# Supporting nomadic co-workers: an experience with a peer-to-peer configuration management tool

Mattia Monga
Dipartimento di Informatica e Comunicazione
University of Milan, Italy

E-mail: `mattia.monga@unimi.it`

## Abstract

*Nowadays the Internet infrastructure is so pervasive that it is common that people connect their laptop computer from a range of different locations: office, home, the hotel hosting them for a conference, or the meeting room where they are working. This is sometimes called mobile computing and it forces the designers of applications to cope with two new requirements: (1) users may connect to the network from arbitrary locations (usually with different network addresses), and (2) they are not permanently connected. Thus, connectivity is intrinsically transient, and machine disconnection is not an exceptional case, but the normal way of operating. We investigated how collaborative work can be supported in a mobile computing setting, where the notion of permanent central server cannot be used. Support tools for CSCW are normally based on a client-server architecture, which appears to be unsuitable in a such a dynamic environment. For this reason we experimented peer-to-peer solutions, which do not rely on services provided by a centralized server. In particular, we have implemented a configuration management tool – called PeerVerSy – which provide collaborative actions even when some of the collaborating nodes are off-line.*

## 1. Introduction

Until a couple of years ago people used to connect to the Internet from their workstations located in their offices. They composed their electronic artifacts on their computers and they shared their work with others by exploiting the mediation of some Internet server. This scenario is still common and useful in several cases. However, nowadays network connections are available in a range of different locations: offices, homes, hotels, meeting rooms, airplanes, etc. Nevertheless, laptop users are often forced to work without any Internet connection. Thus, network applications that rely on servers are sometimes not desirable or even not feasible.

The description of a real world scenario will clarify the problem. Imagine two or three software companies are working on some joint project. They meet together in a conference center in order to plan future activities. Because the future progress of work is still undecided they did not set up any shared server. Moreover, they can connect each one with any other thanks to the wireless connection provided in the conference room, but they are not able to connect to their companies due to firewall restrictions. In such a case, the use of a computer supported collaborative work (CSCW) tool is not an option. These systems are normally based on a client/server architecture and one of them should probably be installed on the personal laptop of one of the working people. Even in this case, consider the possible drawback: what if the person in charge of the server has to go home, where he cannot be reached by incoming connections? If the collaborative tool is some kind of configuration management tool, the server machine keeps a repository of all artifacts on which people is working, and if the server vanishes all the work must stop, even if the person hosting the server was personally responsible of just one single artifact.

This simple scenario should suffices to grasp why new server-less application for collaborative work are needed. In fact, certain architectural assumptions on the distributed infrastructure affect the way cooperative support is provided. The client/server approach is possible, and suitable, in all cases where a reliable and permanent network infrastructure is available to connect the participating nodes. In many cases, instead, people would like to collaborate while they are supported through a much looser architecture. Thus, the reference architecture is a network of peers, each of which contributes to the overall logical structure in an equivalent way. Moreover, peers cannot be assumed to be always on-line. The network connection is intrinsically intermittent, as in the case of wireless connections. More specifically, peers may dynamically join and leave an ad-hoc community. They join it in impromptu meetings, where they synchronize their works. Each peer, however, should continue to provide its functionality even when it is in a disconnected stage. The support infrastructure should handle connections and disconnections in a seamless fashion.

This paper is organized as follows: Section 2 presents PeerVerSy, our peer-to-peer versions system, Section 3 describe the experimental work we did in order to evaluate the tool, Section 4 discusses related works and finally Section 5 draws some conclusions.

## 2. PeerVerSy: a peer-to-peer versions system

### 2.1. Software development in a mobile context

We focussed our investigation on the collaborative work needed to produce software systems. Software developers typically collaborate by exchanging and sharing a number of files. Files are assigned to people according their responsibilities in the project. However, besides the person in charge of a file, several other collaborators sometimes need to view or modify it. In general, for each item we can identify the role of an *owner,* i.e., the individual who has created the artifact or who is in charge of carrying out the work on it. Moreover, there is a number of other collaborators involved in the project who need to manipulate items that are not under their direct control, i.e., artifacts they do not own.

