Adaptive Consistency and Awareness Support for Distributed Software Development*
(Short Paper)

André Pessoa Negrão, Miguel Mateus, Paulo Ferreira, and Luís Veiga

INESC–ID/Técnico Lisboa
Rua Alves Redol 9, Lisboa, Portugal
{andre.pessoa,mcm}@ist.utl.pt,
{paulo.ferreira,luis.veiga}@inesc-id.pt

Abstract. We present ARCADE, a consistency and awareness model for Distributed Software Development. In ARCADE, updates to elements of the software project considered important to a programmer are sent to him promptly. As the importance of an element decreases, the frequency with which the programmer is notified about it also decreases. This way, the system provides a selective, continuous and focused level of awareness. As a result, the bandwidth required to propagate events is reduced and intrusion caused by unimportant notifications is minimized. In this paper we present the design of ARCADE, as well as an evaluation of its effectiveness.

Keywords: Distributed Software Development, Replicated Data Management, Continuous Consistency, Interest Awareness.

1 Introduction

The dominant approach to work synchronization in Distributed Software Development (DSD) is to use a Version Control System, such as CVS [11]. With this approach, programmers work in an isolated environment for most of the time, sporadically synchronizing their work with their colleagues. Despite its widespread adoption, this approach presents two important drawbacks. First, the concurrent work carried out in isolation by different developers may result in synchronization–time conflicts that take time and effort to resolve. Second, it is widely regarded that maintaining programmers *aware* of what others do greatly improves the development process [3].

The acknowledgement of the importance of awareness led to the design of solutions [9,6,2,4] in which the modifications performed by each programmer are propagated to the others in real–time. Despite the improved awareness of this

* This work was partially supported by national funds through FCT – Fundação para a Ciência e Tecnologia, under projects PTDC/EIA-EIA/113613/2009, PTDC/EIA-EIA/108963/2008, PTDC/EIA-EIA/113993/2009 and PEst-OE/EEI/LA0021/2013.

R. Meersman et al. (Eds.): OTM 2013, LNCS 8185, pp. 259–266, 2013.

solution, blindly propagating all modifications to every participant also has its shortcomings. In large projects, a high number of developers constantly modify the source code. As a result, the rate of notifications presented to the user (most of which are not relevant to his current task) is exceedingly high, leading to a distracting work environment. In addition, the bandwidth required to instantly propagate every update may prevent programmers from working in network constrained devices (such as tablets and laptops), which are becoming pervasive in working environments.

We address these issues with a continuous consistency and awareness model called ARCADE (Adaptive Replication, Consistency and Awareness for Distributed Development Environments). ARCADE dynamically controls the frequency with which programmers are notified of remote modifications to the software project. To do so, it takes into account the impact that an update has on the current task of each developer. Then, based on impact, it assigns priorities to each update, such that i) updates with high priority are propagated frequently and ii) updates with lower priorities are postponed for a period of time that depends on the impact factor. As a result, developers are more frequently notified about updates that affect their work more. In addition, network resources are used more efficiently, as savings are achieved by merging and compacting the postponed updates.

This paper is structured as follows. In Section 2 we relate ARCADE to previous work. In Section 3 we describe the model and architecture of ARCADE. In Section 4 we discuss our implementation of ARCADE as a plugin to the Eclipse IDE. In Section 5 we present the evaluation of ARCADE. Finally, Section 6 draws some conclusions.

2 Related Work

The idea of improving awareness by providing notifications to programmers in real–time has already been explored [9,2,6,4]. These systems allow for early conflict detection, potentially saving a significant amount of time and effort. However, none of these systems provides a gradual decrease in awareness as the importance of changes diminishes. As a result, they end up feeding the programmer with information that is frequently irrelevant to his current task and, consequently, unnecessarily consume extra bandwidth.

Our solution is also closely related to the notion of *Divergence Bounding* [8]. The Divergence Bounding model allows replicas to diverge, but define the conditions under which replicas are forced to synchronize. Simple solutions include forcing replicas to synchronize periodically [1] or after a maximum number discarded updates [5]. A more sophisticated approach is provided by TACT [12], which limits divergence according to a multidimensional criteria. However, it does not support varying consistency levels based on interest and locality in the data space. VFC [10] and CoopSLA [7] unify the multicriteria approach of TACT with locality awareness techniques. Both systems provide a variable degree of consistency based on the distance between a user's observation point

and the location of the remote updates. In VFC distance corresponds to the metric distance over a coordinate space in a multiplayer game; in CoopSLA it corresponds to the distance within the tree structure of a text document in a co-operative editor. While both approaches are suitable for their domain, they fail to capture the highly complex dependencies of a software project, as they only consider the distance between the objects of the system, ignoring their semantic relation.

