with an Intel Xeon 2.4 GHz and 1 GB of main memory, and produces an exceedingly large amount of data.
2) The number of documents under revision control (n_d). We considered a model with one, two, and three documents.
3) Working profile (W). The model assumes that workers repeat cycles in which on average they work (modify a single artifact) two units of time and they are idle for a unit of time. When a conflict is discovered, an additional unit of time is spent in the merge operation. The working profile of authorities may differ from the one of generic workers: one can experiment what happens if the authority works a lot (with respect to other workers) on the document that it owns or the other way around.
4) Online profile (O). Every member may be online or offline. This simplification is needed in order to abstract on the network topology. The server machine (in the server-based model) is assumed to be always online, and a worker can reach it if and only if its status is online. A peer A can reach another peer B if and only if the status of A and B is online. We characterized the online profile of a model with the following two parameters.
   a) Online ratio (OR). How much time, on average, a worker is online against total time.
   b) Connection periodicity (OP). How long (in time units) is an online/offline cycle. This represents the dynamism of the changes in the online status. For example, $OP = 4$ means that if a worker is online half of its time $(OR = 50\%)$, online sessions last on average two time units and offline sessions two time units; $OP = 2$ means that if a worker is online half of its time $(OR = 50\%)$, online sessions last on average one time unit and offline sessions one time unit.

The analysis focused on three metrics.

1) Average successful check-in throughput per worker per document. This measures how many successful check-ins were made by each worker during the simulation per time unit: obviously, the higher the better. In the peer-to-peer setting, we separately considered nonauthoritative peers and authority since one of the main advantages of the protocol is that the authority is virtually always able to perform a successful check-in operation.
2) Average failed check-in throughput per worker per document. We count as failed check-in any time a conflict occurs, and the user needs to merge the changes and resubmit another check-in proposal.
3) The rate of successful check-ins. Obviously, the higher the better. However, this is significant only if work is actually done: if nobody is working and only one check-in is performed, the success rate will be 100%, independent from the strategy used.

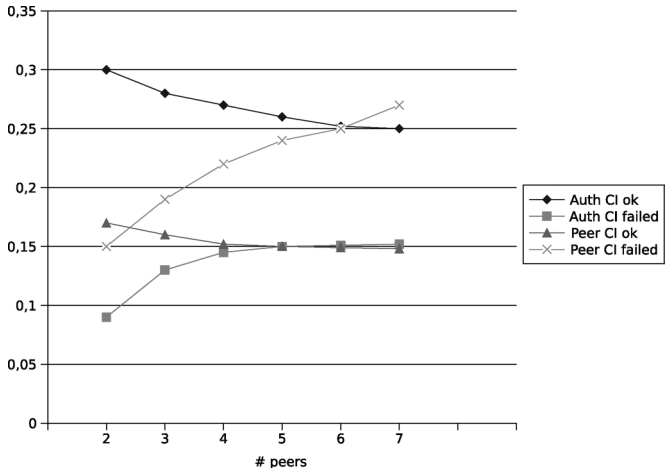

Fig. 3. Check-in performance (on y axis, the throughput of the check-in transition).

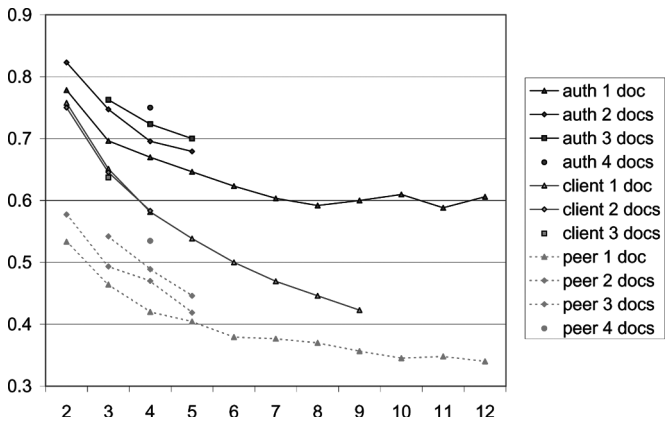

Fig. 4. Client/server versus P2P check-in performance (on y axis, the rate of successful versus failed check-ins).

