INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Vector-Field Consistency para Desenvolvimento Colaborativo de Software

Vector-Field Consistency for Collaborative Software Development

Miguel Cortez Mateus

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Prof. José Manuel da Costa Alves Marques, (DEI/IST)
Orientador:	Prof. Luís Manuel Antunes Veiga, (DEI/IST)
Co-Orientador:	Prof. Paulo Jorge Pires Ferreira, (DEI/IST)
Vogais:	Prof. Paulo Jorge Fernandes Carreira, (DEI/IST)

Maio 2012

Acknowledgements

I would like to start by thanking my advisor, Prof. Luís Veiga, for the guidance and encouragement, and for being more than understanding about my position as a working student; for all the feedback and all text revisions. Without any of these, this dissertation would have not been finished on time.

I would like to thank my brother, for enduring my many out-loud frustrations, and for all the brainstorming sessions and words of advice, which where cardinal in the execution of this work.

I wish to thank all my friends, for never giving up on me; even when I would go off the radar for long periods of time, throughout all my academic years. For all the invitations sent, even when they clearly knew there was no way I could make it (and for persistently reminding me of what I had lost afterwards).

I would like to thank João "Sta" Costa, André "Stif" Andrade, as well as my big brother, for sparing several hours of their social lives, in order to beta-test my prototype, and for all the feedback given.

A very special thanks to my girlfriend, Sofia, for her never-ending patience and understanding over the times when I would stay locked at home for what would seem like an eternity.

Finally, I wish to thank all my family, which supported me at all times in all the choices I have made so far. I would specially like to thank my parents: Mom, Dad, thank you for your undying support, encouragement and assistance. Thank you so very very much.

Lisboa, June 24, 2012
Miguel Mateus

Resumo

O desenvolvimento de software é, maioritariamente, um processo colaborativo, em que equipas de programadores trabalham em conjunto de forma a criar um produto de qualidade. Assegurar cooperação não é por norma um problema, uma vez que as equipas trabalham em conjunto no mesmo *open-space*. No entanto, projectos de maior dimensão podem necessitar de um maior conjunto de intervenientes, que por sua vez se podem encontrar espalhados por diferentes pisos, edifícios ou mesmo múltiplas empresas.

Diversos sistemas foram já desenvolvidos de forma a enriquecer a comunicação e o nível de *percepção* relativo às acções realizadas por terceiros. No entanto, a maioria destes sistemas baseia-se numa abordagem de *all-or-nothing*: onde o utilizador ou é imediatamente notificado de todas as alterações que estão a ocorrer no projecto partilhado, ou está completamente alheio a estas.

Neste trabalho propomos uma nova solução que tem por base a adaptação do algoritmo *Vector-Field Consistency*, e que é caracterizada por dois conceitos distintos: consciência da localização (*locality-awareness*) e um modelo de consistência contínuo. Em que o primeiro representa a capacidade que o sistema tem de tomar decisões com base na proximidade de alterações externas, em relação ao ponto de edição de um dado utilizador; Enquanto o segundo corresponde a um modelo de consistência que faz a ponte entre consistência forte e fraca, sendo capaz de impor um limite sobre o quanto duas réplicas podem divergir. Com a parametrização correcta, este modelo pode estabelecer um excelente equilíbrio entre consistência e disponibilidade.

O algoritmo *Vector-Field Consistency* (VFC) foi originalmente concebido para o âmbito de jogos multiplayer distribuídos, e vai agora dar os primeiros passos em direcção ao desenvolvimento colaborativo de software. Esta adaptação irá permitir que o programador possua uma maior percepção face a alterações externas que possam afectar directamente o seu trabalho, e conforme o impacto das alterações externas for reduzindo, também o grau de percepção irá diminuir.

Nesta dissertação estudamos, em primeiro lugar, o estado da arte relativo a diversos modelos de consistência, e a ferramentas de gestão de software já existentes; assim como diversas técnicas usadas no desenvolvimento colaborativo de software. Em seguida explicamos como o algoritmo VFC, e outros mecanismos, foram adaptados ao nosso novo âmbito. E por último, descrevemos em detalhe a forma como a nossa arquitectura foi aplicada ao Eclipse IDE[1], por intermédio de um plug-in, com o objectivo de criar um novo nível de colaboração distribuída para programadores, e a sua avaliação.

[1]Eclipse.org: http://www.eclipse.org/platform

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.

The *Vector-Field Consistency* (VFC) was originally applied to the context of distributed ad-hoc gaming, and will now take its first steps into collaborative software development. This adaptation will allow a software developer to have a higher degree of awareness over remote changes which might directly affect his work, and as the impact of changes will gradually grow further away from the developer's task, so will the level of awareness.

In this dissertation we first study the current state of the art concerning consistency models and various already existing software management tools, along with multiple techniques used for distributed collaborative software development. Then we will explain how the VFC algorithm and multiple other mechanisms were adapted into our new context. We then describe in detail how our architecture was applied to the Eclipse IDE[2], under the form of a plug-in, to provide a new level of distributed collaboration to software developers, and how our solution was evaluated.

[2]Eclipse.org: http://www.eclipse.org/platform

Palavras Chave
Keywords

Palavras Chave

Vector-Field Consistency

Modelo de Consistência Contínua

Noção de Localização

Trabalho Cooperativo

Representação TreeDoc

Ambiente de Desenvolvimento Integrado (IDE)

Keywords

Vector-Field Consistency

Continuous Consistency Model

Locality-Awareness

Collaborative Work

TreeDoc Representation

Integrated Development Environment (IDE)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Being today's software development mostly a collaborative process, where individuals work together in order to develop a product of higher quality, communication and coordination become fundamental concerns. Generally, the rich person-to-person interaction afforded by shared physical workspaces allows people to maintain an up-to-the-minute knowledge concerning each others' work; this degree of interaction allows a small group of people to collaborate more effectively.

Although it might seem an easy task ensuring such high level of communication and feedback over others' actions when dealing with a small group of elements, working in the same office and possibly in the same room; as projects grow larger and more complex, proximity between developers cannot always be assured. In fact, facing an increasing complexity, projects are usually divided in modules and sub-modules which are then assigned to different teams, possibly from different sectors, or even outsourced. These teams will not always share the same physical space, and thus, as physical distance between elements or teams increases, instant communication will gradually become harder to maintain.

One particularly important aspect of communication, concerning software development, is *awareness*:

> *"Awareness is an understanding of the activities of others, which provides a context for our own activity. This context is used to ensure that individual contributions are relevant to the group's activity as a whole."* [10]

For example, it might be extremely useful for a developer to have knowledge of changes in the code performed by individuals from the same or other teams, regardless of the physical distance separating them. This property is already partially achieved through version control tools [4, 43]. However, coordinating the efforts of multiple teams working in parallel on a module is a non-trivial task. Thus, 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. However, providing *context* is more than simply sharing up-to-date *content*. And so, an alternative way of preventing conflicts is done by ensuring that two developers will never try and change the same section of code at the same time. Current software configuration systems promote workspaces that isolate developers from each other. Such degree of isolation, however, is still susceptible to conflicts when merging modifications, as different workspaces

may share a set of objects. Thus, it is still important to provide an insight of who is changing each file or object, even when working in relative isolation.

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 2.3), 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 [56], they prove insufficient in the scope of collaborative software development, as they do not provide ***locality-awareness***[5].

A program is organized as a hierarchical structure of objects or elements constructs. Each of these typically corresponds to abstractions in a programming language, namely blocks, classes, methods, attributes. These elements usually establish relations among each others, for example a class can extend other classes or interfaces, as well as it can hold several methods and attributes. Hence, a modification over an object will have a different level of impact on the various objects of a project. For example, if an individual (working on a class A) is developing a project in collaboration with two other developers, knowing that one of the developers is working on a super-class of A, while the other is changing a method of a sub-class of A. Surely he will be rather more interested in the work of the first developer, as it might affect his work directly. Hence, it would be fair to say that the first developer's work is *closer* to the individual's. To this semantic degree of proximity, associated with the impact of a remote modification over one's work, we call ***locality***. And so, 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, modifications 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.

1.1 Contributions

This dissertation 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 parametrized 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.

To adapt the VFC model to collaborative software development, we need to adapt it to our new definition of locality. In our new scope, position refers to the section of code a developer is working on, and the distance from other developers' changes is measured based on the relationship among language constructs (class and interface hierarchies, among others), thus allowing a user to have more frequent

feedback of changes to the code that might affect and possibly conflict with his work. The user can benefit from this level of information, avoiding conflicts that otherwise would inevitably occur.

1.2 Results

The main result taken from this dissertation is the adaptation the *Vector-Field Consistency* algorithm to the context of distributed software development. Aside from the most theoretical component of this particular work, we have produced the following:

- A survey of the related work regarding consistency enforcement policies in distributed systems and collaborative software development;
- The adaptation of a commutative replicated data type (*TreeDoc*) capable of managing concurrent editions over the same file, assuring *consistency* and *intention* even if operations are applied out of causal order.
- A plug-in for the *Eclipse IDE*, which enables the collaborative development of a project, based on either *Maximum Consistency* or *Vector-Field Consistency*;
- An evaluation of the success of the implementation in terms of the following criteria:
  - *Qualitative*: based on the functionality provided and how it adapts to the developers' needs, and also through consideration of the level of intrusion over the hosting IDE;
  - *Quantitative*: measuring its efficiency based on factors, such as: the number of messages exchanged, the usage of the bandwidth and the detection of actual conflicts;
  - *Comparative*: comparing the two consistency models: *Maximum Consistency*, and its optimization *Vector-Field Consistency*.

1.3 Document Roadmap

In Chapter 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. Each section of this chapter performs an analysis and comparison of the various alternatives.

In Chapter 3, we show the most relevant aspects of the adaptation of the VFC algorithm, followed by the description of the architectural solution designed. Later, in Chapter 4 we present how the previously designed architecture was adjusted to the Eclipse Platform, specifying the main details of the implementation process, and the various features created. Chapter 5 presents the quantitative, qualitative and comparative evaluation of the Eclipse plug-in developed. And finally, the last chapter concludes the dissertation and addresses future work analysis.