3 ARCADE

In this section we describe ARCADE's model and architecture. Our design is applicable to various object-oriented programming languages. However, throughout the rest of the paper we assume a Java–based DSD project.

3.1 Location and Impact

An object oriented project is composed of numerous entities (e.g., classes and methods) with different types of relations (e.g., inheritance) that can be expressed through a *dependency graph*. At any moment during the development process, we can map the activity of a programmer to a specific line of code (e.g., the one in which the cursor lies). This line, in turn, is a part of a code block that corresponds to one entity of the dependency graph. Hence, we can map the current focus of a programmer to a node of the graph. In the context of our work, we call such a node a *location*. We consider two types of locations. *View location* corresponds to the user's *observation point*, i.e., the location in which he is more interested. *Input location* refers to the location in which he is making changes. Typically, the view and input locations of a single programmer coincide. Most importantly, however, our approach is concerned with the relation between a programmer's view location and the input locations of the other programmers.

The dependency graph of an application shows the nature and strength of the relation between any two locations. From the graph, for any update concerning input location B, we can infer its importance regarding a programmer with current view location A. We refer to this relative importance between locations A and B as *impact*. Based on impact, we can control the frequency with which programmers receive the updates to the different locations of the project.

To exemplify, consider that programmers P_1 and P_2 are editing classes C and D, respectively. If D is a sub-class of C, it is likely that P_2 is interested in receiving the updates performed by P_1. If, on the other hand, the two classes are not related, it is likely that the work performed by one programmer does not affect the other. However, they should still be loosely informed about each other's work, in order to maintain a global level of awareness.

3.2 Controlling Notification Frequency

To clearly define the frequency with which programmers are notified, we introduce the notion of a *priority scale*. A priority scale consists of a set of monotonically decreasing priority levels to which locations are assigned according to

their impact factor. Each level of the priority scale defines under which conditions an update to a location assigned to that level is allowed to be postponed. High impact locations are assigned the highest priority level and, consequently, the programmer is more frequently notified about events referring to it. As the impact of the locations decreases, they are assigned to lower priority levels; as a result, updates are postponed for longer intervals and, if possible, merged or discarded.

In each priority level, the conditions for postponing updates to a location are specified by three parameters: *time* (θ) defines the maximum amount of time a programmer can stay without being informed of changes to a location; *sequence* (σ) limits the number of updates that can be postponed or discarded without notifying the programmer; and *value* (ν) limits the divergence between the local copy of a location and its most recent state, according to some application-dependent metric. When any of these constraints is violated, every postponed update to the corresponding location is sent to the developers concerned.

In the same project, there may be a single priority scale shared by every user or multiple priority scales for different user groups. Even with a single scale, it is important to note that the enforcement of such scale is performed independently for each programmer. As such, even if two programmers have a given location in the same priority level of their particular scales, the exact timing with which they are notified of modifications to that location may differ. The reason is that the notification timing of a location depends on i) when the location is assigned to each programmer's priority level and ii) the elapsed time since updates concerning the location were propagated to each programmer. For these same two reasons, the notification timing of two different locations placed in the same priority level of a single programmer may also differ.

3.3 Architecture

In ARCADE, a single server provides the consistency and awareness management service to a group of clients running at the machines of the programmers. Both the clients and the server maintain a full replica of the project. The server replica contains, at any moment, the authoritative (i.e., the most recent) version of every element of the project. On the other hand, the client replicas have an outdated view of the project, managed according to the priority scale of each programmer. Despite the logically centralized architecture, the deployment of the system can be fully distributed (e.g., by having one client acting as the server).

It is the server's job to hold information regarding the view locations of all users, conduct update propagation and consistency enforcement, and maintain a continuously up-to-date representation of the dependencies among project elements. As operations arrive, the server measures its impact over the work of each developer of the current session. An incoming update is immediately applied to the local replica of the server and then stored until the consistency constraints force it to be sent to a particular client.

The two main components of the ARCADE server are the *Dependency Manager* (DM) and the *Consistency Manager* (CM). The DM is responsible for

Input Location	Same method	Used method/ Same method in superclass		Same class		Superclass		Used class	Same package	Other
Modification Focus		Signature	Body	Declaration/ Field	Method	Declaration/ Field	Method	Declaration/ Field	Any element	Any element
Impact Factor	1	1	3	1	2	1	3	1	4	5

Fig. 1. Impact factors

Priority Level		1 (Critical)	2 (High priority)	3 (Medium priority)	4 (Low priority)	5 (Minimum priority)
Notification Requirements	θ	1 second	10 seconds	30 seconds	60 seconds	120 seconds
	σ	1 update	5 updates	20 updates	50 updates	100 updates
	v	-	10%	20%	30%	50%

Fig. 2. Priority scale

building and maintaining the dependency graph of the software project. The CM is in charge of, for each client, translating the dependency relations between the client's view location(s) and the project's elements into levels of impact, as well as ensuring the corresponding awareness and consistency requirements.