Fig. 3 shows the trend of the throughput of successful and failed check-in operations against an increasing number of workers (expressed in number of check-in transitions fired in the time unit). The number of successful check-ins decreases with a higher number of workers both for the authority and the peers (data consider only one document); correspondingly, the number of failed check-ins increases. While the number of peers working on a single document heavily affects the average number of failures for normal peers, the authority failures increase very slowly with the number of peers.

Fig. 4 shows a performance comparison between a user in a client–server architecture and a user in our peer-to-peer solution: the diagram depicts the rate of successful check-ins versus failures. Obviously enough, a peer acting as the authority of a document always performs better than the average client,

since a normal peer has less chances to get a successful check in. Authority check-ins are almost always accepted, since they fail only when two or more peer check-ins were accepted during the work activity of the authority. When the workers on a single document are three or more, the average peer get more failed check-ins than successful ones. In a client–server setting, this happens with six or more nodes (a rate of 0.5 means that the number of successful check-ins is equal to the number of failures). However, when a team is working on several documents, a peer-to-peer setting is able to exploit the ability of distributing the authority responsibility, thus the average success increases. Moreover, a "good" choice of the authority who owns an artifact (ideally, the one who works most on it and who is frequently online) could improve performance figures even more: our model shows that it is possible to get results with peers and authorities acting according to different working profiles. The picture shows the results obtained in the most trivial symmetrical case, without exploiting any smart strategy of authority assignment.

In a more realistic setting, the dynamic redistribution of authorities should assure that most of the work is done by authoritative peers, where our approach has a clear advantage with respect to a centralized architecture. Moreover, the reduced performance of normal peers is still quite acceptable (even with an increasing number of peers) given that they gain the flexibility of doing cooperative actions also when offline.

B. PeerVerSy: A Prototype Application

In the second step of our evaluation, we experimented the effectiveness of our solution in an educational environment. In this phase, our distributed approach was used to manage a number of projects developed in a software engineering class for undergraduate students at Politecnico di Milano.

To implement the strategy described in Section IV, we developed a prototype application called PeerVerSy. PeerVerSy is a serverless configuration management system that allows the repository to be distributed and replicated over different nodes of the network in a peer-to-peer fashion. In addition to the distributed repository, the application provides other cooperative software development features, like management of bug lists and to-do lists.

PeerVerSy is built on top of PeerWare [16], [43], a middleware suitable for peer-to-peer settings developed within our group at Politecnico di Milano. PeerWare provides the abstraction of a shared "global virtual data structure" (GVDS) built out of the local data structures contributed by each peer. PeerWare takes care of dynamically reconfiguring the view of the GVDS as perceived by a certain user, according to the current connectivity state. The GVDS 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.

Precisely, nodes are structured as a forest of trees, with distinct roots, which most likely represent different classifications of the documents contained into the data structure. Within this graph, each node is linked to at most one parent node and may contain different children nodes. Standalone 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. Nodes are labeled, and two nodes may have the same label, as long as they can be uniquely identified. Nodes and documents are, in fact, identified by path names starting from the root node label, as it happens in conventional file systems.

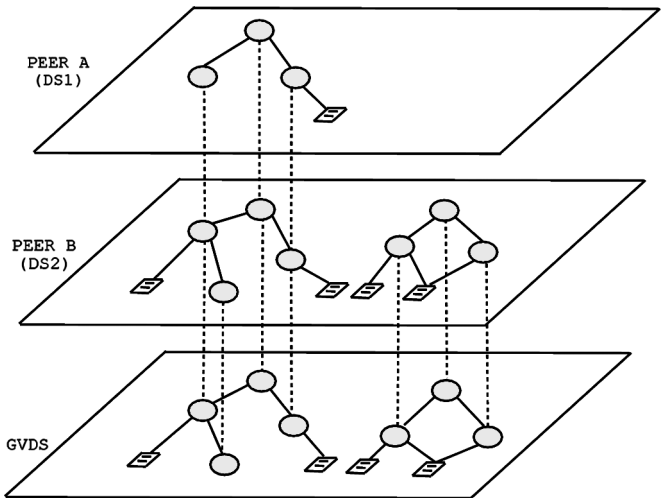

Fig. 5. Example of the GVDS managed by PeerWare.

At any time, the local data structures held by each peer are made available to the other participants as part of a GVDS. This global view is structured in the same way as the local data structure but its contents are obtained by merging all the local data structures belonging to the peers that are currently connected, as shown in Fig. 5.

