Supporting Configuration Management for Virtual Workgroups in a Peer-to-Peer Setting

[Full paper]

Davide Balzarotti
CEFRIEL
Via Fucini, 2
I 20133 – Milano, Italy
balzarot@cefriel.it

Carlo Ghezzi
Politecnico di Milano
Dip. di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
I 20133 – Milano, Italy
ghezzi@elet.polimi.it

Mattia Monga
Politecnico di Milano
Dip. di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
I 20133 – Milano, Italy
monga@elet.polimi.it

ABSTRACT

In this paper we describe a configuration management tool suitable for the untethered scenarios typical in a mobile environment. The scenario envisions a number of homogeneous peers that are able to provide the same services, disconnect frequently from the net, and perform part of their work while disconnected. In these contexts the absence of a host is not the exceptional case, but rather the normal behavior. Thus, a traditional architecture based on a central repository exposes the system to failures when the server is unavailable. Instead, we build our system on a peer-to-peer middleware able to provide the abstraction of global virtual data structure, i.e., a data structure composed by all the data actually connected in a given instant. Thanks to this, we can exploit the service provided by the network even if relevant hosts are disconnected.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Mobile computing, middleware, peer-to-peer

1. INTRODUCTION

The recent advances in the area of wireless networks and the popularity of powerful mobile computing devices, such as laptops, PDAs, or even mobile phones, is fostering the diffusion of a new form of distributed computing usually called *mobile computing*. Mobile computing enables a broad spectrum of services to support *nomadic users*, i.e., users who connect to the network from arbitrary locations and who are not permanently connected. As an example, scenarios where nomadic users can read their e-mail or browse the web while moving from a place to another are being enabled by wireless technology.

SEKE '02 Ischia, Italy

This is likely to affect the way people work and what they request to their supporting tools. In a fully nomadic scenario, no fixed network topology can be assumed. This affects the software architecture of the underlying infrastructure. In particular, the pure client-server paradigm, where some machines play the role of service providers for other machines, appears to be unsuitable to enable the required intrinsic dynamism. Machine disconnection is not an exceptional case. Computational nodes are often forced to accomplish their computations *off-line*, and this, in turn, requires some form of reconciliation with the *on-line* part of the environment when disconnected machines later rejoin the network of nodes. In particular, the presence of two classes of computational elements, namely, clients and servers, is a weak point: a service cannot be exploited every time servers are not available. For this reason, researchers of mobile computing were recently attracted by the so-called *peer-to-peer* architectures, where all nodes are peers, i.e. they are functionally equivalent and any could provide services to any other [10]. When the topology of the network environment is not known *a priori*, peer-to-peer settings have some advantages:

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

These advantages are available at the cost of the loss of the centralized control. Distributed coordination in a network environment

can be the source of difficult problems, as the proof of the impossibility of a *distributed consensus* [8] demonstrates. Nonetheless, it is possible and convenient to build middleware components that provide primitives aimed at supplying a common framework where coordination and cooperation of peers is facilitated, and the changing network topology is hidden. In this paper we assume the existence of such a middleware. In particular (see Section 3.1),we base our work on PEERWARE, a middleware suitable for peer-to-peer settings developed at Politecnico di Milano [5, 1].

The increasing availability of distributed computational infrastructures is affecting the way software itself is developed. Software production is often dispersed among geographically distant locations, and software processes become necessarily network aware cooperative processes. Software development environments, however, are still far from supporting these new forms of virtual workgroups through specific network-aware services [7]. As an example consider the case of configuration management (CM) tools. Software processes are typically supported by CM tools [2] that help developers to keep consistent their work, and, despite of dynamic nature of software teams, these are typically client-server applications. For example, one of the simplest, but most popular of these tools, i.e., CVS [6], is based on the concept of the *artifact repository*. This is a central database of configuration items, accessed by developers that *check-out* their working copy from it, and *check-in* a new version incorporating their work. Clearly, the availability of a repository machine is critical, because without it no check-out or check-in operations are permitted, even in the frequent case that no concurrent work is done on a particular item.

In this paper we describe a configuration management tool suitable for the untethered scenarios typical in a mobile environment. The scenario envisions a number of homogeneous peers that are able to provide the same services, disconnect frequently from the net, and perform part of their work while disconnected. In Section 2 we describe the requirements for what we want from such a tool. In Section 3 we describe the architectural structure of the tool. Finally, in Section 4 we draw some conclusions.

