Vector-Field Consistency for Collaborative Software Development

Miguel Mateus, Paulo Ferreira and Luís Veiga
Instituto Superior Técnico - UTL / INESC-ID Lisboa

Abstract. Software development is, mostly, a collaborative process where teams of developers work together in order to produce quality code. Collaboration is, generally, not an issue, as teams work together in the same office or building. However, larger projects may require more people, who might be spread through-out different floors, buildings and different companies. Several systems have been developed in order to provide better means of communication and awareness over the actions of others. Still, most of them rely on an *all-or-nothing* approach: where the user is either immediately notified of all modifications occurring in a shared project, or is completely oblivious to all external changes.
We propose a new solution based on the adaptation of the *Vector-Field Consistency* algorithm which relies on two distinct concepts: *locality-awareness* and *continuous consistency model*. Where the former represents the ability of system to make choices based on the proximity of remote changes in relation to a particular user's position. While the later corresponds to a consistency model between strong and weak consistency, which is able to control and impose a limit over how much two replicas can diverge. With the correct parametrization this model can establish a great balance between consistency and availability.
We describe in detail how our architecture was applied to the Eclipse IDE, under the form of a plug-in, to provide a new level of distributed collaboration to software developers, and how it was evaluated.

1 Introduction

Coordinating the efforts of multiple teams working in parallel on a software engineering project or module is a non-trivial task. A considerable fraction of the effort in software development is still wasted resolving conflicts, which are only detected when the work of the separate teams or elements is merged. Having a tool able to provide nearly real-time awareness would greatly ease the management of these conflicts, and could even reduce the occurrence of some by allowing the developers to anticipate them.

Distributed collaborative tools are a great way of providing support for interaction among developers and, thus, mitigate the communication deficit originated from physical distance. However, whenever dealing with distributed systems it is necessary to deal with the issue of consistency, as all elements associated with the development of the project must have the same view of the source code. Enforcing consistency requires additional communication; consequentially it is impossible to provide each individual with the modifications performed by all others users in real-time.

Several distributed systems provide optimistic consistency (see Section 2), relaxing it and assuming that replicas will eventually converge. Yet, it might not be desirable in certain systems to postpone information based on such vague assumption, and thus, some applications have means of defining how much two replicas are allowed to diverge. However, more than providing up-to-date information among replicas it is necessary to understand the nature of such information and the impact it has on each participant, in order to know how to deal with it. And even though some applications already provide some sense of context, assigning different weights to operations [1], they prove insufficient in the scope of collaborative software development, as they do not provide *locality-awareness*.

This way, an individual would surely be willing to delay the retrieval of less relevant (and thus more *distant*) changes to the code, if he could guarantee that the *closer* the modifications were to its code, the sooner they would be delivered to him. Conversely, changes having a critical impact to its work would be retrieved almost instantaneously. This differentiated degree of update notification offers better communication while keeping network usage low; as savings can be achieved by the merging of overlapping changes.

This work proposes the adaptation of the VFC (*Vector-Field Consistency*) model, previously used in multi-player games, to the distributed collaborative development of software. VFC uniqueness comes from its capability to dynamically change the degree of consistency associated to data elements, based on locality-awareness techniques applied for each user. The consistency degree is parameterized through the definition of 'observation-points'. These points indicate the position of the user, around which consistency is strong; growing weaker as the distance increases.

In Section 2, we first refer to the concept of *Software Configuration Management*, then we study several existing systems that introduce different collaborative features into software development, and finally we survey the state of the art over consistency models. In Section 3, we describe our architecture, showing the most relevant aspects of the adaptation of the VFC algorithm. Later in Section 4 we specifying the main details of the implementation. Section 5 presents the evaluation of the Eclipse plug-in developed. And finally, Section 6 concludes the paper and addresses future work analysis.

2 Related Work

Collaborative Software Development Work can be shared in two distinct ways: by submitting changes to a shared repository [2]; and by allowing others to see the changes performed by a given developer as they occur [3,4]. The distinction between private and public work, more than desirable, is a necessary property in order to provide a stable working environment, controlling and moderating the impact of non-local changes to one's work. However, the transition between these two aspects needs to be carefully executed, as systems require knowing when to share private work of a given developer to others; as well as what to share. Some empirical issues associated with an ill-management of transition between private and public workspaces are covered in [5]. Total isolation might lead to a larger number of conflicts, and can even make an individual's work obsolete when faced with changes from others. However, developers do not want their work progress to always be visible to others, as this would cause an overload of public information that would not be 'relevant' to the current context or activity of the other programmers. Finally, it might not be desirable to make partial changes visible to others, because intermediate states might be inconsistent.