Changes in connectivity among peers determine changes in the contents of the GVDS, as new local data structures may become available or may disappear. The reconfiguration, however, takes place behind the scenes and is completely hidden to the peers accessing the GVDS.

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 [44].

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. In particular, the atomicity property can be assumed to hold for local operations, but it is an impractical assumption in a distributed setting. Hence, they are not guaranteed to be atomic. They only guarantee 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 the publish/subscribe engine on which PeerWare itself is based (the distributed event dispatcher JEDI [45]) provide the framework needed to implement our configuration management operations. By using PeerWare, we abstract away from the actual

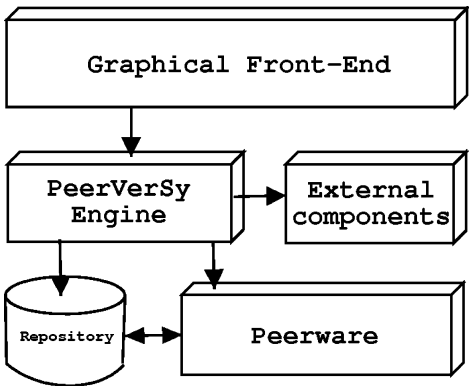

Fig. 6. Architecture of PeerVerSy.

network topology and perform actions on online items without knowing where they are stored.

The prototype application is built on top of PeerWare as shown in Fig. 6.

The local repository is the local space where each node stores its documents (both master copies and replicas). Each artifact is composed of the data and metadata associated to all the versions of the document. The repository is managed by PeerWare in order to provide to each node a virtual view on the content of all the local repositories. This allows users to read documents also if they do not own a local copy in their data spaces.

The core engine provides a wide set of functionalities to check-in and check-out documents, to post messages into the bug list, to define baselines, to add new artifacts to the repository, and other CM-related operations. The engine relies on the public subscribe services provided by the middleware layer to send and receive messages related to the notification of new versions of artifacts.

We decided to implement into the engine only the features that are not context specific. Other functionalities are provided by external plugins that are dynamically loaded and executed by the engine only when they are needed. There are several types of component, such as components used to validate the data introduced into the repository, components used to differentiate and merge documents, and components simply used to execute an action in particular situations. A component can also be associated with a particular type of artifact, for example, it is possible to select a MergeComponent that will be loaded to resolve conflicts between Java source files.

A special type of component is the one responsible for managing the distribution of authorities between peers. These components are selected by the project manager to establish a project-specific process. By writing a customized component, it is possible to implement the policy that is best suited for a particular software process.

Although PeerVerSy provides a quite complete set of functionalities, it was essentially designed to evaluate the feasibility and usability of a peer-to-peer configuration management system in a controlled environment. The case study consisted of a software engineering class involving 20 students equipped with tablet PCs and a wireless LAN card. During the course's laboratory sessions, we required students to collaborate in small teams to develop a software project. Each group of students worked together in a laboratory with a wireless network infrastructure once a week for four weeks. When the laboratory session was over, they were supposed to continue their work independently at home. Moreover, they could meet together on campus or whenever and wherever they wanted, setting up an "ad hoc" network to synchronize their local repositories and to exchange the latest versions of the artifacts they were working on.

At the end of the class, we compared the work experience of these students with the rest of the class that developed the same projects using CVS. We found that the tool was extremely attractive from the instructor perspective, who did not need to set up a server-based configuration management tool. Security regulations of the laboratory impose a high cost in setting up a central server to be accessed by many students. Instead, our solution allows students to use their own machines, installing and configuring the program under their own responsibility. We also received positive feedback from participating students, who were happy to be able to freely cooperate whenever and wherever they wanted and could also work in a disconnected fashion when necessary. However, we were not able to evaluate the impact of the new tool on software development in a more objective, let alone quantitative, way. Since the students were not sufficiently skilled and were not used to work cooperatively, we did not get a relevant feedback from this testing phase.