In order to keep the system consistent, developers use *configuration management* tools [12]. The main idea is that a reference version of the system is maintained in a central repository. People work accordingly to the rule that one has to *check-out* an artifact from the repository if s/he wants to modify it. After the modification is performed, then the new version is accepted in the repository with an operation called *commit* or *check-in.* 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:

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

However, in a *mobile computing* scenario these assumptions are too strong. In fact, collaborative applications for nomadic users face two new requirements:

1. users connect to the network from arbitrary locations (possibly with different network addresses)
2. they are not permanently connected, and, while disconnected, they want to work without virtually notice the difference

In other words, no fixed network topology can be assumed and machine disconnection is not an exceptional (or faulty) case, but the normal way of operating: even by assuming the reliability of the network, people *want* to be off-line sometimes. 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 dynamism, because of the intrinsic dependency from what we called *server tyranny*[1].

### 2.2. A tool for version management

Our approach is demonstrated by the tool for version management we built. The tool is called PeerVerSy and a working prototype can be downloaded from `http://sf.net/projects/peerversy/`.

In traditional, client/server version management tools (for example CVS[7]), two (or more) persons may work on the same artifact (a file): both are required to check out the artifact from the server machines. Different control policies for concurrent work are possibile. An example of *optimistic* concurrency control[8] is depicted in Figure 1: Alice and Bob are both working on `foo.c`; Alice finishes her work and the new version is committed in the repository (this becomes the new reference version); when Bob tries to check in his own modified version he gets a conflict from the server and he must merge his modifications with Alice's ones before trying a new check in.

This schema relies on the assumption that the server is available when check-in and check-out operations have to be performed. Alice and Bob have to be connected when they do those operations, otherwise they get an error (and the operation fails).

Instead we design `PeerVerSy` such that users might freely accomplish their computations and collaborative actions as check in and check out also when *off-line.* An automatic reconciliation step is performed when connection is established again, possibly arising conflicts. The client/server approach 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

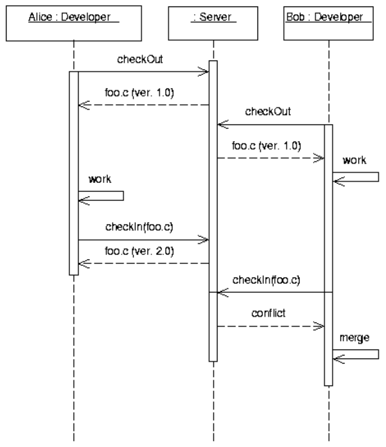

**Figure 1. Optimistic CM with a central server**

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 or on the file system of any connected node. However, these operations, while performed in unthetered or even off-line mode, should be fully compliant with the cooperative system policies. We base our systems on change notifications: when a peer joins the network, it notifies the changes made to its own items since it went off-line. In this way, any cached copies kept by such peers become invalid.

### 2.3. PeerVerSy implementation

The architecture of our system is quite simple. Every node is functionally equivalent to any other and it holds a (partial) replica of the repository. 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.

Thus, we needed a middleware able to simulate a shared memory among the peers, also in the presence of disconnections. We chose PEERWARE [6, 2], developed at Politecnico di Milano[1], because it 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. 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 2.

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. The operations provided by PEERWARE together with a publish/subscribe engine on which PEERWARE itself relies on (the distributed event dispatcher JEDI, see [5]) build the framework needed to implement our configuration manage-

[1]Available for downloading at `http://peerware.sourceforge.net`

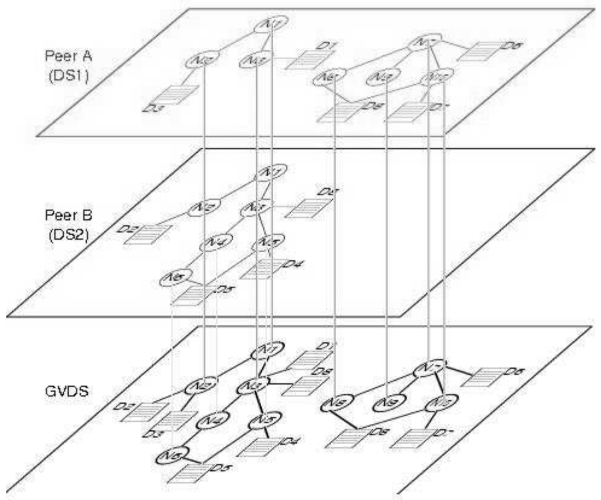

**Figure 2. The GVDS abstraction provided by Peerware**

ment operations.