Awareness consists in the understanding of one's surroundings, and in this case, the understanding of what others are doing, and how their actions will affect the rest of the participants. The complexity and interdependency of software systems [5] makes awareness essential for collaborative software development. Several approaches [3,4,6,7] attempt to grant awareness to developers, by implicitly providing information to others based on one's actions. They differ from one another in the type of isolation supplied.

Several systems have been developed in order to provide means of non-collocated collaboration in software development. For instance, *Palantír* [4] detects users who will be affected by a given set of modifications, based on a dependency graph; *State Treemap* [6] helps users to be aware of the divergence at regular intervals of time, controlling it; *JAZZ* [8] uses the concept of teams to limit awareness to a group of potentially interested individuals. The first two work

as auxiliary tools, additional to any development platform; while the latter was developed as a plug-in for an IDE (Eclipse), benefiting from the properties of *contextual collaboration* [9], supporting more sophisticated interruption management schemes and thus reduced friction. These systems provide awareness, avoiding and resolving conflicts early and thus, having the potential to save a significant amount of time and effort that would otherwise be spent in resolving the conflict at a later stage. However, they are all based on an *all-or-nothing* approach, being unable to provide a gradual decrease in awareness as the impact of changes diminishes.

Continuous Consistency Models Optimistic replication systems usually promise higher availability, performance and concurrency by letting replicas temporarily diverge assuming the existence of *eventual consistency*. The *eventual consistency* model states that, when no updates occur for a long period of time, eventually all updates will be propagated through the system and all the replicas will be consistent. Such policies might, however, be considered too vague for certain applications. Hence, definition of a middle-ground between a pessimist and an optimist approach can bring numerous benefits to a large variety of systems. This technique of allowing eventual consistency to a certain degree, called *bounded divergence* [10], is usually achieved by blocking accesses to a replica when certain consistency conditions are not met.

TACT [1] is a middleware layer that accepts the parametrization of consistency requirements. This new approach, tries to explore the continuum between the two extremes of the consistency spectrum by letting applications decide the maximum degree of inconsistency among replicas. If correctly parametrized, this degree might lead to a significant improvement in performance and availability [11].

VFC (Vector-Field Consistency) [12] presents itself as a new consistency model which unifies several forms of consistency enforcement and multi-dimensional criteria to limit replica divergence, with techniques based on locality-awareness. As such, based on awareness, VFC is able to manage the changing degree of needed consistency between replicas. Although considering locality as an accountable factor to manage consistency have already been tried before [13], most previous work adopt an *all-or-nothing* approach, in which objects within a given ranged are considered critical and outside of that range are all discarded. Given a replica, VFC answers this problem by creating several degrees of consistency based on observation points, referred to as *pivots*, around which the consistency is required to be strong. The consistency requirements gradually weaken as distance from the *pivot* increases, defining *consistency zones*.

Operation Commutativity In order to identify concurrency in the collaborative edition of documents, some information must be kept regarding already applied operations. One option is to rely on a history of previous operations, a common solution in *operational transformation* algorithms [14]. However, in [15] this approach is criticized, stating that such solution may require comparing each incoming operation with many previous operations unnecessarily, which might affect performance and also generate unnecessary ambiguities. Instead, they suggest that operations can be associated with the target-object. This way, conflicts will only occur in operations applied over the same or adjacent objects. The *Commutative Replicated Data Type* (CRDT) [16] approach, considers that documents are composed by a sequence of atoms, which are univocally described through means of an identifier that remains unchanged through the entire document life span. The atoms constituting the document can be any time of non-editable element, such as a character. The total order of the atoms must represent the order by which they appear on the actual document. The purposed implementation for this concept is mentioned in the same article, and is entitled *TreeDoc* [16]. The *TreeDoc* represents a document as a structure of atoms organized in a binary tree, where the left branch of a given atom corresponds to document positions prior to that atom, and the right branch to positions

after that atom. The total order of each element of the tree can be obtained by traversing the tree in infix order.

3 Architecture

The main goal of this work is to enrich an existing *Integrated Development Environment* with a new distributed collaborative concept, based on the adaptation of the *VFC* [12] algorithm. This new concept provides a higher level of awareness over the overall state of a distributed project while, at the same time, trying to ensure the lowest degree of intrusion possible to the programmer's work; as well as reducing the bandwidth usage and network latency.