Chapter 2

Related Work

This section is organized in three subsections: The first one introduces *Software Configuration Management*, and refers a set of tools that provide members of a project with means for concurrent development. The second subsection mentions a number of ways of incorporating collaboration in software development, pointing desirable properties as well as common mistakes in the design of such systems. Also it compares some already existing collaborative software development systems. The third and final part, surveys a large variety of consistency models.

2.1 Software Configuration Management

Nowadays software products are becoming larger and more complex. Furthermore, customers are demanding more and with better quality, while at the same time constantly changing requirements. This way, unless properly managed with the right techniques, a software development project can easy fail to deliver a quality product.

As projects become larger and larger, parallelizing work becomes unavoidable. As such, several teams must work in parallel in the same module, or in different parts of the same module. However, eventually teams will have to synchronize their work, and at this point, sometimes it will lead to conflicts. Conflicts occur when two changes, that do not match, are made by different parties to the same fragment of the code. Most of the time they are difficult and time consuming to resolve and they often require manual intervention.

In short, software is hard to manage mainly for three reasons. The first is that developers can easily change code throughout an information system. Second, modifications performed might affect the behaviour of the entire system due to the interdependencies among modules. And third, because development of software is usually divided in teams, the changes one programmer makes often impact the work of others. Software engineers typically try to develop formal procedures that structure the work of building software, in order to reduce the occurrence of such problems. *Software Configuration Management* (SCM) [16] addresses the problems of managing the evolution of software by defining a set of principles and tools that enable to keep an evolving software product under control, and thus contributing to satisfy both quality and delay constraints. Some of these tools include *version control*, *workspace management* and *system building*, which will now be studied in greater detail.

2.1.1 Version Control (file locking)

The main purpose of version control is to manage multiple versions of the code that might exist during the software development process. When considering multiple teams of multiple elements working concurrently in the same program module, it is necessary to define various versions of that given module that can be altered independently by each element. This sort of multi-user management focuses on preventing concurrent changes from overlapping each other. Allowing a developer, who wants to change a file, to create a copy and setting a lock on that file (*check-out*) [7]. Although everyone can still read the file, only that developer can create a new revision for that file (*check-in*). This way, only one developer at a time has write access to the central "repository" copy of a given file.

At *check-in* time, when the developers have completed their changes, they need to merge their code with the version in the repository, allowing the modifications to be seen and used by the remaining elements of the project. The versioning tool supports merging by providing a facility that compares the two files and displays the lines that differ [20]. The developer responsible for merging must then select the lines that need to appear in the integrated module.

File locking has merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

2.1.2 Workspace Management

SCM tools can also focus on helping developers to work in a given file without locking it, and still not being concerned about modifications on the code they are working on; by providing private workspaces [8]. A workspace is simply a region containing a set of files of interest (*w.r.t.* a given task). Hence, in assigning only a single individual to each workspace, it acts as a capsule where the programmer can work, isolated from the outside world, for the duration of a given task, preventing developers from interfering with another's work. When the task is completed, the files from a specific workspace can be safely checked-in with a minor chance of conflicting. However, several workspaces may share a set of files, thus comes the need for synchronization and dealing with conflicts.

Workspaces differ from the file locking approach, in the means used to isolate developers' actions, as the former tries to increase concurrency through a logical division of work in non-conflicting tasks.

2.1.3 Concurrency Control (version merging)

Some tools for version control provide concurrency control, which means that files are not locked when checking them out, and there may be simultaneous modifications to the same files by multiple users. Such tools must provide facilities to merge changes into the repository so the changes from a developer, who is concurrently committing new versions of the file, are preserved when other developers check in. These tools are generally able to identify non-conflicting changes between different versions, concatenating them. Otherwise they display the lines that differ and the developer responsible for the merge needs to manually select the lines that need to appear in the integrated module[4]. Visual SourceSafe (VSS) is an example of an SCM tool in which both locking and concurrency mechanisms are implemented [43].

2.1.4 System Building

When developing a system, a build operation can be done over a subsystem or a module (incremental), or the entire system (from scratch). In incremental building, only files that have been modified since the last build should be rebuilt. SCM tools can facilitate building work by capturing necessary information and making the building process repeatable, based on information of previous versions of a project's files [12].

Software Configuration Management is an essential activity that must take place whenever developing software. With focus on controlling developers' abilities to alter code, configuration management is based on a series of tools that try to ensure that the evolution of the software product goes steadily despite the increasing complexity of software systems and constant changes in the requirements.

2.2 Collaborative Software Development

Software development is, mostly, a collaborative process where teams of developers work together in order to produce quality code. The objective of this process is to allow a group of people to address software development tasks in collaboration, while separated by great geographical distances, thus fuelling outsourcing and off-shore development. The success of distributed teams working together in an effective way is essential, and is a definitive factor deciding the success or failure in a project of high complexity [3].

Distributing work among teams and developers can become especially difficult due to the many inter-dependencies between objects, or artefacts, created during the life of a project. As seen in the previous section, a variety of tools and methods is used to ensure a degree of isolation among the developers, thus reducing conflicts. By doing so, such tools are, in essence, creating a distinction between private and public aspects of a developer's work.

The first, focus on isolating the changes made by a developer from other parts of the code, thus ensuring a developer's work-in-progress is not shared (*isolation*). Conversely, a developer is able to work freely without being affected from changes of others. While the second, allows a user to share his modifications to the code with others. In this case, information can be shared in two distinct ways: by submitting work to a shared repository [4]; and by allowing others to see the changes performed by a given developer as they occur [32, 42].

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 [9]. 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, mainly because they want to appear competent in the eyes of colleagues and managers. Also, because people have several contextualized and different strategies to release their private information, they expect others to do the same; thus avoiding the overload of public information that is not 'relevant' to their current context or activity. Finally, it might not be desirable to make partial changes visible to others, because intermediate states might be inconsistent.

Awareness

Whenever dealing with public workspaces the topic of awareness comes to hand. In collaborative software development, awareness of others, and their work, provides information that is crucial for smooth and effective collaboration. 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. This property is particularly useful for coordinating actions, managing coupling, discussing tasks and anticipating others' actions. The complexity and interdependency of software systems [9] makes awareness essential for collaborative software development. Although awareness is easily ensured in face-to-face work, as projects grow larger and more complex, proximity between developers cannot always be assured, and thus, communication and implicit information that is available to a co-located team becomes difficult to guarantee [18].

Three techniques may help developers maintaining a certain degree of awareness despite of the geographical distance separating developers:

- *Explicit communication*: Participants of this communication channel tell each other about their activities and intentions;
- *Consequential Communication*: A developer can observe the work of others and, thus, obtain information about their activities and plans;
- *Feedthrough*: A developer observes changes to project artefacts indicating who has been doing what.

Awareness through *explicit communication* is, by far, the most flexible technique and the easiest to implement, as it can be achieved by complementing the software already in use with external tools such as email, text chat, instant messaging, etc. However, an explicit mechanism of communication requires additional effort from the participants and thus, many collaborative applications aim for the alternative techniques [29, 32, 42, 53]. The latter approaches 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. For example Night Watch [32] completely eliminates the notion of private work by making all aspects of the work publicly available to others (constantly publicizing information). In opposition, State Treemap [29] allows users to decide when their work is stable and refined enough to be public to others.

Although, at a first analysis, such level of awareness might look like an advantage, as previously discussed, the need for isolation and for controlling the release of private information is an important aspect in any collaborative tool. For example, in a project involving a large number of developers, changes will be a constant and may occur in independent sections of the code. Therefore, an individual working in a module of this project is, probably, not interested in being overwhelmed by information that is not relevant to the scope of his work. Thus, dependencies between developers' working activities are a cause for concern and attention.

Change Impact Analysis

More than providing up-to-date information between developers it is necessary to understand the nature of such information and the impact it has on each participant.

In [2] several techniques to measure the impact of remote modifications to the code are presented. One example is the dependency graph approach, which tries to determine the impact of a given change in the code by defining a set of dependences associated with the logic of the target-program; for example, syntactic relationships between the statements of a program representing aspects of the program's control flow and data flow. Also, it can be found in [39] a collection of new object-oriented techniques for determining the effects of a set of source code changes, assuming classes, methods, fields and their interrelationships as the atomic units of change.

Hence, collaborative efforts, especially those with several complex interdependencies, could greatly benefit from such approaches. These techniques, unfortunately, are computationally expensive and very time-consuming, not being recommended for massive collaborative development.

2.2.1 Developing a Collaborative Tool

Whenever developing a collaborative tool certain aspects must be well thought-out; the study in [17] refers to numerous issues related with computer supported collaborative tools, that are considered the major causes of why this sort of applications fail to deliver the expected results. Three main motives arise:

- The first concerns the disparity between who does the work and who gets the benefit. The application fails because it requires that some people do additional work, while those people are not the ones perceiving the direct benefit from the use of the application. Thus, it is necessary to ensure that the logic of the application provides balanced benefits to all users.
- The second refers to the problem of intuitive breakdown in decision-making. The design process fails because application developers' intuitions are generally not oriented for multi-user applications. So, tools are usually created based on the potential benefits for people similar to the developers, and fail to see implications of the fact that its use will require extra work of others. Especially if applications are oriented to a heterogeneous crowd, instead of focused on a small and homogeneous group, it might be easy to overlook certain necessities of a given class of users.
- The third, and last, relates to the underestimated difficulty of evaluating collaborative tools. It is difficult to learn from experience because these complex applications evaluation metrics (based on the methodologies of social psychology and anthropology) are often too ambiguous or generalized to provide a meaningful analysis.

This way, whenever developing collaborative tools one must have a good understanding of how groups and organizations function and evolve, in order to provide an application capable of fulfilling every user's requirements.

Knowing that a frictionless environment in development is a property to which most users aspire, several other studies [3, 36] refer to the desirable qualities a new collaborative tool must possess in order to divert from a number of usual points of friction that negatively affect the team's efficiency. Among others they propose functionality that: reduces the *costs of start-up* associated to the assignment of a team to an on-going project, enhances *concurrency* in work collaboration, and maintains a high level of *communication* and *awareness* between elements of a project.

Building collaboration into IDEs

For people to be able to integrate collaboration in their work there are usually three options:

1. The first is through the use of new software applications which already integrate this sort of functionality, this leads to a notorious break of productivity due to the overhead associated with the adaptation to the new software (*context change*).