The main point is that by using PEERWARE we can abstract from the actual network topology and perform actions on *on-line* items without knowledge on where they are stored.

## 2.4. The algorithm

Reconciliation is based on the assumption that each peer is the *authority* for a set of items (it *owns* them), and the copy of an artifact owned by the authority is the *master* copy. In addition to the master copy, the others peers can keep a local copy (*replica*) of the documents they do not own in order to allows 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 also from the local copies of a document and 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 accepts the changes and updates the master copy.

At the beginning the authority is assigned to the peer that inserts the document into the system, but because the choice of the authoritative peer is critical, the bindings among authorities and peers are not static and can be moved from a peer to another one in order to improve the overall performance of the system. For instance, if the node $X$ is the authority of the document $d$ but $Y$ is responsible for the last ten check-ins, it is reasonable that $Y$ will be promoted the new authority of $d$.

When a peer enters the community, 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 peers that owns a replica of $i$ must update their copies.

When a peer $X$ wants to check-out a document $d$ whose authoritative peer is $A$ ($\neq X$), if $d$ is present in the local repository of $X$ the operation boils down to getting a local copy of $d$. Instead, if $d$ is not present in the local part of repository under control of $X$ a network search is issued to retrieve a valid copy. Only in the case that no copy is found the check-out operation fails.

When a peer $X$ wants to check-in a new version of a document whose authoritative peer is $A$ ($\neq X$), $X$ notifies its request $A$ if is reachable: $A$ can reject the proposal or accept it, making it persistent in its local part of the repository as the new master copy. If $A$ is not reachable by $X$ the check-in proposal is recorded in the local repository hosted by $X$ and when $A$ becomes eventually available, the proposal is notified to it. In both cases, if $A$ has an item newer then the one proposed by $X$ a conflict arises and the $X$ proposal is refused. In this case, $X$ must resolve the conflict and then it can submit a new version.

An example with three peers is shown in Figure 3. Alice is the authority for the artifact `foo.c`, and she can check out and check in it freely (no conflicts are ever raised for her operations). Bob checks `foo.c` and get the current version from Charlie, Alice might be off-line without affecting the operation. When a new version of `foo.c` is accepted in the repository, any interested peer is notified about it when is able to receive the notification (i.e., when it is on-line). A check in request by Bob, is notified to the authority (Alice) when she is available and, if all goes well, accepted, and finally notified to all interested peers. Users are unaware of the connectivity state of the other peers, thanks to the underlying middleware.

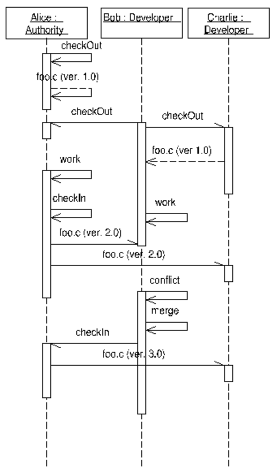

**Figure 3. Three peers collaborating by using PeerVerSy**

## 3. Tool evaluation

In order to evaluate the effectiveness of our solution in a real use case, we did a preliminary assessment of PeerVerSy during a course of Software Engineering for under-graduated students at Politecnico di Milano. The experiment involved 20 students equipped with tablet-pcs and a wireless LAN card. During the laboratory part of the course, we required that students collaborate in small groups to develop a software project. Each group of students worked together into the wireless laboratory one day a week for four weeks. When the lesson was over, they were supposed to continue their work at home. Moreover, they could meet together at the university campus or whenever and wherever they wanted, setting up an “ad hoc” network in order to synchronize their local repository and to exchange the last versions of the artifacts they were working on.

We found that the tool was extremely attractive from a teacher perspective. In fact, it is often impossibile to set up a server based configuration management tool: security regulation of the laboratory imposes an high cost on setting a central server accessed by a hundred students. Instead, our solution allows that students use their own machine, installing and configuring the program on their own responsibility. We received also a positive feedback from participating students, who were happy to be able to freely cooperate wherever they want, and even work at home without any Internet connection. However, we were not able to evaluate the impact of the new tool on software development. In fact, the students were not sufficiently skilled and accustomed to configuration management to get a relevant feedback on this.