C. Evaluation in a Real Scenario

The previous experiment was useful in sustaining our hypothesis about the feasibility of a totally decentralized configuration management approach. Moreover, analytical results proved that for small groups of developers, our totally distributed protocol may also present advantages with respect to a client–server architecture. Unfortunately, PeerVerSy, our first implementation of the algorithm, lacks the robustness required by a commercial product (e.g., it does not provide any security and authentication facilities), and its limitations make it difficult to promote the tool in a real environment. Developers are reluctant in putting their documents in the hands of a new prototype SCM system, and it was not our purpose to develop from scratch a new product to compete on the market. For this reason, we are now reimplementing the PeerVerSy algorithm as a set of scripts that can be applied on top of an existing SubVersion [13] repository. This solution allows us to focus only on the distributed algorithm, letting the document management to the underlying well-known and well-tested product. In addiction, the use of external scripts makes it possible to disable the distributed features in any moment, coming back to the plain centralized repository.

The scripts running on each host are responsible for enforcing our policies during check-in and check-out operations and of exchanging messages with the other peers in order to maintain the local repository up to date. As far as the network communication concern, we decided to substitute PeerWare with a more lightweight and scalable system named Reds [46]. Reds is a highly configurable framework that can be used to build publish subscribe applications for large dynamic networks. Thus, nodes that own a replica of an artifact can

subscribe themselves to receive messages that are published by the document authority whenever a new version becomes available. Moreover, real-world wireless connections suffer from unannounced disconnections, and the new middleware provides support for them.

We believe that such an unintrusive implementation will attract the interest of many people, free to try the advantages of a totally distributed configuration management system with their preexisting repositories.

VI. Conclusion and Future Work

In this paper, we have discussed a novel approach to support cooperation for small groups of nomadic developers. These groups can exchange data, but they can rarely rely on the availability of a centralized repository server.

We presented a solution to support nomadic workgroups by allowing any online hosts to be able, in principle, to cache the artifacts and make them available for use even if the hosts that own them are temporarily disconnected. Of course, we pay for this flexibility in terms of a more sophisticated coordination effort.

As in all cooperative processes, human factors play a very important role in our system. However, other support systems for distributed cooperation (like Co-Op) require going through complex consensus building stages. In our case, instead, reaching consensus about modifications is restricted to an interaction between two individuals: the owner of the artifact and a user who modified it. Furthermore, our system allows the user who more frequently modifies an artifact to be promoted to the owner role. This further reduces the need for negotiation among team members as needed in highly dynamic context.

The approach we described in this paper has been implemented in a prototype as part of our efforts in developing a suite of software process support tools that fit the needs of educational environments in which students are equipped with mobile devices and form dynamic virtual communities. The approach has been validated informally in practice and modeled with stochastic Petri nets in order to assess the performance we should expect. We are currently refactoring our tool as a collection of scripts on top of a full-fledged versioning system (SubVersion). This should make it possible to use the tool (now providing all the features of a real-world configuration management tool) in production environments.

Acknowledgment

The authors would like to thank C. Bellettini and L. Capra for their high-valued contribution in setting up and analyzing the results of the simulation of PeerVerSy by exploiting a stochastic Petri nets model.

References

[1] J. Grudin, "Computer-supported cooperative work: History and focus," *Computer*, vol. 27, no. 5, pp. 19–26, May 1994.

[2] E. Kirda, P. Fenkam, G. Reif, and H. Gall, "A service architecture for mobile teamwork," in *Proc. 14th Int. Conf. Softw. Eng. and Knowl. Eng.*, 2002, pp. 513–518.

[3] H. Gall and S. Dustar, "Architectural concerns in distributed and mobile collaborative systems," *J. Syst. Archit.: EUROMICRO J.*, vol. 49, no. 10/11, pp. 457–473, Nov. 2003.

[4] L. Baresi, S. Dustdar, H. Gall, and M. Matera, Eds., *Ubiquitous Mobile Information and Collaboration Systems*, Lecture Notes in Computer Science, vol. 3272. New York: Springer-Verlag, 2004.

[5] A. Sheth, W. van der Alst, and I. Arpinar, "Processes driving the networked economy," *IEEE Concurrency*, vol. 7, no. 3, pp. 18–31, Jul.–Sep. 1999.

[6] W. van der Aalst and K. van Hee, Eds., *Workflow Management: Models, Methods, and Systems*. Cambridge, MA: MIT Press, 2004.

[7] D. Hollingsworth, "The workflow reference model," Workflow Management Coalition, Brussels, Belgium, Tech. Rep. TC00-1003, Nov. 1994. issue 1.1.