2. REQUIREMENTS FOR THE SUPPORT TOOL

In a cooperative work effort, configuration items are parceled among collaborators. In general, for each item we can distinguish an *owner*, who has created the artifact or who has the duty of carrying on the work on it. However, there are typically other workers who need or want to manipulate items that are not under their control, i.e., artifacts they do not own.

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 software processes are concerned, this means that the entire service is blocked, until servers arise again to bring new life in the developers' work.

In a highly mobile settings, disconnected work is not an exceptional case. Developers may wish to check-out the modules they need also when the owners are not connected to them. This, of course, requires the system to provide support for some caching policy. Moreover, check-in should be a transparent operation, which should not require knowledge of who is on-line when the operation is executed. A check in request should be executed asynchronously when the owner of the item becomes available on-line. Finally, when new versions of configuration items that are under one's control become available on-line, a notification should be submitted to all interested peers, to enable them to keep an updated view of the system.

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

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

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

3. AN ARCHITECTURAL VIEW

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

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

Two different architectural choices are feasible:

$$\bigcap_{i=1}^{n} R_i = \emptyset \quad (1)$$

$$\bigcap_{i=1}^{n} R_i \neq \emptyset \quad (2)$$

The choice (1) gives a system with no replicated information. This solution allows efficient implementations (see for example DVS [3, 11, 12]) and does not introduce the risk of getting inconsistent replicated information. However, items can be accessed only if the

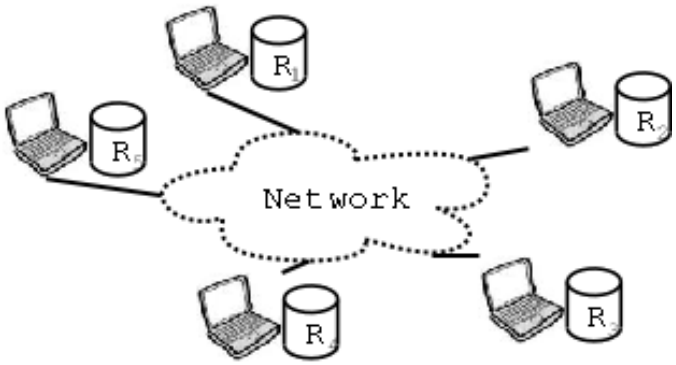

Figure 1: Distributed repository of configuration items

unique host that provides them is on-line, and this does not satisfy our requirements about check-out.

The choice (2) suits better our requirements, but it needs more machinery to compose conflicts among different versions of configuration items. The situation is similar to the Domain Name System [9], in which the data regarding associations between IP numbers and host names are replicated on several DNS servers. DNS servers do not rely on one large centralized name repository. Instead, 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.

The same strategy can be applied to the implementation of a distributed configuration management system. Each peer is *authoritative* for the configuration items it owns, and its copy of such items is the "master" copy. Every check-in of a new version becomes definitive only if it is authorized by the authoritative peer. If a peer X wants to check-in a document whose authoritative peer is A ($\neq X$) two cases may occur:

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

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

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

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

The operations we described here to support distributed configuration management are implemented on top of a middleware which is sketched in the next section.

3.1 The underlying middleware

PEERWARE [5, 1] 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.