Thus, in order to evaluate the effect of our server-less policy on software development, we designed a simple simulation of a team of developers using PeerVerSy to coordinate their work. In our simulation setting a number $N$ of developers work on the same artifact. One of them is the authority owning the artifact. Every peer is characterized by two parameters: its attitude in being on-line and its attitude in working on the artifact. The state machine modelling the artifact is depicted in Figure 4.

Figure 5 shows the results of simulation in a setting with 5 peers, each of them having a work attitude of 0.2 (working 1 time out of 5 times units) and a on-line attitude of 0.5. The simulation was repeated with different values for the authority attitude in being on-line (work attitude fixed at 0.2). The graph shows the ratio between tried check-in operations and successful ones. The ratio is getting better as the authority goes online often. This is sensible, and it confirms our intuition that if we could assume that some node is always online (online attitude 1.0) the best possible system we could design is a server based one. Please note that according our simulation in a server based system like CVS, we should expect about a 25% of conflicts a team composed by four developers (the authority does not count, since its operations never fail in our approach)

Figure 6 shows the results of simulation in a setting with 5 peers, each of them having a work attitude of 0.2 (working 1 time out of 5 times units) and a on-line attitude of 0.5. The simulation was repeated with different values for the authority attitude in being working. The graph shows results both for on-line attitude 0.5 and online attitude 1.0 (approximately analogous to a server based system). Choosing an authority with a high attitude to work, compared with the attitude of other peers, can greatly diminish conflicts.

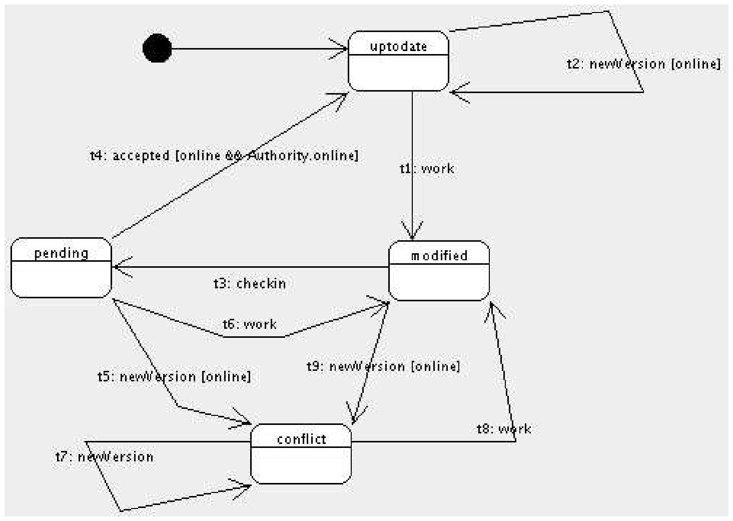

**Figure 4. States of an artifact**

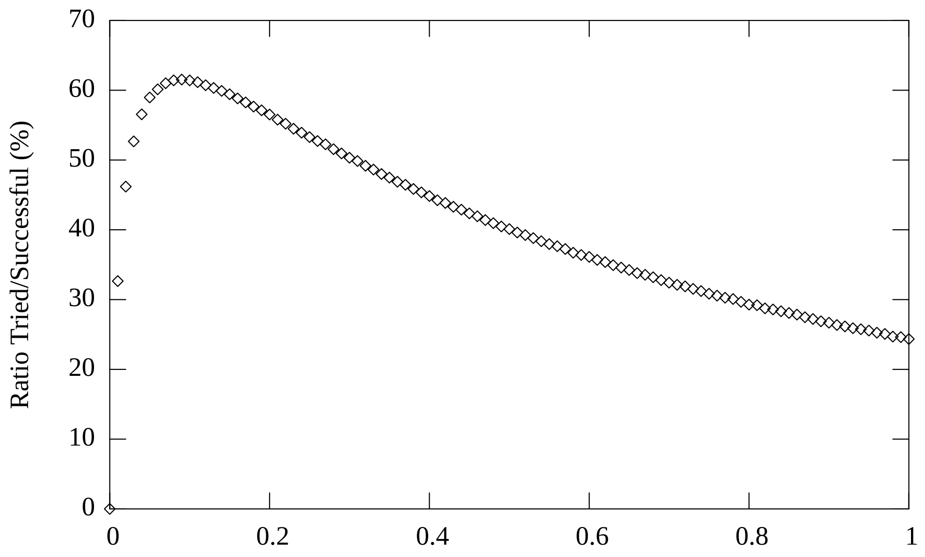

**Figure 5. Percentual conflicts with 5 peers (work attitude 0.2, online attitude 0.5) related to authority online attitude**