VFC for Collaborative Software Development The *VFC* algorithm is based on several key entities, that must be adapted to the new context. Namely, *replicated objects*, *pivots*, *consistency zones* and *distance*. Probably the adaptation with greater impact is based on the fact that the *distance* (and therefore *consistency zones*) can no longer be measured by the distance between two coordinates, as spacial distance no longer makes sense in the scope of a *Java* project. The new relation between the objects will be given by their *semantic distance*.

The concept of *pivot* is also subject to significant changes. Every *Java element* is now a *replicated object* an can at any point be assigned to a *pivot*. Additionally, a single user might have more than one *pivot* assigned at the same time.

In previous applications of *VFC*, changes were silently applied to the local replica in a way that was completely transparent to the user. However, simply updating the state of the project on the background might not only be insufficient as it might have a high negative impact on the programmer's work. In such cases, it might be desirable by the programmer to have some sort of mechanism that, without being exceedingly distracting, could provide him with a significant level of awareness regarding where the changes are happening in the project. In the VFC approach, a *consistency zone* has three type of constraints associated to it, when any of this constraints are violated, all operations stored in that zone are sent to the user. These are: *Time*(θ), defining the maximum amount of time a user can stay without being informed of the changes; *Sequence*(σ), limiting the maximum number of unseen updates; and *Value*(ν), limiting to how much a client can diverge from the server replica.

System Architecture The architecture supporting the adaptation of the *VFC* algorithm, which we call *VFC-IDE*, is based on a *client-server* architecture. For each project, one of the many programmers holding a replica of the project can initiate a VFC session (acting as the server-replica). This server-replica is the one in charge of enforcing the VFC consistency algorithm among the multiple replicas of a project; receiving the submitted changes from all clients and managing update propagation based on a star topology.

Server The server is the user instance in charge of receiving and managing requests from all other replicas of a given project, as well as assuring inter-replica consistency. It is the server's job alone to: hold information regarding the multiple pivots of all the session users, conducting update propagations and VFC enforcement, and maintaining a continuously up-to-date representation of the dependencies among project artifacts. As operations arrive, their impact over the work of each user of the current session is measured. The incoming operation is immediately applied and then stored, without any kind of transformation, until consistency constraints demand it should be sent to a specific replica. Ergo, the server keeps, at all times, a view of the overall state of a project, composed by the changes applied in every user's local replica.

The two main components of the Server instance are the *Dependency Manager* and the *Consistency Manager* (see Figure 1). The former translates the various *Java elements* composing a project into special artefacts that related to each other, creating a dependency structure. It is responsible for detecting artefacts that were changed, created or deleted, and updates the dependency structure. The latter, is in charge of, for each user, translating dependency relations between pairs of changed artefacts into levels of impact. This impact is then used to determine to which of the user's consistency zones the operation belongs to. The consistency manager possesses all the knowledge over where each participant of the session is currently located, as well as what observation points they have explicitly declared.

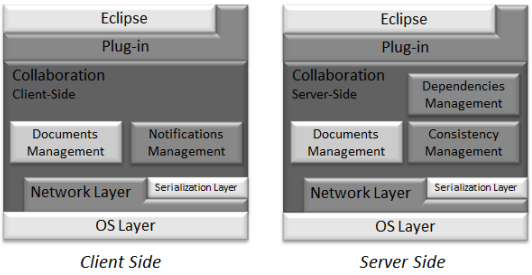


Fig. 1. VFC-IDE Architecture

Client A client is any user who successfully enters a VFC session. These instances are responsible for dispatching local operations to the serve, as soon as they occur. To feed the location-aware mechanism, a client must inform the server of his points of interest (*pivots*). This can be performed implicitly, as the server can infer the edition position from document updates; or explicitly, through means of a message designed for that special purpose.

Every time the server detects that a given remote change might have a significant impact on another user's work, a notification message is sent to both users. The *Notification Manager* (see Figure 1) is the client-side component in charge of translating this message into some event that the final user is able to understand. Aiming not to distract the programmer from its work unless it is strictly necessary, the notification messages will be transformed into increasingly intrusive alerts as the level of impact grows.