2. The second is by continuing using the same applications but complementing them with, already existing, communication tools (such as e-mails and instant messengers). This alternative might have less effect over productivity on a first look, however the artefacts end up scattered through multiple applications who might not even support proper means of organization or data interchange (ex: instant messengers).

3. A third solution is to integrate, in an already existing application, functionality for collaborative support. To this we call *contextual collaboration*. This method excels in comparison to the previous two in its reduced friction, embedding collaboration seamlessly into host applications, sparing users from the time and effort of context switch to other tools for purposes of communication or collaboration [6, 21].

2.2.2 Existing Systems

Several systems have been developed in order to provide means of non-collocated collaboration in software development; in this sub-section four relevant systems will be described and compared according to some of the properties covered so far.

GILD Extension

The article [53] introduces a prototype written as an Eclipse plug-in that can be used to help students in computer programming courses to work as a team in programming assignments. By extending an already existing teaching-oriented development environment (Guild)[1] it provides features for code sharing and coordination support. The extension is based on the assumption that version control systems like CVS are too complex for simple projects assigned to a small number of individuals. Also, it states that part of the collaborative aspect of programming is lost in CVS; as developers are not aware of what their team-mates are doing.

The prototype lets developers share their work by providing two operations: *upload* takes a snapshot of one's local changes in his workspace and sends it into a shared repository; *download* obtains the changes from the repository and puts one's workspace in sync with the team. In order to deal with eventual conflicts, this extension still uses CVS to version the files as they are uploaded and downloaded. This way, when conflicts do occur, they are indicated with a warning and developers can manually solve them. Based on an interface including the *diff* facility a user is able to preview the effects of the uploading and downloading operations. However, the lack of a repository activity indicator means that users will have to compensate by explicitly monitoring each other's progress.

[1] Guild: Groupware enabled Integrated Learning and Development. http://gild.cs.uvic.ca

PALANTÍR

Palantír [42] is a system designed to bring workspace awareness to developers. It builds on the top of existing software configuration management (SCM) systems, and concentrates on the collection, distribution and presentation of relevant workspace information. Knowing that workspaces remain isolated and changes made by different developers are not visible until check-in time, *Palantír* provides information about current changes in a software development project; measuring its impact on each user's work. The key feature of this system is to invert information flow from pull to push, freeing developers from constantly having to manually collect new information from a repository, by providing an up-to-date view of the activities in other workspaces.

Palantír is composed by 5 modules: *workspace wrapper*, *event service*, *internal state*, *extractor* and *visualization*:

- *Workspace Wrapper* is responsible for capturing events from the SCM and user's actions, translating them into *Palantír* events, and than sending them to the event service. A workspace wrapper must be specific to a given SCM and thus, this module provides transparency to the system's logic
- *Event Service* is in charge of distributing the events, wrapped by the previously described module, to other users.
- *Internal State* is a module that exists on the client side and maintains an overview of the activities in both local and remote workspaces. And even though it might cache more information than necessary to a developer's current *visualization* style, it is extremely useful as it gives support for multiple styles. Also, internal state is responsible for subscribing only to events that affect the workspace of the client, rather than simply processing every incoming event. This gives *Palantír* higher scalability and prevents a user from being overloaded with information irrelevant to its workspace; consequentially some indirect conflicts might be missed.
- *Extractor works* as a filter that selects a set of events of interest to be visualized, this can be defined based on various developer's preferences such as: the definition of the minimum level of impact of an event or by only monitoring a few selected workspaces.
- *Visualization* is responsible for organizing and displaying the selected activities as they happen in the various workspaces. Some examples of this interface can be seen in [42].

Hence, *Palantír* supports close collaboration among developers by visualizing current changes and showing in real time the impact of those changes on the developer's workspace. The degree of impact can be measured based on three main factors: the number of lines of code modified, analysing the difference between the abstract syntax tree of two versions, and through a dependency graph.

JAZZ

JAZZ [6] is a project developed by IBM which seeks to integrate collaborative capabilities into the Eclipse IDE, enabling small teams of software development to work together more productively. It does so by providing awareness of development processes and artefacts in a way that reduces context switches between tools inside and outside the IDE.

The most visible enhancement to Eclipse is the Jazz Band, this utility allows the insertion and removal of members, defining a team. Within a team numerous functionalities are available, providing multiple levels of awareness. The Jazz Band behaves much like the contacts' bar from a common instant messenger, displaying status information of each member, and like an instant messenger, conversations can be established between two users through a chat window or even voice-over-IP. Also, through the Jazz Band, screen-sharing sessions can be initiated, where one user can see exactly what another user is seeing and doing.

In addition to people-awareness, resource-awareness is also provided via extension of the IDE's Package Explorer. Using coloured icons attached to each file, users can see: in which files their team-mates are working on at a given moment, which files have been locally modified but not committed, and files where changes have been checked-back into the repository.

Another new feature brought by the *JAZZ* plug-in is the existence of markers, which let developers highlight a region of code in the editor. This allows for chats to be anchored to a region of code, and later be reviewed by any team member. Other markers are used to signal that a member has modified a particular area of the code; hovering over a marker shows the difference between the local code and the remote code.

JAZZ focus mainly on reducing the overhead associated with the use of collaborative tools. By focusing on the transparency of every feature, it allows for developers to implicitly provide information of their actions to others while reducing the inherent interruptions and distractions associated with collaboration.

STATE TREEMAP

State Treemap [29] is a widget that provides awareness of divergence between users on a large number of documents. This approach suggests that modifications made by a developer should only be visible to others when that developer validates his modifications, as well as concurrent modifications must only be integrated when the user decides. These suggestions allow users to work un-distracted and in isolation until their work is ready to be propagated, and prevents other users from receiving modification from intermediate and inconsistent states.

Most version control systems, like CVS, allow users to work offline, creating divergence over others versions. To reach a consistent state, when a user finishes a task he commits his version into the repository, merging it with already existing versions. At this stage, however, if new changes committed by others in the mean time have impact on this work, the user might need to re-do his work. *State Treemap* helps users to be aware of the divergence at regular intervals of time, controlling it.

Treemaps [47] help system managers visualize quick changes in large trees of directories and subdirectories, each leaf displays a rectangle proportional to the correspondent file size. The *State Treemap* approach tries to display developers' workspaces as treemaps, where a colour is associated with each file (leaf) showing its current state: up-to-date, locally modified, remotely modified, potential conflict, etc. Hence, through observation of the treemap, a developer can have a general picture of the situation. Assuming that divergence is represented through shades of grey, the user will notice divergence if his treemap starts getting darker. Also, if a user pays close attention he can see where a divergence is located.

However divergence is neither quantifiable nor reliable, as the system only identifies *potential* conflicts.

Also, because Treemap does not handle dependencies between objects, if two people are working on two objects, and these are strongly related, *State Treemap* will not detect a conflict.

System\Feature	*Explicit Communication*	*Consequential Communication*	*Feedthrough*
Guild's Extension	No	No	Yes
Palantír	No	No	Yes
Jazz	Yes	Yes	Yes
State Treemap	No	No	Yes

Table 2.1: Feature comparison among collaborative systems

So far we have observed 4 distinct systems with distinct properties. *JAZZ* uses the concept of teams to limit awareness to a group of potentially interested individuals, while *Palantír* detects users who will be affected by a given set of modifications, based on a dependency graph. Both *Palantír* and *State Treemap* require for a change of context in order to be consulted, *JAZZ* and *Gild's Extension* being developed as plug-ins to an IDE (Eclipse) benefit from the properties of *contextual collaboration*, supporting more sophisticated interruption management schemes and thus reduced friction. Also, *Gild's Extenstion* and *State Treemap* differ from the other two solutions as they require for an explicit check-out from a repository in order to obtain the content of the modifications.

Finally, all provide awareness that allow developers to coordinate their activities, 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.

2.3 Consistency Models

In distributed systems, replication has been the main approach in order to increase availability and performance. However, this brings the problem of consistency and correctness among replicas, as concurrent access to copies on different sites must be synchronized so the order by which operations are received do not produce inconsistent results. The main problem in such systems is, possibly, the partitioning of the underlying network into several components. These components can modify objects locally, even though each node is unable to access directly the remote copies of the data. This way, information from each node might be contradictory, preventing the system from functioning properly.

In order to deal with this issue, two base strategies have been developed depending on either updates are synchronized before (*pessimistic*) or after (*optimistic*) modified objects are used. These strategies represent a trade-off between data availability and frequency with which conflicts occur.

Pessimistic Consistency

Pessimistic strategies strongly limits availability so as to ensure that a committed sequence of operations is never rolled back. This is ensured by relying on a primary replica to synchronously propagate changes to the remaining replicas. By relying on single-copy consistency policy, users of such systems will observe the system as a single, highly available, copy of the data. Such approach will undoubtedly simplify consistency management due to its lock-based implementation while, at the same time, reduce concurrency and parallelism.

Although a pessimist approach may perform fairly well in small and local-area networks, the same cannot be said in a wide-area network, such as the Internet. Due to high latency and unreliable connectivity, such a synchronous approach may lose much of its initial advantages.

Optimistic Consistency

A completely different approach is the optimistic strategy, where applications can tolerate relaxed consistency, leading to the need of less frequent communication which ultimately results in an improvement of performance and availability. By allowing the access and modification of data without *a priori* synchronization (with the assumption that conflicts will occur only rarely) and with updates propagated in background, optimist replication appears to be the solution for wide area, highly scalable systems. A large variety of systems have been created based on this style of replication, including *Bayou*, *DNS* and *Usenet* [40].

A fully optimistic replication system represents a limit of the consistency spectrum, allowing the states of replicas to diverge indefinitely based on the principle that consistency will eventually occur (*eventual consistency*). On the other end of this spectrum resides pessimistic consistency which, by enforcing linearisability, ensures that a sequence of accesses to a certain object among different sites will be executed in the same order in every node and produce an effect equivalent to serial execution in a single site. Certain applications might, however, benefit from a middle-ground between such extremist approaches. We call *bounded divergence* to techniques which allow eventual consistency until a certain degree, blocking access to a replica if specified conditions aren't met. These mechanisms will be explored in greater detail in Section 2.3.3.

When referring design choices to an optimistic replicated system, two main choices can be made regarding where an update can be submitted:

- *Single-master* designates one replica as the master. This way, all updates originate from the master and are then propagated to other replicas, called *slaves*. Even though this solution can be praised for its simplicity, it heavily suppresses availability as well as concurrency.
- *Multi-master* allows updates to be submitted at multiple replicas, independently, and propagates them in the background. This method significantly increases availability while compromising the system's simplicity. It also brings new problems associated, which will be covered in the section below.