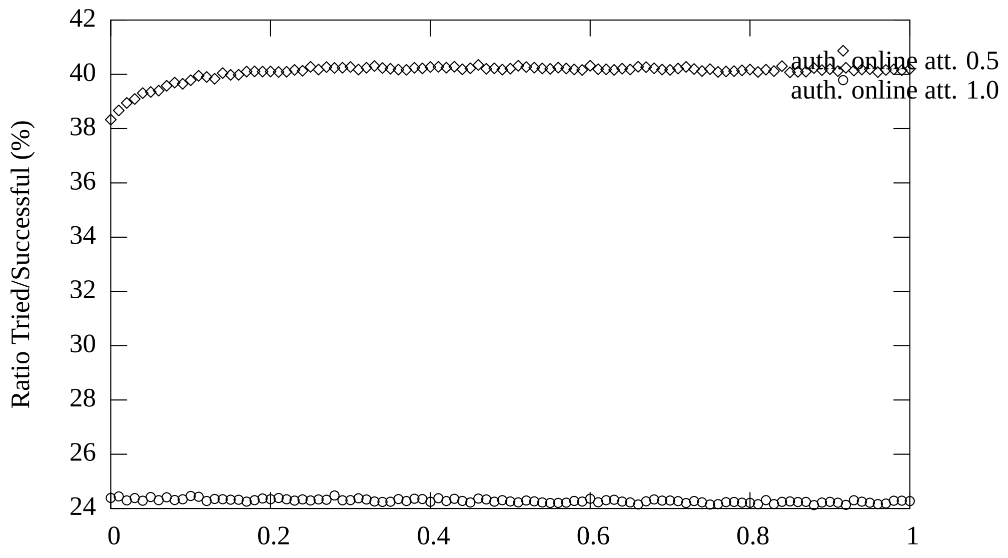

**Figure 6. Percentual conflicts with 5 peers (work attitude 0.2, online attitude 0.5) related to authority working attitude**

Figure 7 shows the results of simulation obtained by changing the number of the equivalent peers (work attitude 0.2 and online attitude 0.5) belonging to the team. Again, the simulation confirms our intuition that the number of conflicts is getting worse as the number of peers grows. That is, as Fred Brooks has pointed out in his “The Mythical Man-month” [3], collaboration does not scale very well. However, it is worth noting that keeping a node always online improve slightly the situation (see the line with authority attitude in being online equal to 1.0)..

## 4. Related work

Rational ClearCase Multisite [11] 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 to 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. Our approach enables a much more flexible access policy. Moreover, the replica of the whole repository can be too expensive in a network of laptop computers. In the best case is a waste of resources since every developer typically modifies just a fraction of the repository. PeerVerSy uses a *lazy* replication policy and only artifacts actually used are replicated on a peer.

DVS [4] is a research system that allows one to distribute the configuration management 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 configuration management application and the underlying middleware. In fact, DVS has been implemented on top of NUCM [13] (Network-Unified Configuration Management). NUCM defines a generic distributed repository and provides a policy-neutral interface to realize configuration management systems.

The philosophy of our solution is similar to the one adopted by Co-Op [9]. They claim that each member of the team must have a piece of the database that is necessary for his/her activities. Thus, check-ins and check-outs are possible also when people are off-line. They also claim that exchange of information among the members of the team should occur in separation from other sources of control activities and their solution uses e-mail[2] to share change scripts. During the 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 script, they all have to have the same reference version of the file. It means that, before a given version of a file may become a reference version,

[2]They suggest that their system can be used also by exchanging floppies or by exploiting ftp. In any case users need to agree on a protocol or to set up an infrastructure