In both the server and client, for each (textual) document of the project that has changed since the beginning of the session, an instance of a *Document Manager* is created. This component is used to make the architecture completely independent of the structure used to manage a collaborative document edition. Each instance of this component has two versions of the document associated to it: one being the current state of the distributed document, and the other a stable workspace version of the document. However, in the server's side the stable version corresponds to the last compilable state of the document; while on the client's side it is the current state of the local replicated document, containing only changes performed locally or remote changes that were approved by the user. This ensures that the users can produce and test code without worrying about unstable versions of the project, caused by continuous remote changes.

Document Structure In order to ensure that external operations applied locally preserve their original *intention* [17], a document structure encapsulating concurrency control over each replicated document had to be created. We avoid the use of complex concurrency control, while

at the same time providing freedom of edition, through the use of a *Commutative Replicated Data Type* structure known as *TreeDoc* (see Section 2).

Sequential Operations Optimizations can be applied to the *TreeDoc* operations, in order to reduce the number of messages passing through the network, as well as providing some level of balancing to the tree-structure.

Whenever a user pastes a group of characters (as a single action) the normal procedure would be to, for each character inserted, identifying its *docpath* and adding a node to the TreeDoc. This would, however, lead to unnecessary CPU usage, and would also generate a series of nodes with only one right branch. As an optimization we allow the insertion of a sequence of characters to be translated into a *subtree* with its root on the position of the original insertion. Also, when a inserting or deleting a section of text, we may detect a group of nodes that correspond to sequence of consecutive right children and pack them into a single operation; these composed operations have a single *docpath* pointing to the node with lower depth.

Precluding need for Causality Support To be able to ensure that the intention of all operations is preserved regardless of causality constraints, a new type of *TreeDoc* node had to be created. A *Ghost Node* represents a node that was not yet inserted in the *TreeDoc*, but which existence was already inferred by the insertion or removal of other nodes. For instance, if a user 'A' individually inserts three characters in a document (see figure 2), and for some reason the insertion of the last character is the first operation to arrive to user 'B'. The two nodes that make part of the path to the third node are created in the form of *ghost nodes*. A *ghost node* will turn into a live node when the delayed operation finally arrives. Consequentially, this will also allow for remove operations to be executed even before its causal insert arrives the replica. For example, if the same user 'A', in the meantime, deletes the second character (Y), and this operation arrives to user 'B' before the actual insertion of Y does; in this case the *ghost node* will turn into a *dead node*. An insert operation over a *dead node* is always ignored.

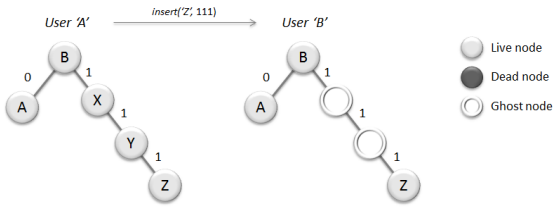


Fig. 2. Ghost node Example - User 'B' receives character Z first

TreeDoc Unbalanced As a result of not being possible to safely remove a *dead* node from the structure, the *TreeDoc* will continuously grow in size throughout the edition of a document. Also, due to the sequential nature of document edition, the *TreeDoc* tends to become highly populated with nodes that have only a right child, thus significantly increasing the depth of the structure.

To solve these problems a couple of operations (*flatten* and *explode* [18]) capable of re-balancing the tree and removing dead nodes already exist. We have, however, taken a different approach. Instead of re-balancing the tree when the document reaches a period of inactivity, we simply remove the TreeDoc. This way, when a server-replica detects that a document has not been changed, by any of the clients, for a long interval, it destroys the TreeDoc associated with

the document, and informs all replicas to proceed accordingly. If, somewhere in the future, a client restarts editing the document, the balanced TreeDoc is immediately generated.

Resource Structure We propose a solution based on the concept of *version-vectors* managed on the client-side to handle concurrency between operations over resource (files or folders). Each replica manages a map containing the project relative paths of every resource contained in a project. Associated with each path there is a *version-vector* which contains, not only the current version of the resource, but also the *id* of the user which originated that specific version. Hence, every time a local resource operation is performed over a given path, its correspondent version is incremented and the version *id* set to the one of the local user. This vector is sent to the server-replica as part of the operation over the resource.

When an external resource operation arrives to a replica, the *version-vector* must be extracted and compared with its local equivalent: *If the remote version is greater then the local one*, the operation can be safely applied and the incoming version-vector becomes associated with the path of the targeted resource; *If the remote version is lower then the local one*, the operation can simply be discarded, and the local version-vector remains untouched; *If the remote version equals the local one*, the version *id* (correspondent to the replica which originated the operation) is used as a disambiguator; where the version with the lowest *id* prevails. *If no local version exists*, the operation is always applied and the incoming version-vector is added to the map of resource paths.