Clients are responsible for dispatching the operations performed locally by the programmer to the server, as soon as they occur. In addition, the client translates, through a *Notification Manager* component, the updates received from the server into notification messages that the final user is able to understand (see Sect. 4).

4 Implementation

We implemented our approach as a plugin to the Eclipse IDE called *ARClipse*. ARClipse was written in Java for the Galileo version of Eclipse. In this section we discuss the most relevant aspects of the implementation of the plugin.

The impact factors and the priority scale considered in ARClipse are shown in Figures 1 and 2, respectively. We considered five impact levels, with level 1 having the highest impact and level 5 the lowest. Our impact function is based on our own notion of relevance in a Java project. However, ARCADE is flexible enough to allow different implementations. Similarly, any other distribution of impact factors across the priority levels, as well as a different priority scale, is also possible.

It is worth noting that impact also depends on the type of update. In most cases, only changes to a method signature are actually relevant; modifications to its body are, typically, not relevant for programmers using or even extending it. However, those programmers may still benefit from receiving (less frequent) notifications about the latter, which ARClipse allows.

To ensure that the updates received do not leave the local project in a non-compilable state, ARClipse maintains a dual view of the project: the programmer

is notified about and can see the remote updates in the IDE, but they are not considered when the project is compiled, unless the programmer explicitly accepts them. We provide two modes of presenting remote updates. In *real–time* mode, remote updates are immediately visible in the form of comments: to each line of code containing a remote modification, ARClipse adds the prefix "//[R]". In *user–time* mode, remote modifications are shown in a separate window.

ARClipse adds two types of visual notifications to Eclipse, pop–up dialogs and new file icons. Pop–ups are shown to actively notify the user of remote updates. To minimize intrusion, ARClipse provides two types of dialogs. When a high impact update is received, a *warning dialog* containing the identification of the updated location is shown in the IDE. In addition, the icon of the corresponding file is tagged with a red circle. A *summary dialog* is presented periodically to the programmer, showing a list with the updates received since the last summary. The icons of files with non-critical modifications are tagged with a yellow circle.

5 Evaluation

We conducted a series of experiments to measure: i) the network performance (in terms of messages exchanged and data transferred) and ii) the CPU and memory utilization of ARCADE. The experiments were conducted on two Dual Core machines connected through a Gigabit Ethernet LAN. One machine executed the server, while the other executed a variable number of bots simulating the behaviour of programmers.

Each simulation consisted in a 30 minutes session, in which we recorded the results obtained by ARCADE and inferred the results that would be obtained by a solution that propagates every update in real-time. This solution is representative of DSD systems that provide non–variable continuous awareness. The simulations were repeated, at least, five times; further executions were conducted, when necessary, to remove outliers. The results presented correspond to the averages of the several executions.

5.1 Network Usage

Figures 3 and 4 present the results obtained regarding the number of transferred messages and the overall network traffic, respectively. The figures clearly show that ARCADE is able to significantly reduce the usage of network resources when compared to the baseline system. On average, the savings obtained surpass 50%.

In addition, the results show that savings do not decrease as the number of users increases. These results are due to the fact that different bots edit, typically, in different modules of the project. This behaviour tries to mimic the real patterns of DSD, in which the programmers added to the development team are typically assigned new tasks that are mostly independent from the other programmers' work (e.g., develop a new module). As a result, the work developed by the new programmer does not introduce a significantly higher impact to other programmers.