2.3.1 Concurrency Control

In a *single-master* system it is usual to have several replicas trying to access the same object, this way concurrency control is needed to solve conflicts between participants, and enhance performance of coupled activities.

Locking In some systems a solution for conflicting operations would be to restrict the user's access to a certain resource. This approach, however, represents the main point of performance loss in replicated applications. To reduce the probability of a lock request being refused, several technique have been developed, for example: *Tickle Locks* [15], used for collaborative application, granting access to locked

resources if the current holder is inactive; and *Multi-granularity Locking*, introducing terminologies like shared and exclusive locks [14, 31]. However, many issues are still unavoidable, for instance: the overhead of requesting and obtaining locks, determining the optimal granularity of the lock (finer granularity might augment concurrency while, at the same time, increase locking overhead), and the decision of when a resource should be locked and release.

Transactions This method is commonly used in concurrency control, although, it does not seem to be the most appropriate for highly concurrent systems due to its demanding response-time requirements. Also, with transactions, users' actions may be lost if a conflict occurs. Finally, if transactions are implemented using locks the problems referred above remain.

Whenever dealing with systems where operations can be applied concurrently over different replicas, sites must support mechanisms to ensure each operation will, eventually, be visible in every single node. Furthermore, one must guarantee that the order by which modifications are applied will not compromise the final state of each replica; otherwise different replicas would have dissimilar views of the final state. In order to fulfil these requirements a series of steps are necessary, namely: defining a criterion for correct order, identifying and resolving eventual conflicts and committing operations.

As the order in which a given node receives operations from other replicas will not always match the execution order, some policies must be considered as to define a valid order. This ordering policy is referred to as *scheduling*, and can be classified into *syntactic* and *semantic*.

Syntactic scheduling determines the total order of operations based solely on information concerning when, where and by whom were operations submitted. Such schedulers try, mainly, to preserve properties such as happen-before relationships among operations [26]. The usage of timestamps associated with each operation as well as vector clocks [34] assigned to each replica, are the most common techniques when dealing with such relationships [23]. Alternatively, semantic scheduling considers, besides causality [28], semantic relationships between operations. Some implementations might go even further as to ignore some causal relations.

Operation Commutativity

Some studies, in particular [38], refer *commutativity* as one of the main techniques to increase concurrency in replicated systems, provided that it reduces the number of rollbacks and redos when a tentative schedule is re-evaluated. Based on the principle that in distributed systems, operations relate with each other in an either causal or a concurrent order, this technique provides a new flexibility degree by allowing concurrent operations to be executed in a different sequence among replicas while at the same time maintaining consistency. In order to identify concurrency, 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 [27]. In [38] the use of commutativity for peer-to-peer replication is proposed, and criticized the operation-log strategy, 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. This can be achieved by assigning a version vector of operations to each object.

An alternative solution is based on the concept of a *Commutative Replicated Data Type* (CRDT) [35].

This 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* [35]. 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.

A solution fitting the same purpose, although focused on a P2P collaborative editing, is *WOOT* [33]. *WOOT*, much like the latter model, handles concurrency based on the context of the operations. However, in this particular system, commutativity is performed through preservation of *intention* [49] of an operation. The *intention* of an operation is the effect observed on the state in which it was generated. Such property is ensured by propagating some contextual information along with the operation, for example: the operation of inserting a character *'1'* in the second position of the string "*abc*" is described as *insert('a'<'1'<'b')*. This way, if a concurrent operation inserts any other character before or after this pair of elements, the intention of the original action is preserved. Essentially, the causality is replaced by preconditions, and as a result, the happened-before relation can be violated in some cases.

Should two concurrent operations perform an insertion between the same pair of objects, an ambiguity must be resolved concerning the correct order by which these two should be applied; otherwise *insert('a'<'1'<'b')* and *insert('a' <'2' <'b')* could result in *"a12bc"* and *"a21bc"* in different replicas. *WOOT* solves this issue by defining a total order based on the id of the replica from where the operation originated. Several optimizations have been done to *WOOT* since it was first presented, like *WOOTO* and *WOOKI* [54].

Yet, as a result of using context information when defining operations, problems might arise when operations act upon objects that no longer exist in a given replica. In such occasions, systems must rely on a record of deleted objects, *tombstones*. With these structures comes the inherent problem of continually growing space overhead, due to the uncertainty about when would be safe to purge *tombstones*. The *LOGOOT* approach [55] suggests an alternative to the use of *tombstones* for massive collaborative editing systems.

Distributed OPerational Transformation (dOPT)

A different philosophy, developed having collaborative editors in view, is *operational transformation (OT)* [11]. This semantic scheduling technique also tries to preserve properties like *intention* and consistency [49]; however, in lieu of relying on a different format of operations (passing contextual information [33]), it transforms operations against concurrent operations that have been executed locally. For example, considering two concurrent operations applied to the string *'abc'*:

- *insert('x', 1)* inserting character 'x' between 'a' and 'b';
- *delete(1, 2)* delete 'c' from the second position.

Depending on the order by which this pair is executed, different results might be attained.

Consequentially, two replicas can respectively diverge into *"axc"* and *"axb"*. In the first replica, operational transformation solves this inconsistency by transforming the *delete* operation based on the already observed insertion; the transformation results in the execution of the operation *delete(1, 3)*, producing a final state *"axb"*. Analogous transformation would be applied to the second replica, in the case of other concurrent operations, in order to preserve *intention*. The detection of causality and concurrent operations is guaranteed with version vectors, thus, each operation piggybacks the version vector of the replica where it originated.

Even though the usage of version vectors might not be an issue in a system with a fixed and low number of replicas, the size of operations will grow considerably as the number of replicas increase. Also, the time efficiency of operations on version vectors will decline with a growing and constantly changing number of replicas. Hence, relying on version vector heavily restrains the scalability OT systems.

Despite the achievements of this first attempt to transform operations, with the purpose of achieving concurrency, numerous problems remained unsolved. Thus, further work was developed covering some of the main issues, for instance [48] and [27]. The latter revolves around divergence associated with *Effects Relation Violation (ERV)*, introducing a new consistency model *CSM*. While applying a transformation between two operations represents an effortless task, a problem arises whenever the transformation is performed over a greater number of operations. In such case, the OT must be converted into a composition of OTs, for example: *OT(O1, O2, O3)* is, in reality, *OT(OT (O1, O2), O3))*. Hence, *OT(O1, O2, O3)* and *OT(O1, O3, O2)* can lead to different results.

Although defining the right transformation order can be usually done based on the position (second argument) of the operations, sometimes there is not enough information to infer the effects of such relations. Therefore, ERV is related to the problem of deciding a correct transformation path, as different paths in an OT could, eventually, lead to divergence among replicas.

Earlier solutions use the *identifier* of the replica, from where the operations originated, to disambiguate the order of transformations to which relations could not be inferred. These, however, are flawed for leading to the detection of false-conflicts (in [27] these phenomena are explained with a greater level of detail). The *CSM* model states that such problems are originated from engaging in a premature *id*-based decision. Alternately, it suggests that in the occurrence of a conflict, the replica should go back to a previous state where the relation between a pair of operations is known; by undoing transformation performed since that state. In order to achieve this, a history buffer is kept in each site, much like GOTO [50], recording all operations executed on that site by its execution order. Hence, ensuring only one transformation path at all sites is no longer a necessary condition for convergence, as in [46, 48].

Ultimately, two main approaches for semantic scheduling were studied: *commutativity* and *operational transformation*. The latter, represents a more matured solution, which is especially effective in systems with a reduced or constant number of nodes. In exchange it might have a greater degree of complexity, and due to constantly having to compare and transform operations, performance might be affected. The former, by not relying on state vectors, greatly improves scalability. In exchange, this strategy requires structure capable of identifying position univocally and, due to being relatively new, is still flawed.

IceCube

A whole new strategy, concerning scheduling of operations, is devised by IceCube [22]. This is an application-independent replication toolkit, hosting an optimistic reconciliation algorithm, which deter-

mines the best scheduling of a set of operations by imposing semantic constraints on operations. A constraint is an object that implies a precondition, and can be defined by various sources: the user of the application, the application or by a data type.

IceCube supports two types of constraints:

- *Static constraints*: relating two operations without a reference to the state of the effected objects. These constraints reflect the intention of the user by imposing execution orders, atomicity or even determining mutually exclusive operations from which the scheduler can choose. Static constraints can be categorized into four types: overlapping (two operations update the same object), commutative, mutually exclusive, and best-order (where one execution order can be more favourable than other).
- *Dynamic constraints*: verifying dependencies (similar to Bayou) or pre-conditions (dependences) and post-conditions (implications) of operations.

As an example, a user might try to reserve Room 1 or 2 (mutually exclusive); in case Room 2 is chosen, rent a projector (implication), which is possible only if sufficient funds are available (dependence). Also, the user might state that if the projector cannot be rent it is useless to rent a projector, thus cancelling the reservation (atomicity).

In order to deal with concurrency, all concurrent operations must be sent to one site for merging, dispatching the merged log to all sites afterwards. Consequently, all sites must be connected during reconciliation and frozen until reconciliation is completed. Unfortunately, a central site represents a serious bottleneck; hence, IceCube might not be the perfect choice when dealing with highly scalable systems. Nevertheless, IceCube is able to ensure convergence and intention preservation, while providing support for multiple applications and data types [45].

2.3.2 Update Definition

Both *operational transformation* and *commutativity* are techniques employed in the scope of *operation-transfer*. This is one of the two main design choices regarding definition of updates: *State-transfer* and *Operation-transfer*. These two approaches define what is propagated as an update whenever changes are performed on a replica.

In the former, used in [23, 37] every modification results in overwriting the entire object, this way maintaining consistency only involves sending the newest replica content to the remaining replicas. Although this method excels in its simplicity, it might become inefficient when dealing with large objects. A practical example would be a distributed file system, where a simple modification applied to a file would require the propagation of the entire file, whatever its size, among the replicas. Also, the fact that two replicas are concurrently modifying the same object might trigger a conflict where one of the updates would have to be discarded, even if modifications are non-related. This way, concurrency can be seriously compromised. Several methods have been devised to synchronize replicas which support *state-transfer*, such as the *Thomas' write rule* [52], *version vectors* [34] and *version histories* [19].