VFC Enforcement The *Consistency Manager* at the server-side of each VFC session is the sole component in charge of enforcing *Vector-Field Consistency* to all the active clients. As soon as a consistency constraint assigned to a given user is exceed, the *consistency manager* must immediately send all the pending operations, associated to the activated zone, to that particular user. Either triggered by the creation of new classes and methods, or simply by instantiating new types or invoking methods inside a class; nearly every single line of code can generate new dependencies and levels of impact between project artefacts. Hence, we periodically identify *dirty* files (files which have been edited since the last check), and recalculate the dependencies for the changed artefacts. This, however, might cause some operations to be assigned to consistency zones based on dependencies that are not completely up-to-date. Hence, we can only assure that the *consistency constraints are consistent with dependencies identified on the last dependency-update*.

4 Implementation

Having in mind factors like, portability, extensibility and community activity (and having preference for an open-source tool), our prototype was implemented on the *Eclipse* IDE. In this section, we specify the most relevant implementation details of our solution.

Real-Time vs User-Time In the context of software development the constant integration of external changes in a local workspace would probably prevent a programmer from ever having a stable version of the project, thus restraining him from the capacity of testing his own code at will. It becomes obvious that external changes must not be immediately applied to the programmer's workspace. Instead, changes that arrive to the programmer's replica of the project must be temporarily kept on hold, and be applied only when it is detected that they will contribute for a new stable version. We provide two distinct modes of edition: *Real-time*, in which external changes arriving the replica are immediately applied to the user's workspace; and *User-choice*, allowing the user to work with some degree of isolation. In the latter, each

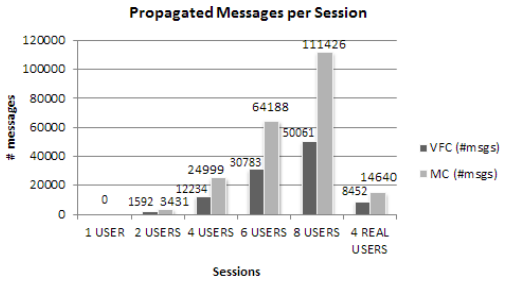
file being edited has two *TreeDocs*, one containing only local changes performed by the user, and the other containing local changes and all pending changes received from the server. These pending changes can latter be applied to the workspace when the user has confirmation that they will lead to a compilable state.

Compilable States The notion of stability (or inter-object coherence) is completely missing in the original VFC algorithm. For the user to know when it is safe to include pending external changes to his workspace, we developed a compilable-states' detection mechanism residing on the server-side, Hence, the server periodically checks if the Eclipse project has any compilation errors. If the project has changed since its last stable version, and there are no errors, than a new compilable version of the project exists. In these cases, an update message is broadcast to all clients. An update message contains only the total number (managed by the server) of the last operation which made the project compilable. This way, when a client accepts the new compilable version, we must check the operation number of all pending operations, and apply to the workspace only the changes that have a number equal or lower to the number contained in the last update message.

Conflict Detection We assume that a conflict occurred, every time a given user performs changes in a line where another user's pivot is currently set. Optionally we can also consider a conflict when two different users are editing the same method, as this operation has a high probability of placing that method in a permanent non-compilable state. Regardless of the reason, each time a conflict is detected by the server, and sent to the conflicting users, a dialog window fades-in at the lower right corner of the screen, alerting the user. To provide the user with a general overview of all conflicting files, a new view was created: *Conflict View*. This view presents all files with unresolved conflicts in the project, allowing users to be aware of possible conflicting actions in *near real-time*, preventing continuous conflicting actions to be performed and only detected in a later stage of the project.

Notifications Conflicts are but one of the types of dialog notifications available in our solution. Every time the server detects that an incoming change has a considerable impact over one user's work, it sends a notification operation to that specific user, with information about the Java Element affected and the associated impact. When the notification arrives the client's side it can be translated into three forms of pop-up dialogs: *Information Dialog*, informing the user of events such as: the remote creation or deletion of file or folder. This pop-up has a neutral colour (blue), fades-in at the lower right corner of the screen and fades out after a certain period of time; *Warning Dialog*, alerting the user of changes with great probability of affecting the user's work, for example: when someone changes the *interface class* of a class where the user is working. This is a yellow dialog, and although also temporary it remains visible for a larger amount of time; *Conflict Dialog*, appearing whenever conflicts are detected. Its colour is red and only fades-out after the user clicks on the dialog.