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

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

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

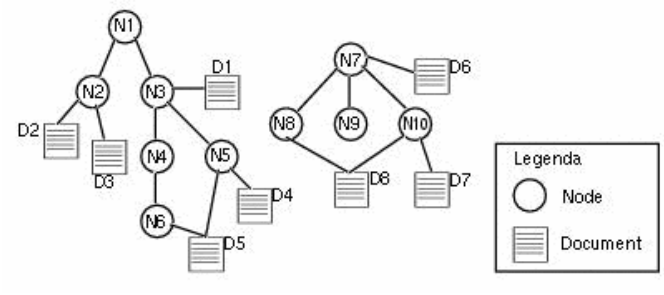

Figure 2: An example of the PEERWARE data structure managed by a peer

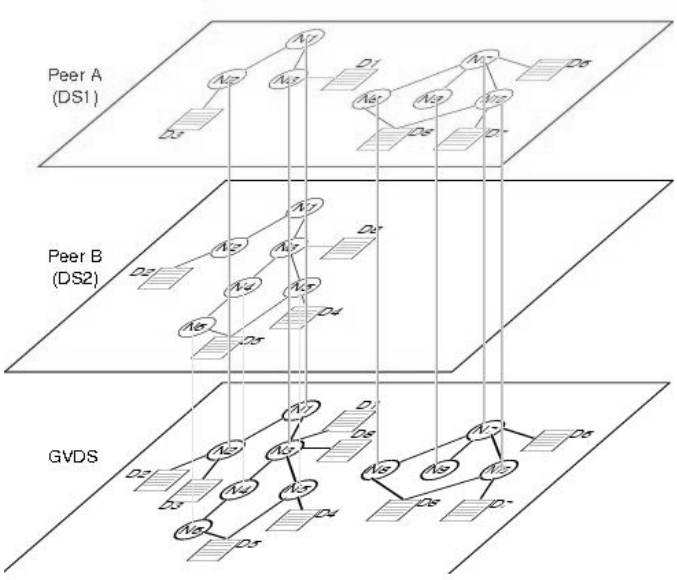

Figure 3: An example of the global virtual data structure managed by PEERWARE

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 [13]. In particular the operations for creating or destroying a node ($createNode(node,parent)$, $removeNode(node)$), for inserting or removing a document ($placeIn(node,document)$, $removeFrom(node,document)$), and for publishing an event occurred on an item ($publish(event,item)$) are defined only on the local data structure. PEERWARE provides three operations that can be performed both on the local and the global data structures:

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

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

The operations provided by PEERWARE together with a publish-subscribe engine on which PEERWARE itself relies on (the distributed event dispatcher JEDI, see [4]) build the framework needed to implement the configuration management operations described in Section 3. In particular, by using PEERWARE we can abstract from the actual network topology and perform actions on *on-line* items.

4. CONCLUSIONS AND FUTURE WORK

We have discussed how a highly distributed system supporting mobile users can provide an infrastructure for cooperative design activities. In particular, we presented the design of a configuration management tool which is especially oriented to supporting scenarios in which users' connectivity to the network can change dynamically. A possible scenario is that a software development team, provided with laptops and some kind of wireless connection, sets up impromptu meetings during which they can, for example, correct bugs on the fly and merge the patches in the baseline.

In these contexts, we cannot assume that a number of hosts, acting as servers, are always available on-line. The absence of a host

is not an exceptional case, but rather the normal case. The traditional solution to the configuration management problem, which is based on a client-server architecture, is clearly not suitable for these kinds of scenarios: it exposes the system to failures when the server is unavailable. Along the same line, repositories based on distributed file systems in which each item exists in only one location suffer from analogous limitations. Instead, we built our system on a peer-to-peer middleware able to provide the abstraction of a global virtual data structure, i.e., a data structure composed by the data contributed by the peers that are connected in a given time instant. Our solution is based on caching copies, and therefore making them available for use even if the hosts that own them are disconnected. The outcome is a genuine peer-to-peer architecture, where any on-line machine can in principle replicate the unavailable resources. We pay, of course, this advantage in terms a harder coordination effort.

The approach we described in the paper is being implemented as part of our current efforts in the provision of a suite of software process support tools well suited for the challenging scenarios of mobile work

5. ACKNOWLEDGMENTS

This work has been supported in part by a grant from Microsoft Research and in part by a grant from Compaq. We would also like to thank the anonymous reviewers for their comments on this paper.

6. REFERENCES

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

[2] E. H. Bersoff. Elements of software configuration management. *Software Engineering*, 10(1):79–87, 1984.

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

[4] 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.

[5] G. Cugola and G. P. Picco. Peerware: Core middleware support for peer-to-peer and mobile systems. submitted ESEC'01, 2001.

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

[7] J. Estublier. Software configuration management: A road map. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, May 2000.

[8] M. Fisher, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):274–382, 1985.

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

[10] A. Oram, editor. *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*. O'Reilly & Associates, first edition, Mar. 2001.

[11] 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.

[12] A. van der Hoek, D. Heimbigner, and A. L. Wolf. A generic, peer-to-peer repository for distributed configuration managemenet. In *18th International Conference on Software Engineering*, page 308, Berlin - Heidelberg - New York, Mar. 1996. Springer.

[13] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, Berlin, 1997.