The latter, used in Bayou [51] or IceCube [22], can offer significant gains in bandwidth usage by transferring only the semantic operations performed on an object since a previous synchronization, rather than

the object itself. Operations can be presented in different forms, such as, database statements or deltas between revisions. Bayou, for example, propagates modifications in the form of SQL statements. However, as operations performed on an object are application-specific, *operation-transfer* protocols generally require knowledge of the semantics of the application performing each update. Also, it is necessary to maintain a history of modifications already performed on each replica in order to know which operations to propagate and to whom, and to ensure that all operations are eventually applied.

Summarily, *state-transfer* can be transparently applied in any application by propagating the entire file in every update, thus increasing the cost of each update and boosting the occurrence of conflicts and reducing concurrency. Alternatively, *operation-transfer* provides higher concurrency, interleaving compatible modifications, while requiring more complex and application-specific reconciliation algorithms.

Some systems, like [4], use a *hybrid* solution. Based on a short history of past updates and timestamps, they can decide whether to perform a *state-transfer* or an *operation-transfer*. This way, when updating another replica whose timestamp is recorded in the history, it sends only the necessary operations; otherwise it sends the entire object.

2.3.3 Continuous Consistency Models

Optimistic replication systems seen so far, 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*, is usually achieved by blocking accesses to a replica when certain consistency conditions are not met.

TACT Model

TACT [56] 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, having in view that: a higher discrepancy level also augment the probability of inconsistent accesses. However, if correctly parameterized, this degree might lead to a significant improvement in performance and availability [57]. Thus, the parametrization should be carefully made according to a number of factors, such as: the ratio and rate of reads and writes, the network characteristics, and the frequency of errors.

The latter degree of consistency is based on three metrics, *Numerical Error*, *Order Error*, and *Staleness*:

- *Numerical error*: limits the total weight of writes that can be applied throughout all replicas before being propagated to a given replica. The weight of each write is defined by the application;
- *Order error*: limits the number of *tentative* writes, subject to reordering, allowed by a replica. The order is defined based on the time stamps carried by the operations;

- *Staleness*: places a time limit over the delay of update propagation among replicas. The stale period is determined by comparing a real-time vector, containing the time values of the last received updates per replica, with the current time

This model can be adapted to the particular needs of a given application by allowing defining its semantic consistency, using conits. *"A conit is a physical or logical unit of consistency, defined by the application"*. The level of divergence is measured between conits, based on a three-dimensional vector containing the three metrics previously described. The interesting property concerning TACT comes from the fact that instead of forcing the whole system to a single uniform consistency level, it allows each replica to define its own independent consistency level.

TACT represents a huge step regarding bounded diverge when compared to previous work [1, 25], as these systems manage consistency based solely on a single criterion associated with the semantic of the target-application. TACT, by using multidimensional criteria, is able to express most of the previous systems metrics. Furthermore, this new approach demonstrates a behaviour similar to the one studied in [24] by setting a numerical error bound and assigning a weight to write operations.

Vector-Field Consistency Model

Despite its high adaptability, TACT lacks a notion of locality-awareness. It does not provide spatial relation between neither data objects nor users. Therefore, it will not be the perfect solution for applications in which the places where modifications and reads occur is used as leverage, defining the updates that should be sent immediately or lazily and to whom. Thus, demanding a continuous change in the consistency requirements between replicas. VFC (Vector-Field Consistency) [41] 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 [30], 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*. Knowing that *pivots* change with time, so do the objects' consistency needs during the life of an application. The definition of each *consistency zone* is handled in a fashion similar to TACT, using a three-dimensional vector.

Summarily, VFC combines and extends more elaborate models (like TACT) to bring a more flexible and gradual consistency based on *locality-awareness*. This solution presents a complete set of advantages, either concerning its flexibility, easiness of use, and transparency related to user's perception of the new model. Also, by selecting critical updates and postponing less critical ones, VFC succeeds in reducing the network stress. The algorithm was originally conceived having in view multi-player games, where spatial position of the player in the world would define its consistency requirements concerning other players and objects, and it's now taking its first steps towards collaborative work.

Chapter 3

Solution Design

3.1 Introduction

The main goal of this dissertation is to enrich an existing *Integrated Development Environment* with a new distributed collaborative concept, based on the adaptation of the *VFC* (Vector-Field Consistency) [41] 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. The balance between awareness and intrusion can be achieved by notifying the programmer, as soon as possible, of external changes that might directly affect his work; and gradually postponing or filtering information that is not relevant to the current scope of the programmer's task. By doing so, the programmer can be informed in *near real-time* (NRT) of actions from other users that are conflicting with its work; thus simplifying the resolution of conflicts that would otherwise only be discovered at *checkout-time*. This requires a runtime analysis of the project's structure, as well as of each programmer's task-scope.

As a prove of concept this new functionality will be applied as a plugin to the Eclipse IDE [1] running a Java project. [2]

In this chapter we will start by explaining the approach taken towards the adaptation of the already existing *VFC* algorithm to the new context of distributed collaboration software development; and later the architecture of the solution will be presented in detail.

3.2 Approach

This section concerns the approach taken, in order to fully adapt the already existing version of the *VFC* algorithm (initially designed for multi-player gaming) to a distributed collaboration software development, followed by its instantiation to Java projects.

[1] Eclipse.org: http://www.eclipse.org/platform
[2] Java Programming Language: http://www.oracle.com/java

3.2.1 VFC Adaptation

As described in Section 2.3.3, the *Vector-Field Consistency* algorithm is a *Continuous Consistency Model* that relies on the notion of locality-awareness, thereby using information about the position of every user, associated to a *replicated object*, in order to determine different consistency constraints among the many users of a distributed system. As a user changes its location in the replicated workspace, its distance towards the remaining users is re-evaluated, as are the consistency constraints. The furthest a user location moves from another user's location, the weaker become their consistency constraints.

The *VFC* algorithm is based on several key entities:

- Replicated Object: Element of the replicated universe.
- Pivot: Focus of the user, or a user's observation point. This element is associated to a certain *replicated object* and determines the position of maximum consistency for that given user.
- Consistency Zones: Areas associated to a consistency constraint. These regions are formed around every pivot, and their distance from the pivot is inversely proportional to its degree of consistency.

This algorithm was originally applied to a distributed *ad-hoc* gaming scenario, and will now be adapted to collaborative software development.

In order to easily understand the adaptations that need to be made, let us imagine the scenario of a *FPS*, [3] where a great number of users are playing at the same time; and let's consider the game world as the *replicated universe*. It might be unfeasible to inform every single player of all the actions being performed by all others in real-time. However, it is fairly perceivable that some actions are more relevant than others to a player. For example actions that occur beyond a player's range of sight can hardly be perceived as relevant, while on the other hand, an action that occurs a few meters away from the player might be paramount. Being such the case, to the *Vector-Field Consistency* algorithm, each player can be considered a *replicated object*, the position of each user in the game world can be translated as their observation point (*pivot*), and the *consistency zones* can be increasing radial areas growing away from each *pivot*.

Even at first glance, we can see several adaptations that need to be made. Probably the adaptation with greater impact is based on the fact that the distance (and therefore consistency constraints) 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. It becomes necessary to entirely reformulate the concept of distance in order to realistically express the relation between all the *Java* elements. The new relation between the objects will be given by their *semantic distance*. Consequentially, the *consistency zones* can no longer be defined in terms of uniform radial areas, and must be assigned based on the semantic distance to each *replicated object*.

Providing additional complexity is the fact that, while the gaming world has static boundaries, the structure of *Java* project is at a constant change. Either triggered by the addition of new resources to the project or through the creation and destruction of new elements (classes, methods, ...), the distance between a *pivot* and the remaining objects needs to be constantly recalculated.

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

[3] First Person Shooter

one observation point. For instance, a user might start working on a new class without wanting to lose focus on the previous class he had been working on; even more, a user might want to make an isolated change on a given element without setting it as a new observation point. And finally, after some time, a user might finish his current task and start a new one, being this the case he must be capable of marking previously assigned *pivots* as obsolete before creating new ones.

Moreover, the *VFC* algorithm was primarily developed having in mind *state-transfer updates*, enforcing updates over the states of other *replicated objects* based on their position. However, this is not entirely desired in software development, as it would require the propagation of every document which had changed over time. Instead, we use *operation-transfer updates*, which allow us to propagate solely the operations which have been performed on remote replicas.

One of the main goals of the *VFC* algorithm was to reduce network bandwidth usage. This was achieved by allowing a user to skip state updates from objects far from its observation point. Even though such feat cannot be achieved in a collaborative software development based on operation propagation, where every operation must eventually arrive to every replica in order to ensure consistency, we can emulate the original behaviour by compressing the log of operations which have not yet been sent to a given user. This compression can be performed by detecting pairs of operations which cancel each other, for instance the insertion of a character which was later deleted; thus saving bandwidth. Additionally, we can group operations that are to be propagated in a single operation, thus saving in replicated header information.

In the original *VFC* algorithm, for each consistency zone exists a consistency vector, this aspect remains a constant during the adaptation phase. The consistency vector is a three-dimensional vector which enforces a boundary over how much a given *replicated object* can diverge from its replicas. Each dimension is a numerical scalar defining the maximum divergence of the orthogonal constraints time (θ), sequence (σ), and value (ν), respectively. Where time (θ) specifies the maximum amount of time a *replicated object* can stay in an unsynchronized state; sequence (σ) designates the maximum number of unseen updates; and value (ν) stipulates the maximum difference between *replicated objects* contents. Whenever one of the constraints threshold is surpassed, an update must be performed, thus ensuring consistency.

3.2.2 User Awareness

In previous applications of *VFC*, changes that occurred in other replicas were silently applied to the local replica in a way that was completely transparent to the user. However, in the scope of distributed software collaboration, 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. For instance, if some remote operation were to change the interface class of the class a programmer was working on, in that programmer's perspective, the code would suddenly become erroneous for no apparent reason. 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 and what impact they might have over his work.

Subsequently, we must complement our solution with a mechanism capable of interpreting changes in a particular artefact, and translating them into different forms of user alerts based on their impact. This will allow for changes which are of little significance to be applied silently, and changes which are highly intrusive to be notified accordingly.