Pivot in the IDE To aid the users in the creation and destruction of explicit pivots, we developed an additional context menu. Using our plugin, when a user right-clicks in any position of an opened document, one of the available options is the addition of a new pivot. When a new pivot is added, the position in the document is translated into a *Docpath* of the *TreeDoc* and a pivot operation is sent to the server. In the server that *Docpath* is mapped into the Java *artefact* that exists on that position.



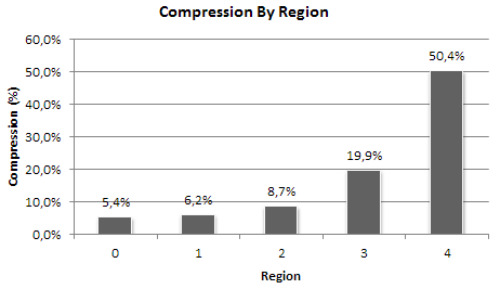


Fig. 3. Total propagated operations per session.

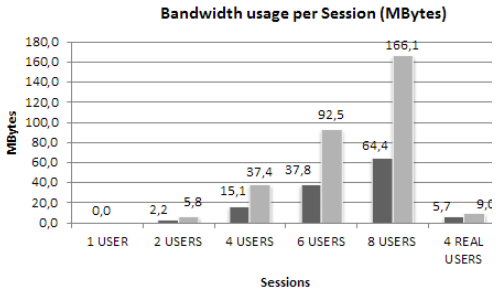


Fig. 5. Total messages sent per session.

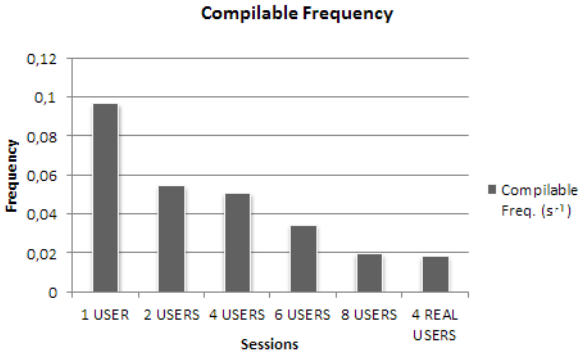


Fig. 6. Frequency of compilable states detection.

5 Evaluation

The VFC-IDE prototype was evaluated in a threefold perspective: qualitative, quantitative and comparative.

Qualitative Evaluation The goal of this work was to enhance the Eclipse Platform by providing a VFC-based collaborative development in the least intrusive way possible. In this section we overview the actual benefits of the continuous consistency model based on the impact of remote changes to the work of a programmer.

The adaptation of the VFC algorithm, to the scope of collaborative software development, was the main focus of our work. And indeed, in practice, we observed the effects of the VFC constraints allowing for certain regions of the project to diverge more than others, based on the point of edition of the user. This gives the user a near-real-time awareness over changes that are likely to affect him. Additionally, there were no lost of performance perceived by the users.

We proved that by using the VFC-IDE plugin, the users can benefited from a higher degree of awareness, based on the selective scheduling of updates, which enables them to immediately detect conflicting operations. Additionally, the features provided by our solution allow for a quicker identification, and resolution, of occurring conflicts.

Quantitative Evaluation The Eclipse plugin was tested in two distinct scopes. In the first, we perfomed 30 minutes sessions using a bot developed specifically for this purpose (*IntelliBot*), and which was designed to simulate the behaviour of a real software programmer; thus being capable of performing valid semantic changes to a Java project such as: create resources, extend classes and interfaces, add and use methods, delete methods and manage attributes of a class. The second set of tests were executed by four real programmers, using one Macintosh running a *Helios* version of Eclipse, and three PCs, each one with a different version of

Eclipse (*Galileu*, *Helios* and *Indigo*). Note that even though the *IntelliBot* tries to mimic the programmer's behaviour, in order to increase the level of stress on the server's side, its degree of activity (measured in operations per second) is far greater than that of a normal programmer. With these tests we evaluated the savings in number of exchanged messages, used bandwidth and effectiveness of the operations compression.