[8] G. Cugola and C. Ghezzi, "Software processes: A retrospective and a path to the future," *Softw. Process: Improv. Pract.*, vol. 4, no. 3, pp. 101–123, 1998. [Online]. Available: citeseer.nj.nec.com/cugola98software.html

[9] L. Osterweil, "Software processes are software too," in *Proc. 9th Int. Conf. Softw. Eng.*, Monterey, CA, Mar. 1987, pp. 2–13.

[10] P. K. Garg and M. Jazayeri, Eds., *Process-Centered Software Engineering Environments*. Piscataway, NJ: IEEE Press, 1996.

[11] J. Estublier, D. Leblang, G. Clemm, R. Conradi, A. van der Hoek, W. Tichy, and D. Wiborg-Weber, "Impact of software engineer research on the practice of software configuration management," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 4, pp. 383–430, Oct. 2005.

[12] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability maturity model, version 1.1," *IEEE Softw.*, vol. 10, no. 4, pp. 18–27, Jul. 1993. [Online]. Available: citeseer.nj.nec.com/paulk93capability.html

[13] Subversion, 2005. [Online]. Available: http://subversion.tigris.org/

[14] Monotone, 2005. [Online]. Available: http://venge.net/monotone/

[15] MOTION Consortium, *Mobile Teamwork Infrastructure for Organization Networks*, 2000. iST Project IST-1999-11400. [Online]. Available: http://www.motion.softeco.it/pages/mainSummary.html

[16] G. Cugola and G. P. Picco, "Peer-to-peer for collaborative applications," in *Proc. Int. Workshop Mobile Teamwork Support (Co-Located With 22nd ICDCS)*, H. Gall and G. P. Picco, Eds, Vienna, Austria, Jul. 2002, pp. 359–364.

[17] D. Balzarotti, C. Ghezzi, and M. Monga, "Freeing cooperation from servers tyranny," in *Web Engineering and Peer-to-Peer Computing*, Lecture Notes in Computer Science, vol. 2376, E. Gregori, L. Cherkasova, G. Cugola, F. Panzieri, and G. P. Picco, Eds. New York: Springer-Verlag, 2002, pp. 235–246.

[18] Politecnico di Milano, *A Virtual Campus PoliMi*, 2003. [Online]. Available: http://www.elet. polimi.it/res/vcampus/

[19] E. Di Nitto, L. Mainetti, M. Monga, L. Sbattella, and R. Tedesco, "Supporting interoperability and reusability of learning objects: The virtual campus approach," *J. Educ. Technol. Soc.*, vol. 9, no. 2, pp. 33–50, 2006.

[20] T. Hodel, H. Gall, and K. Dittrich, "Dynamic collaborative business processes within documents," in *Proc. 22nd Annu. Int. Conf. Des. Commun.*, 2004, pp. 97–103.

[21] W. F. Tichy, Ed., *Configuration Management*, Trends in Software, vol. 2. Hoboken, NJ: Wiley, 1994.

[22] J. Estublier, "Software configuration management: A road map," in *The Future of Software Engineering*, A. Finkelstein, Ed. New York: ACM Press, May 2000.

[23] A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. L. Wolf, "A testbed for configuration management policy programming," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 79–99, Jan. 2002.

[24] A. Oram, Ed., *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, 1st ed. Sebastopol, CA: O'Reilly Associates, Mar. 2001.

[25] *CVS: Concurrent Versions System*, 1989. [Online]. Available: http://www.cvshome.org/

[26] *ClearCase MultiSite Manual*, Rational Software Corporation, Lexington, MA, Aug. 2000. (release 4.0 or later).

[27] A. Carzaniga, "Design and implementation of a distributed versioning system," Politecnico di Milano, Milan, Italy, Tech. Rep. 98-88, Oct. 1998.

[28] A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. L. Wolf, "A reusable, distributed repository for configuration management policy programming," Univ. Colorado, Boulder, CO, Tech. Rep. CU-CS-864-98, Oct. 1998.

[29] BitKeeper, 2001. [Online]. Available: http://www.bitkeeper.com/Welcome.html

[30] B. Milewsky, "Distributed source control system," in *Software Configuration Management*. Lecture Notes in Computer Science, vol. 1235, New York: Springer-Verlag, 1997, pp. 98–107.