3.2.3 Compilable-States

One major issue, that only appears in the context of distributed software development, is related to the notion of *stable versions* of the project. In this scope, a project's state is considered *stable* when the project has no compilation errors. This notion of stability (or inter-object coherence) is completely missing in the original VFC algorithm.

In other forms of *Software Configuration Management* (reviewed in Section 2.1), for example *version control*, the management of compilable versions of a project is each programmer's responsibility. Members contributing to the same project must only commit their set of changes onto a global repository when their local version is compilable; and while doing so, they must guarantee that their local changes do not conflict with changes already applied by other users. Thus ensuring that the version in the repository is compilable at all times.

As we change from version control into *near real-time*, committing specific sets of changes only at user-chosen times no longer becomes a possibility. Hence, 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.

3.3 VFC-IDE 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). The remaining programmers are then able to start working collaboratively by joining this running session. 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. A given programmer can, at the same time, act as server for one or more projects, and the client for multiple other VFC sessions (Figure 3.1).

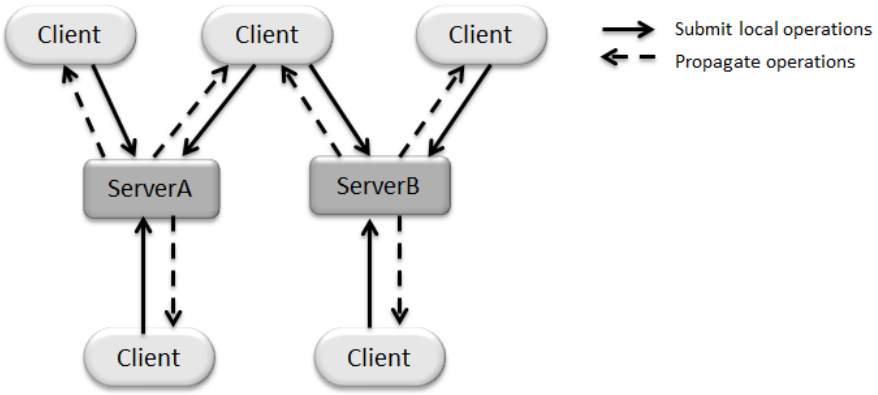

Figure 3.1: Client-Server Architecture

In the next section we describe the several layers that compose our architecture, followed by network details in Section 3.3.2. Then in Section 3.3.3 and 3.3.4 we will refer some architectural details concerning respectively the server and the client. Later, we present the adaptation of document representation which ensures consistency among replicas and preservation of *intent* [49], even with concurrent operations, or operations arriving and applied out of their causal order (Section 3.3.5). Next, the steps taken to allow concurrency in the manipulation of project's files and folders are explained in Section 3.3.6. And finally, in Section 3.3.7, we show how the VFC algorithm enforces consistency.

3.3.1 Architecture Layers

As it will be shown in this section, there are several architectural differences between the client and the server instances. In truth, what we call a VFC Client is, in fact, client with no awareness of the consistency protocol being used by the server. This means that the client can enter a collaborative session with a server running any type of consistency protocol, from *Maximum Consistency* to *Optimistic Consistency* (passing, of course, through *Vector-Field Consistency*), as long as the communication protocol is the same. This provides our solution with a greater level of portability, and will also be extremely useful through out the evaluation phase (Chapter 5).

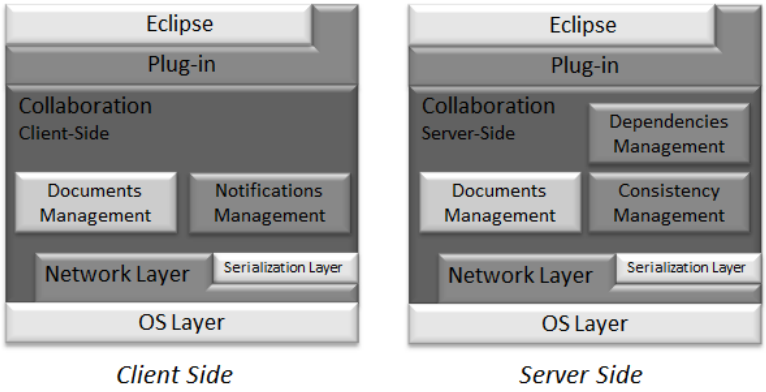

Figure 3.2: VFC-IDE Architecture

Regardless of being a client or a server instance, the *VFC-IDE* architecture is divided in three major layers (Figure 3.2):

- **Network Layer**: Provides services and communication protocols, allows the propagation of notifications and operations between clients and the server-replica of each project. This layer is also in charge of assuring that if the communication between a client and a server temporarily fails, no pending operations (or operations produced in the mean time) are lost; and that they are correctly dispatched as soon as the connection is re-established.
- **Collaboration Layer**: This is the main layer of our architecture, and it is in charge of managing the resources of a project and the representation of a document's structure, based on the *Tree-Doc* algorithm, which allows for text operations to be applied in any order regardless of possible causal dependencies (Section 3.3.5). As a server, this layer is also responsible for managing the consistency constraints among the various replicated artefacts of all the replicas in the same VFC session, keeping track of the observation points of each user, as well as maintaining an up-to-date representation of dependencies between the artefacts that constitute a project (Section 3.3.3). As

a client, it must be able to interpret incoming operations and translating them into various forms of user notifications (Section 3.3.4).

- **Plug-in Layer**: This layer integrates the VFC-IDE functionality in the *Eclipse IDE*. It identifies and retrieves the current observation-point of the developer, wraps local modifications and sends them to the *Collaboration Layer*. The plug-in is also in charge of translating the information received from the *Collaboration Layer*, injecting new content into the workspace, providing hints and notifying the developer about the actions of others. The plug-in layer is the element that allows the user to directly or indirectly interact with a server-replica.

3.3.2 Network

As a result of the total transparency between a client and the consistency protocol in use, all communications between client-replica and server-replica become simplified. Since the server-replica is in charge of receiving external changes and translating their impact over each user's work, the client has no need to exercise any additional computation when performing local changes. Hence, each time a local operation is performed in a replica, that operation can be immediately sent to the server-replica.

The number of different messages exchanged between server and client is fairly small, hence we will present and describe each type of available message traversing the network:

- *SessionRequest*: Can only be produced on the client's side, and it is sent to the server when a client desires to join a running session. The server processes the request and returns a reply in the form of a *SessionReply*;

- *SessionReply*: Produced only on the server's side as a response to a previously received request to join the server's session;

- *EndSessionAction*: Sent by the client when he wishes to leave a previously joined session;

- *EditorAction*: Any operation regarding a change in a given document. These operations carry information about: the location in the document's structure where the change occurred, the content of the change, and the resource (file) affected. Changes to documents are decomposed into insert changes, concerning the insertion of one or more characters; and delete changes, for deletions in a single or multiple characters. This operation message can be sent by the client to the server or vice-versa;

- *ResourceAction*: Is triggered when resources of the project (files and folder) suffer changes, like creating, deleting or moving a file. These operations are always decomposed into create or delete operations of a single resource. For instance moving a folder 'A' each contains a single file 'b', will be translated in a set o resource actions: Create 'A' in the new path, Create 'b' in the new path, Delete 'b' in the old path and Delete 'A' in the old path.

- *NotificationAction*: When a Editor or Resource action arrives at the server and proves to be of significant impact to a given user, this message is dispatched. This type of action only contains the changed resource and its impact on the user work. The user who was the target of this message must then translate this information into a UI-notification.

- *PivotAction*: These actions are only sent by the client to the server, and can either represent the addition or the removal of an existing pivot. Pivot actions are only triggered if the client explicitly declares interest in a given artefact.
- *UpdateAction*: Whenever the server detects that the project has reached a new compilable state, it immediately broadcasts this message, informing every participant in the current session that it is safe to apply pending changes to the document.
- *AckAction*: With the exception of *SessionRequest* and *SessionReply*, this message is always sent as a response to an action of a client or server. Aside from containing the response code of the requested operation, an acknowledgement might occasionally contain a message describing the status of the execution.

In order to reduce stress in the network, multiple *Editor Actions* can be packed into a single action. The same procedure is also valid for *Resource Actions*. Hence, when a consistency constraint is exceeded, instead of sending all pending operations one by one, we can pack them into groups of operations; thus saving having to send replicated information in the header of each operation's message.

Joining a VFC Session

Each time the server receives a request from a client wanting to join the running session, the server opens a new connection strictly dedicated to that user. Then, a series of steps must be taken in order to assure that the new member of the session possesses a replica of the project consistent with the one on the server (see Figure 3.3):

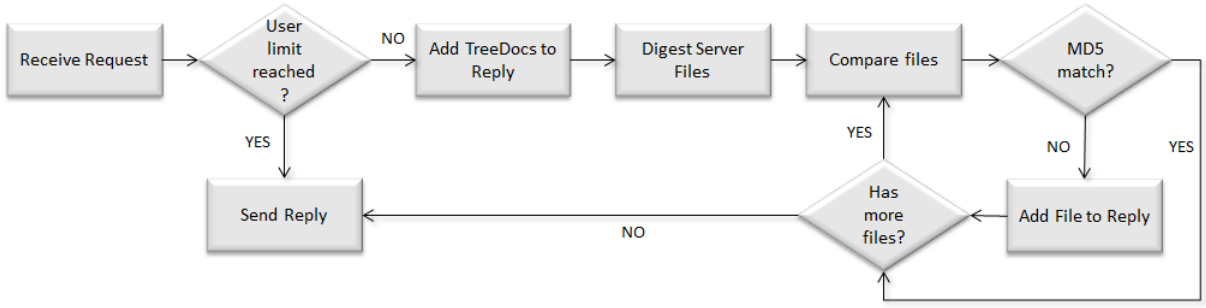

Figure 3.3: Starting a VFC Session (server-side)

1. Together with the request message, the client sends a *digest* (MD5) of each file associated with the project to be shared;
2. Server checks if the user's limit for the project has been exceeded;
3. The server performs a similar task, and compares, for each file, if the *digests* match;
4. For each mismatching file, or if a file does not exist on client's side, the server sends the entire contents of the file to the client;
5. If a given file has already suffered changes on the server side since the beginning of the session, then that file already has a *TreeDoc* structure associated (see Section 3.3.5). In these cases, instead of sending the contents of the file, a *state update* is performed by sending the corresponding *TreeDoc* of the most recent compilable state to the client, regardless of the result of comparing the *digests*.