Propagated Operations By measuring the different number of messages emitted from the server, using VFC and MC approaches, we are able to observe a significant number of messages saved by compression. Throughout multiple sessions (with an increasing number of clients) we were able to detect an average compression of 50% of the messages sent by the server (see Figure 3). Even though the compression rate is fairly significant, it seems to be almost completely unaffected by the number of participants of a session. Also, the average compression rate seems to vary very little in time, neither having an increasing nor decreasing pattern as time advances. We can then infer, by cross-checking actual statistics with the behaviour of the compression algorithm, that the level of compression is mainly dependent on the programming style of the participants. Regarding consistency zones, we can notice that. First, as we move to regions with looser consistency levels the compression rate increases; with a major compression improvement in the region with the lowest consistency constraints (see Figure 4); and second, we can see that the *Time* constraint is the predominant constraint triggered in the regions of lowest consistency. With these two premisses, we can predict that constraints triggered by *Time* are associated with a greater level of compression. Consequentially, we are able to deduce that the longer a set of operations is delayed its propagation, the higher the compression rate expected.

Bandwidth Usage Concerning bandwidth usage, by keeping track of the number and size of the messages being dispatched by the server, we were able to conclude that the gains in bandwidth stress are proportional to the reduction in the number of messages studied in the previous section. As it can be seen in Figure 5, the bandwidth savings achieved by using the VFC algorithm correspond to an average of 60% when in comparison to the MC algorithm. As it was mentioned in Section 3, as operations are performed in the same document, the associated *TreeDoc* grows in depth. In turn, this leads to an increase in the size of the *DocPath* of each operation, thus gradually increasing the average size of each message throughout a VFC session. Throughout our tests we identified that, the average size of propagated messages increases by approximately 1KB during an interval of 30 minutes of intensive usage. It is, however, interesting to notice that the evolution of the average message size, in a session with four real users, is visibly less accentuated. This can be caused either by the lesser frequency of the operations performed by real users, or due to the distribution of the operations among a greater number of files, which would inhibit the increasing length of *DocPaths*.

Compilable States We can conclude that the frequency with which compilable states are detected is satisfactory, even in sessions with a significant number of users (see Figure 6); as the lowest frequency values happen in the session with real users, and corresponds to an average distance of 50 seconds between each state (with a maximum of 4 minutes between states). However, it is important to notice that the server is only capable of detecting compilable states if they, indeed, exist. For instance, if a user starts editing a different region of code after leaving the artefact he was currently editing in an un-compilable state, the project will remain un-compilable until the erroneous artefact is fixed. Notice that, as external changes are only added to the programmer's workspace after his explicit approval, each programmer can ensure their own local compilable state at all times.

System Resources Using yet another Eclipse plugin, we were able to monitor the evolution of CPU and memory usage of our solution. In this particular session we divided the duration

of the *IntelliBot*'s activity into two groups: the first group, of three clients, ran the bot for the entire duration of the 30 minutes session; the second group, containing 5 clients, ran the bot only for the first 20 minutes. As expected, this led to an increasing level of stress at the server's side for the first 20 minutes, followed by a gradual decrease of activity. In the client there were very little significant changes to the memory usage, only slightly exceeding the average 100 Mbytes used by an instance without VFC. At the server side we can observe a greater memory overhead, sometimes nearly reaching 400 Mbytes. The accentuated memory usage can be easily explained by the queuing of incoming operations originated from the many different participants, which can only be safely removed from memory once they have been sent to all other users. As in a session the 8 participants using *IntelliBot* the rate by which operations are performed tend to exceed the rate by which they are dispatched through constraint violations, the messages will gradually accumulate until the activity decreases.

Comparative Evaluation Through the observation of the carious tests performed, we are able to conclude that the VFC approach provides significant benefits in terms of the usage of network resources, when in comparison to the Maximum Consistency (MC) alternative. More over, we are able to conclude that the gains increase even more significantly as we impose a greater delay between receiving and sending the updates at the server side. Consequentially, if the bandwidth usage is reduced, we can conclude that, with the VFC approach, we are able to better scale the number of participant in a session. And also, we are capable of running a VFC session in an environment with high bandwidth constrictions.

6 Conclusions

This paper addressed the adaptation of a *continuous consistency* algorithm (Vector-Field Consistency), initially designed for multi-player gaming, to the new scope of distributed collaborative software development. The adapted consistency model was then complemented with a *locality-awareness* algorithm capable of determining and assessing the impact of remote changes to the work of a given programmer; thus determining if remote changes should be immediately sent to the programmer, or postponed. By postponing operations that did not directly affect a programmer's work, we were able to compress the log of pending operations and reduce the total number of messages traveling the network; which produced significant gains in terms of bandwidth usage. On the other hand, by detecting and immediately sending remote changes that had a high probability of affecting the work of a given programmer, we were able to increase the level of awareness of each programmer over the actions of the remaining participants of a collaborative project.