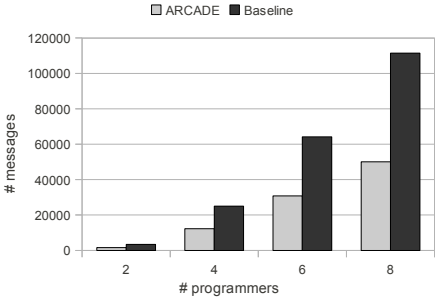

Fig. 3. Total propagated messages

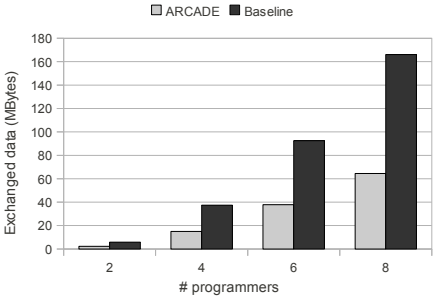

Fig. 4. Total data transferred

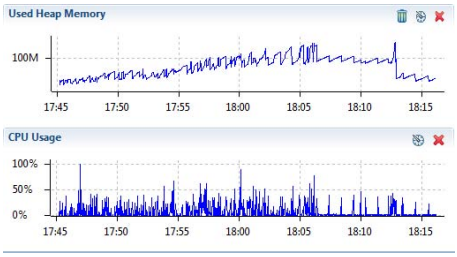

Fig. 5. Usage of client resources over time

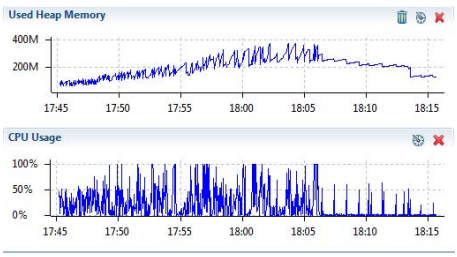

Fig. 6. Usage of server resources over time

5.2 System Resources

Figures 5 and 6 show the results measured at the client and the server, respectively, regarding CPU and memory utilization. Memory utilization at the client stays below 100 MB for most of the duration of the simulations. This is a low value that most likely will not have a significant overhead on the performance of the system. By comparison, the Chromium browser uses over 120MB to display a single Flash web page. At the server side, memory utilization is higher, as expected, due to the fact that the server retains a large number of updates that can only be safely removed when they are propagated or discarded. However, even at the highest load of 8 programmers, memory usage never surpasses 400MB.

Regarding CPU, the load at the clients averages 30% and very rarely exceeds 50%. Similar values are displayed by Chromium when presenting the page mentioned above. At the server, CPU load averages over 50%, with several spikes achieving close to 100%. These spikes are a direct consequence of the cyclic nature of ARCADE, in which a high number of previously retained updates needs to be periodically processed and sent to the programmers. These values are, nevertheless, acceptable for a dedicated server.

6 Conclusion

In this paper we described ARCADE, a new synchronization model for distributed software development. ARCADE assesses the impact that a remote update has on the task undertaken by a developer according to his current focus of work. Based on the measured impact, ARCADE determines if the update should be immediately sent to the developer or postponed for an interval that depends on the impact factor. As a result, ARCADE selectively increases the level of awareness provided to each developer by informing him more quickly about relevant changes. Furthermore, ARCADE is able to compress the log of postponed updates, thus reducing the overall network traffic. Our evaluation results show that ARCADE has great potential in terms of network savings, without requiring a significant increase in CPU and memory utilization.

References

1. Alonso, R., Barbara, D., Garcia-Molina, H.: Data caching issues in an information retrieval system. ACM Trans. Database Syst. 15(3), 359–384 (1990)
2. Cheng, L.T., de Souza, C.R., Hupfer, S., Patterson, J., Ross, S.: Building collaboration into ides. Queue 1(9), 40–50 (2003)
3. Dourish, P., Bellotti, V.: Awareness and coordination in shared workspaces. In: Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work, CSCW 1992, pp. 107–114. ACM, New York (1992)
4. Fitzpatrick, G., Marshall, P., Phillips, A.: Cvs integration with notification and chat: lightweight software team collaboration. In: Proceedings of the 2006 Conference on Computer Supported Cooperative Work, CSCW 2006, pp. 49–58. ACM, New York (2006)
5. Krishnakumar, N., Bernstein, A.J.: Bounded ignorance: a technique for increasing concurrency in a replicated system. ACM Trans. Database Syst. 19(4), 586–625 (1994)
6. Molli, P., Skaf-Molli, H., Bouthier, C.: State treemap: An awareness widget for multi-synchronous groupware. In: Proceedings of the 7th International Workshop on Groupware, pp. 106–114. IEEE Computer Society, Washington, DC (2001)
7. Negrão, A.P., Costa, J., Ferreira, P., Veiga, L.: Semantic and locality aware consistency for mobile cooperative editing. In: Meersman, R., et al. (eds.) OTM 2012, Part I. LNCS, vol. 7565, pp. 380–397. Springer, Heidelberg (2012)
8. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comp. Surv. 37(1), 42–81 (2005)
9. Sarma, A., Noroozi, Z., van der Hoek, A.: Palantir: Raising awareness among configuration management workspaces. In: International Conference on Software Engineering, p. 444 (2003)
10. Veiga, L., Negrão, A.P., Santos, N., Ferreira, P.: Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. Journal of Internet Services and Applications 1(2), 95–115 (2010)
11. Vesperman, J.: Essential CVS. O'Reilly Media, Inc. (2006)
12. Yu, H., Vahdat, A.: Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Trans. Comp. Syst. 20(3), 239–282 (2002)