[31] *The Linux Kernel*, 1990. [Online]. Available: http://www.kernel.org/

[32] Apache Software Foundation, *The Apache HTTP Server*, 1996. [Online]. Available: http://httpd.apache.org/

[33] *The Mozilla Browser*, 2000. [Online]. Available: http://www.mozilla.org/
[34] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, Jun. 2002.
[35] SVK, 2005. [Online]. Available: http://svk.elixus.org/
[36] 1987. [Online]. Available: http://www.coda.cs.cmu.edu/
[37] A. van der Hoek, D. Heimbigner, and A. L. Wolf, "A generic, peer-to-peer repository for distributed configuration management," in *Proc. 18th Int. Conf. Softw. Eng.*, Berlin, Germany, Mar. 1996, p. 308.
[38] P. Mockapetris, "Domain names—Implementation and specification," Internet Engineering Task Force. Nov. 1987, Tech. Rep. RFC 1035, (Standard: STD 13).
[39] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981.
[40] C. Bellettini, L. Capra, and M. Monga, "Using a stochastic well-formed net model for assessing a decentralized approach to configuration management," *Perform. Eval. J.*, 2006. to be published.
[41] ——, "A comparative assessment of peer-to-peer and server-based configuration management systems," in *Proc. Workshop Cooperative Support Distrib. Softw. Eng. Process. (CSSE'04)*, A. D. Lucia, H. C. Gall, and S. Dustdar, Eds, Linz, Austria, Sep. 2004, pp. 15–26.
[42] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo, "GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets," *Perform. Eval.*, vol. 24, no. 1/2, pp. 47–68, Nov. 1995.
[43] F. Bardelli and M. Cesarini, "Peerware: Un middleware per applicazioni mobili e peer-to-peer," M.S. thesis, Politecnico di Milano, Milan, Italy, 2001.
[44] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Mobile Object Systems.* Lecture Notes in Computer Science, vol. 1222. Berlin, Germany: Springer-Verlag, 1997, pp. 49–64.
[45] G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an event-based infrastructure to develop complex distributed systems," in *Proc. ICSE*, Kyoto, Japan, Apr. 1998, pp. 261–270.
[46] G. Cugola and G. P. Picco, "REDS: A reconfigurable dispatching system," *Proc. 6th Int. Workshop SEM*, Nov. 2006, Portland, OR. submitted for publication.

Davide Balzarotti received the degree in computer engineering and the M.S. degree in information technology from the Società consortile a Responsabilità Limitata (CEFRIEL), Milan, Italy, and the Ph.D. degree in computer engineering from the Politecnico di Milano, Milan, in 2006.

He is currently a Postdoctoral Researcher with the Department of Computer Science, University of California, Santa Barbara. His research interests include software engineering and computer security.

Carlo Ghezzi (SM'00–F'06) received the Dr.Eng. degree in electrical engineering from the Politecnico di Milano, Milano, Italy. He was with the University of Padova, Padova, Italy, and the University of North Carolina at Chapel Hill.

He is currently a Professor of software engineering with the Dipartimento di Elettronica e Informazione, Politecnico di Milano. He is a coauthor of over 130 scientific papers and eight books. His research interests are in software engineering and programming languages. He is currently particularly interested in the theoretical, methodological, technological, and organizational issues involved in developing network applications. He is the Editor-in-Chief of the *ACM Transactions on Software Engineering and Methodology* and the Associate Editor of *Software Process Improvement and Practice*, and *Science of Computer Programming*.

Dr. Ghezzi was a member of the editorial board (and Associate Editor-in-Chief) of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING until 1999. He is a Fellow of the Association for Computing Machinery.

Mattia Monga received the Ph.D. degree in computer and automation engineering from the Politecnico di Milano, Milan, Italy.

He is currently an Assistant Professor with the Dipartimento di Informatica e Comunicazione, Università Degli Studi di Milano, Milan. To support the free-software community, he joined the Debian Project as a Developer. His research activities are in the field of software engineering and security.

Dr. Monga is an Affiliate Member of the IEEE Computer Society and is in the steering committee of CLUSIT, an Italian association promoting awareness, continuous education, and information sharing about digital security. He collaborates with the Association for Computing Machinery as the Information Director of the *ACM Transactions on Software Engineering and Methodology*.