Hence, we can benefit from adjusting the consistency constraints of each region to better fit the characteristics of a project, or group of programmers; thus achieving better compression levels. And finally, we determined that the *Vector-Field Consistency* algorithm holds great potential in terms of bandwidth savings when in comparison to the *Maximum Consistency* alternative, as it is able to intelligently forfeit consistency in order to reduce the number of exchange messages in the network.

Acknowledgements: This work was partially supported by national funds through FCT Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/108963/2008, and PEst-OE/EEI/LA0021/2011.

References

1. H. Yu and A. Vahdat, "Design and evaluation of a continuous consistency model for replicated services," in *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2000, pp. 21–21.

2. P. Cederqvist, “Version management with cvs,” Sweden, 1993.
3. C. O’Reilly, P. Morrow, and D. Bustard, “Improving conflict detection in optimistic concurrency control models,” in *Software Configuration Management*, ser. Lecture Notes in Computer Science, B. Westfechtel and A. van der Hoek, Eds. Springer Berlin / Heidelberg, 2003, vol. 2649, pp. 61–69, 10.1007/3-540-39195-9_14. [Online]. Available: http://dx.doi.org/10.1007/3-540-39195-9_14
4. A. Sarma, Z. Noroozi, and A. van der Hoek, “Palantir: Raising awareness among configuration management workspaces,” *Software Engineering, International Conference on*, vol. 0, p. 444, 2003.
5. C. R. B. de Souza, D. Redmiles, and P. Dourish, “”breaking the code”, moving between private and public work in collaborative software development,” in *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, ser. GROUP ’03. New York, NY, USA: ACM, 2003, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/958160.958177
6. P. Molli, H. Skaf-molli, and C. Bouthier, “State treemap: an awareness widget for multi-synchronous groupware,” in *INTERNATIONAL WORKSHOP ON GROUPWARE*, 2001, pp. 106–114.
7. D. Čubranić and M. A. D. Storey, “Collaboration support for novice team programming,” in *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, ser. GROUP ’05. New York, NY, USA: ACM, 2005, pp. 136–139. [Online]. Available: http://doi.acm.org/10.1145/1099203.1099229
8. L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson, “Jazzing up eclipse with collaborative tools,” in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, ser. eclipse ’03. New York, NY, USA: ACM, 2003, pp. 45–49. [Online]. Available: http://doi.acm.org/10.1145/965660.965670
9. S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson, “Introducing collaboration into an application development environment,” in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, ser. CSCW ’04. New York, NY, USA: ACM, 2004, pp. 21–24. [Online]. Available: http://doi.acm.org/10.1145/1031607.1031611
10. D. Agrawal, A. El Abbadi, and A. K. Singh, “Consistency and orderability: semantics-based correctness criteria for databases,” *ACM Trans. Database Syst.*, vol. 18, pp. 460–486, September 1993. [Online]. Available: http://doi.acm.org/10.1145/155271.155276
11. H. Yu and A. Vahdat, “The costs and limits of availability for replicated services,” *ACM Trans. Comput. Syst.*, vol. 24, no. 1, pp. 70–113, 2006.
12. N. Santos, L. Veiga, and P. Ferreira, “Vector-field consistency for ad-hoc gaming,” in *Middleware ’07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 80–100.
13. K. L. Morse, “Interest management in large-scale distributed simulations,” 1996.
14. D. Li and R. Li, “Preserving operation effects relation in group editors,” in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, ser. CSCW ’04. New York, NY, USA: ACM, 2004, pp. 457–466. [Online]. Available: http://doi.acm.org/10.1145/1031607.1031683
15. H.-G. Roh, J. Kim, and J. Lee, “How to design optimistic operations for peer-to-peer replication,” in *JCIS*, 2006.
16. N. Preguiça, J. M. Marques, M. Shapiro, and M. Letia, “A commutative replicated data type for cooperative editing,” in *ICDCS ’09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 395–403.
17. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, “Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems,” *ACM Trans. Comput.*, vol. 1, no. 5, pp. 63–108, 1998.
18. M. Shapiro and N. Preguiça, “Designing a commutative replicated data type,” Computer Science Dept: University of Copenhagen, Tech. Rep., 2007.