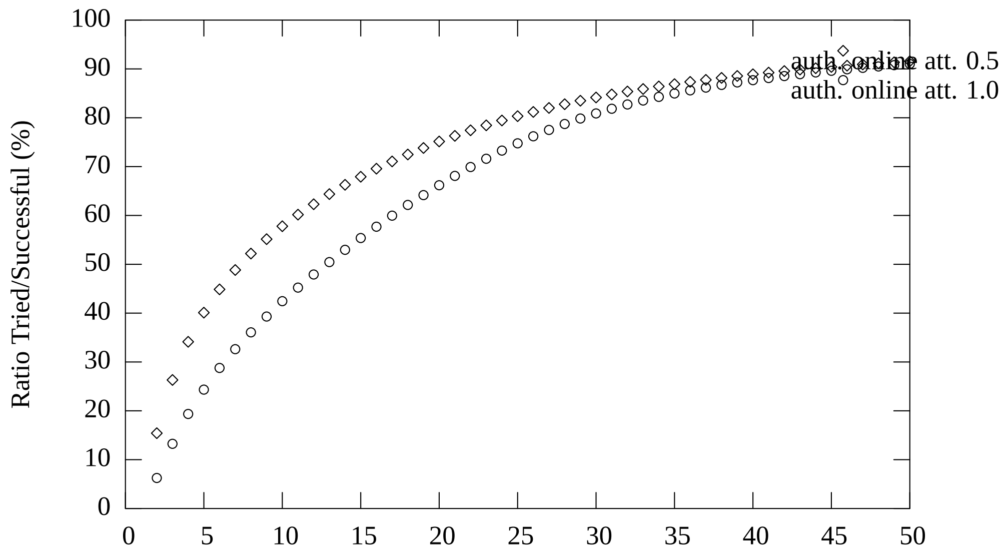

**Figure 7. Percentual conflicts related to the number of peers (work attitude 0.2, online attitude 0.5)**

there has to be consensus about it among all the members of the project: they design a proper algorithm at this end. Our system aims at being more flexible: in particular we wanted to be able to share items also if the sender did not know the address of the possible recipients. In fact, the IP address of any peer can be changed among different connections and we do not want to rely on any server based infrastructure as e-mail. The main advantage of the approach adopted by Co-Op is that their mean of exchanging messages (i.e., e-mail) is probably much more scalable in a WAN settings than a peer-to-peer middleware. However, in the most common LAN setting, our solution require far less infrastructure.

## 5. Concluding remarks

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. 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. We found that such a solution has 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;

- *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;

These advantages are available at the cost of the loss of the centralized control. However, we think that the achieved flexibility is a real plus in most of the modern higly dynamic scenarios.

## Acknowledgements

The work described in this paper was done in collaboration of prof. Carlo Ghezzi and Davide Balzarotti from Politecnico di Milano. It was partially supported by a grant of Microsoft Research.

## References

[1] D. Balzarotti, C. Ghezzi, and M. Monga. Freeing cooperation from servers tyranny. In E. Gregori, L. Cherkasova, G. Cugola, F. Panzieri, and G. P. Picco, editors, *Web Engineering and Peer-to-Peer Computing*, volume 2376 of *LNCS*, pages 235–246. Springer-Verlag, 2002.

[2] F. Bardelli and M. Cesarini. Peerware: un middleware per applicazioni mobili e peer-to-peer. Master's thesis, Politecnico di Milano, 2001.

[3] F. Brooks. *The Mythical Man-Month: Essays on Software Engineering (Anniversary Edition)*. Addison-Wesley Pub Co, 1995.

[4] A. Carzaniga. Design and implementation of a distributed versioning system. Technical report, Politecnico di Milano, Oct. 1998.

[5] 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.

[6] G. Cugola and G. P. Picco. Peerware: Core middleware support for peer-to-peer and mobile systems. submitted for publication, 2001.

[7] Concurrent versions system. `http://www.cvshome.org/`.

[8] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[9] B. Milewsky. Distributed source control system. In *Software Configuration Management (Lecture Notes)*, pages 98–107, 1997.

[10] P. Mockapetris. Rfc 1035 (standard: Std 13) domain names–implementation and specification. Technical report, Internet Engineering Task Force, November 1987.

[11] Rational Software Corporation, Maguire Road Lexington, Massachusetts 02421. *ClearCase MultiSite Manual (release 4.0 or later)*, 2000.

[12] W. F. Tichy. Programming-in-the-large: Past, Present, and Future. In *Proceedings of the 14th International Conference on Software Engineering*, pages 362–367, May 1992.

[13] A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. L. Wolf. A reusable, distributed repository for configuration management policy programming. Technical report, University of Colorado, Boulder CO 80309 USA, Oct. 1998.