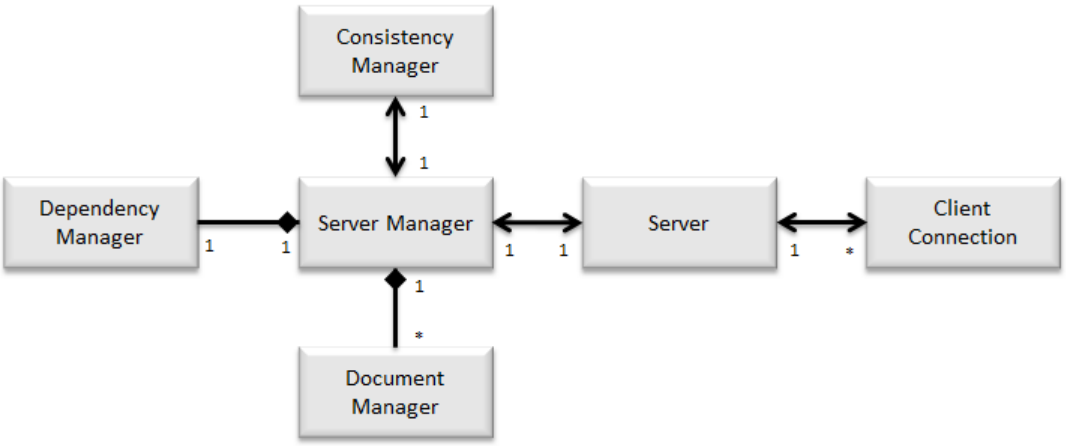

Figure 3.4: Server-Side Architecture

Each dedicated server-client connection has a queue of operations associated. If, for some reason this connection temporarily drops, outgoing operations are placed in the queue are kept on-hold until the connection is re-established.

3.3.3 Server

The server is the 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 artefacts.

The server is always listening for requests to join the current session. And as a new client is accepted into the running session, the server creates a new dedicated client connection and adapts the module in charge of managing consistency to accommodate the new pivots and regions associated with the new member of the project.

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 user-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 *Collaboration Layer* of a VFC server instance, in a distributed project, is composed by the following components (Figure 3.4):

- **Server Manager**: This is the main component of a VFC server instance. It acts as the single point of communication between the *Server* component and all the remaining components, and is in charge of dispatching incoming remote operations to the various components of the server-side, as well as translating server-side events and sending them through the network. The *Server Manager* periodically checks the project's stability, informing the participants of the session if the project becomes error-free. It also handles the entrance and exit of session members.

- **Server**: Creates an abstraction level separating the *Server Manager* from the multiple *Client Connections*. Converts server-side operations into action messages that travel the network (Section 3.3.2), identifies to which *Client Connection* each messages must be directed to.

- **Client Connection**: A new instance of a *client connection* is created each time a new user enters the VFC session. This component is responsible for enforcing the network protocol and, should the client connection ever drop, must periodically attempt to re-establish a connection, making sure that all messages scheduled to be sent in the mean time are not lost.

- **Dependency Manager**: 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 it updates the dependency structure.

- **Consistency Manager**: This is the component 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. Also, it is in this component that all incoming operations are stored, in what we call the *operation log*. This *operation log* will store every operation that arrived the server-side since the beginning of the session. Older operations can be gradually discarded when they have been sent to every user and the manager detects they belong to a state prior to the last compilable state of the project. Finally, every time an operation arrives to the server and is placed on a given user's consistency zone, the consistency manager must verify if, with the arrival of the new operation, any consistency constraints were exceed; consequentially dispatching all the pending operations of that particular zone to the correspondent client-replica. The enforcement of the VFC protocol will be explained more thoroughly in Section 3.3.7.

- **Document Manager**: 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 server's 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 document, and other a snapshot of the document from the last compilable state of the project, detected by the *Server Manager*. In Section 3.3.5, the details of the chosen document structure (*TreeDoc*) are presented.

3.3.4 Client

A client is any user who successfully enters a VFC session. As it was mentioned before, a single client may join multiple VFC sessions, corresponding to different projects.

The *Collaboration Layer* of a VFC client instance is composed by the following components (Figure 3.5):

- **Client Manager**: The Client Manager, much like its server counterpart (*Server Manager*), is the key component to this VFC instance. It receives and interprets incoming messages from the server of a given session, and it also informs the server instance of local changes happening in the replica. There is an instance of a *Client Manager* for each session a user initiates.

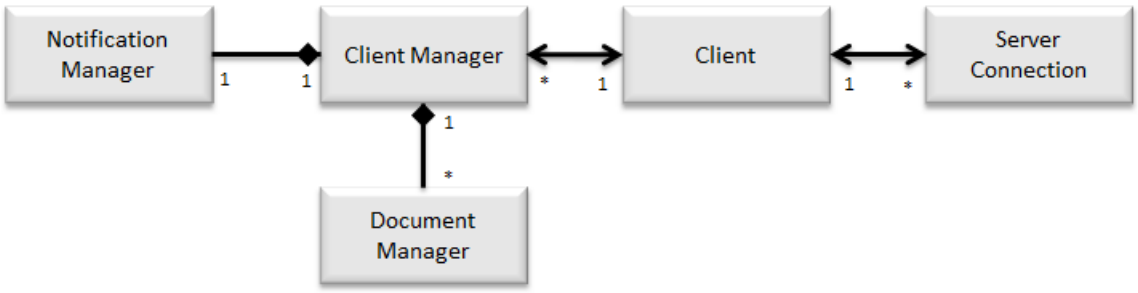

Figure 3.5: Client-Side Architecture

- **Client**: Creates an abstraction level separating each *Client Manager* from the multiple sessions occurring at the same time. For each instance of the *Eclipse IDE* only one instance of this component exists. The *Client* is in charge of receiving operations from the network, converting them into client-side operations and dispatching each one to the correspondent *Client Manager*. Accordingly, when a local operation is to be sent to a server, it is this component's job to map the operation to the correct *Server Connection*.

- **Server Connection**: An instance of a *server connection* exists for each new session the user joins. This component holds the information about the location of a given server in the network, and behaves just like the *Client Connection* of the server-side (see Section 3.3.3).

- **Notification Manager**: Every time the server instance detects that a given remote change might have a significant impact on another user's work, a notification message is sent to both replicas. The *Notification Manager* is the client-side component in charge of translating this message into an 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.

- **Document Manager**: This component works in a way very similar to its server counterpart, and just like on the server-side, each instance of this component has two document versions associated to it. However, in this case, the first version has the current state of the local replicated document, containing only changes performed **locally** or remote changes that were approved by the user; and the second version contains every local or remote change that occurred on the document since the start of the session. As means of ensuring that a programmer can produce and test code without worrying about unstable versions of the project, caused by continuous remote changes, it is up to the user to decide when to add the external changes to its current workspace. The server, however, eases this process by informing the user when the project has reached a stable version. Finally, each time a local change occurs in a given document, the *Document Manager* informs the *Client Manager*, and an operation is immediately sent to the server.

3.3.5 Document Structure

Whenever dealing with situations where operations can occur concurrently over different replicas, every replica must be equipped with means to assure that remote modifications can be applied without compromising the final state of each replica; thus avoiding different replicas to arrive to a state of permanent inconsistency, i.e to permanently diverge and for conflicts to remain unaddressed.

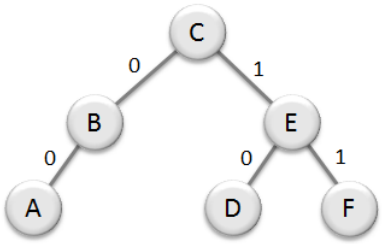

Figure 3.6: TreeDoc of a document containing "*ABCDEF*"

The *VFC-IDE* is no exception and, in order to ensure that external operations applied locally preserve their original *intention* [49] (on client as well as on the server-side), a document structure encapsulating concurrency control over each replicated document had to be created.

As it was studied in Section 2.3, two major approaches emerge: *Operation-Transformation* [50] and *Operation-Commutativity* [33]. *Operation-Transformation* relies on comparing arriving operations with previously applied operations and, if necessary, transforming them thus guaranteeing the original intention is preserved; *Operation-Commutativity* aims at the automatic convergence of replicas without recurring to any means of transformations over arriving operations, this is achieved by representing operations in the form of univocal actions which can be applied by any order.

The first version of our architecture was based on *Operation-Transformation*. However, this approach proved to have an unnecessary high level of complexity, as well significant performance degradation, when compared to the relatively recent approach to operation-commutativity. Hence, it was abandoned in the early stages of the project's development giving way for the ***TreeDoc*** [44] solution.

TreeDoc Structure

The *TreeDoc* is a *Commutative Replicated Data Type* (Section 2.3.1) where all concurrent operations commute; the replicas of a CRDT converge automatically, without having to use any complex means of concurrency control.

TreeDoc gets its name from the structure used to represent a replicated document, where each element in the document is a node univocally assigned to a position in a tree. To these elements content we call *Atoms* and they act as the elements of finer granularity composing a document; in our architecture an *Atom* corresponds to a single character in a document. As each character has a non-ambiguous position, a remote operation now refers to the position in the tree structure and not an absolute position in the document; thus allowing operations to occur concurrently without lost of intention. The total order over the nodes is defined by walking the tree in infix order (see Figure 3.6).

This data type was originally developed for systems where causal order was ensured. However, in this solution we will show that, with some changes to the original algorithm, we can guarantee consistency and preservation of intention, regardless of the order by which operations arrive to a replica; even if causality constraints [28] are violated.

Positioning and Solving Ambiguities

A position of a specific node in the *TreeDoc* can be represented as a path travelled in a binary tree; where travelling to a left child is noted as 0 (zero), and travelling to the right child is 1 (one). This way, a given element can be mapped in the form of a bit-string. For example, in Figure 3.6, the bit-sequence '10' maps the node holding an *Atom* with the character 'D'.

However, as explained by the TreeDoc creators [44], referencing an element solely based on their path in a binary tree might lead to unexpected results. For example when two user simultaneously insert a new character immediately after 'F', position '111' (see Figure 3.7). In these cases the ambiguity is solved, by creating a *major node* containing both nodes, and using the id of the user originating the action as a disambiguator. Total order is then applied to the inserted elements in the same *major node*, assuming that the node of a user with lower id precedes a node of a user with a higher id.

Operation Representation

In our document structure, modifications are classified in two basic operations:

- *Insert Operation*: This action contains the path in the TreeDoc of the node to be inserted (also known as *Docpath*), and the character to be inserted.
- *Delete Operation*: This action contains only the *docpath* to the node to be deleted.

As the TreeDoc ensures that a position is unique to a given node of the tree, when a node is deleted it cannot be simply removed from the structure. Instead, a deleted node is marked as *dead* and, although it does not disappear from the tree, its content is no longer visible in the user's document.

Optimizations can be applied to these two 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.

- *Insert String Operation*: Whenever a user inserts a sequence of characters as a single action, for example when performing a paste or an undo, 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 (see

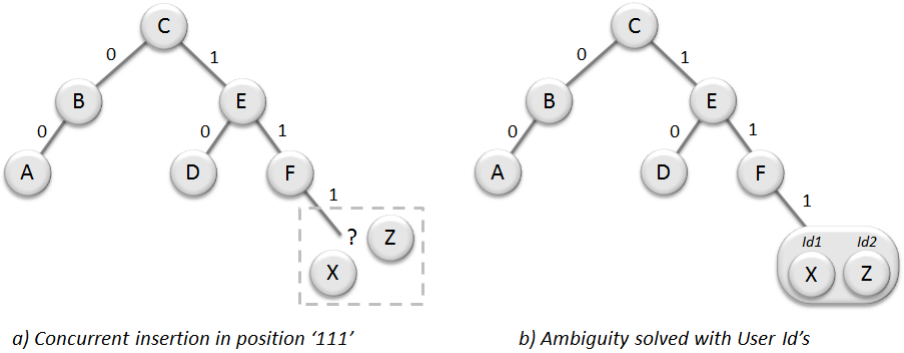

Figure 3.7: Solving ambiguous operations in the TreeDoc

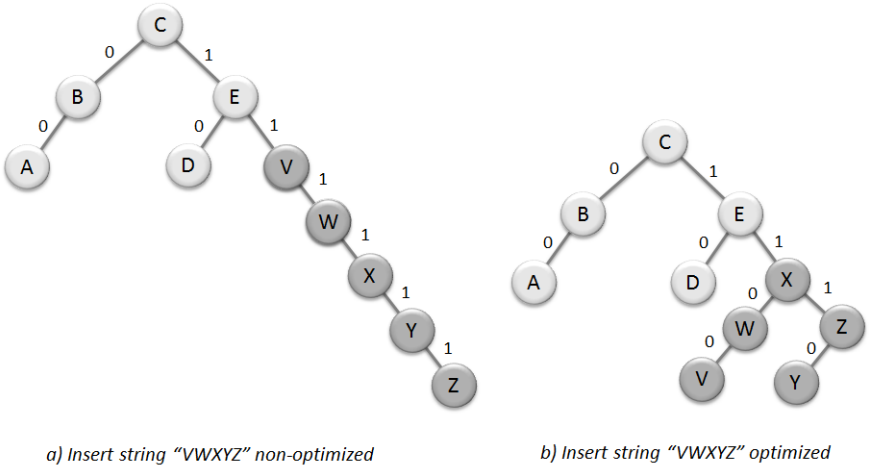

Figure 3.8: Optimization of TreeDoc string insert

Figure 3.8). 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.

- *Delete String Operation*: When deleting a section of text, we may also detect a group of nodes that correspond to a sequence of consecutive right children and pack them into a single delete operation; this operation has a *docpath* pointing to the node with lower depth, and information about the number of right children nodes to be deleted.

Precluding the need for Causality support in Communication

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. So far we implicitly presented two types of node: **Live node**, which corresponds to a character inserted in the document that is visible to the user; and the **Dead node**, which identifies a previously inserted node that has been deleted during the edition of the document. We now introduce the concept of the **Ghost node**.

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 3.9), 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.

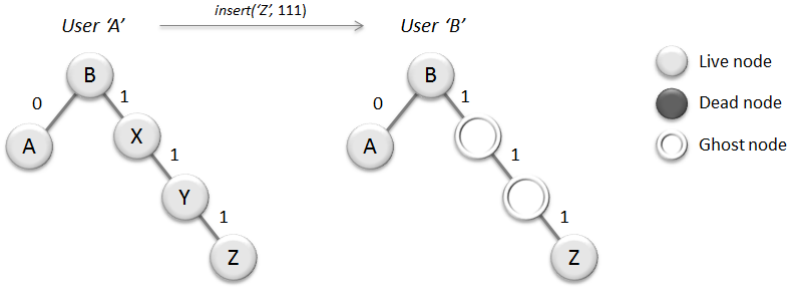

Figure 3.9: Ghost node Example1 - User 'B' receives character Z first

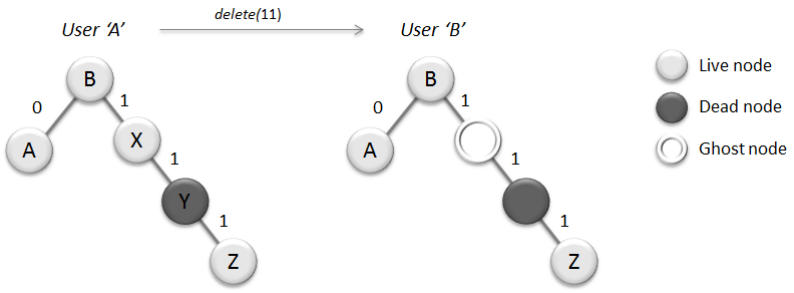

Figure 3.10: Ghost node Example2 - User 'B' receives a delete before the insert

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* (see Figure 3.10). An insert operation over a *dead node* is always ignored.

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 tree structure with become highly populated with nodes that have only a right child, thus significantly increasing the depth of the TreeDoc. This will not only lead to a degradation of performance in insert and delete operations, as it will increase the size of the *docpaths* associated with each operation, and thus the size of messages exchanged among replicas stressing the network.

To solve these problems a couple of operations (*flatten* and *explode* [44]) capable of rebalancing 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.

3.3.6 Resources Structure

An *Eclipse* Project is composed by a hierarchy of *resources*, where these can either be files or folders. So far we have seen how concurrency within a given file can be handled. Nevertheless, issues may also arise with the concurrent manipulation of *resources*.

Should there be no concurrency among resources, the propagation of changes concerning files and folder could be simply performed based on the location of the target resource. However, as multiple users can, at times, change the same resource concurrently, additional mechanisms must be implemented to resolve conflicts and ambiguities. Such conflicts can be presented in multiple forms, namely simultaneously deleting the same file, or concurrently creating a file with the same name in the same project path. While the first case can be simply solved by ignoring *delete* operations over resources that no longer exist, the latter can become more problematic, once the application has no way of knowing which of the resources should prevail.

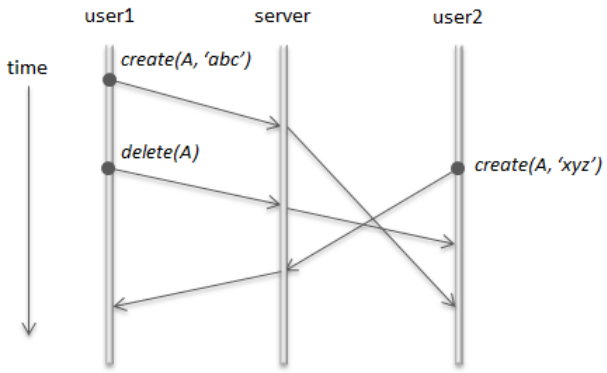

Figure 3.11: Delayed operations causing resource inconsistency

An additional form of concurrency complexity, comes with the possible delay in the arrival of a resource operation to a given replica. In Figure 3.11 we can see an example where, due to delays in the propagation of information, *user1* ends with a file A containing *'xyz'* and *user2* with a file A containing *'abc'*. This situation happens because neither the server nor the users can retrieve any information from the resource operation concerning the actual file to be deleted, only its position in the project.

Consequentially, we need to develop a mechanism capable of distinguishing between different versions of the same resource. Thus assuring that, a remote operation can only be safely applied locally, if the version of the resource the operation refers to is the same as the local resource. This way, and referring back to the example depicted in Figure 3.11, upon receiving a delete not matching its current version of file A, *user2* would not perform the unrelated delete operation. The correct behaviour of *user2* will be explained in the next section.

Version-Vector handling Concurrency

We propose a solution based on the concept of *version-vectors* [13] managed on the client-side.

Each replica manages a map containing the project paths of every resource contained in a project. Associated with each path there is a *version-vector*. The *version-vector* 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 *Resource Action* (Section 3.3.2).

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.

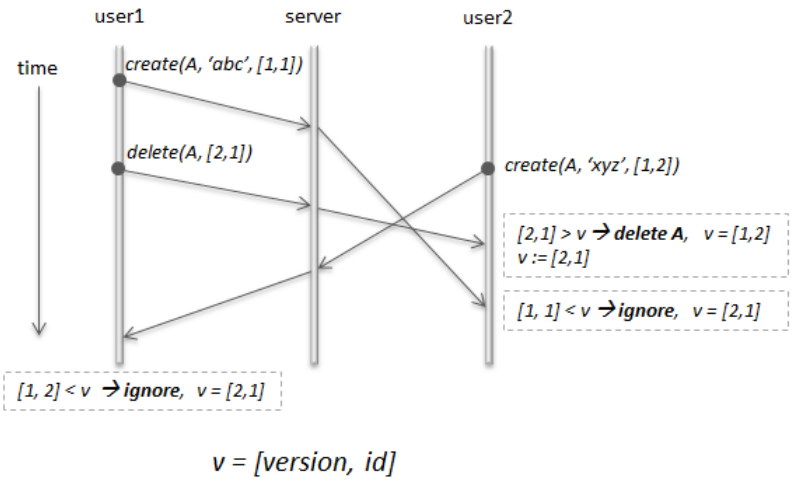

Figure 3.12: Version-vector solving concurrency

- *If no local version exists*, the operation is always applied and the incoming version-vector is added to the map of resource paths.

In Figure 3.12 we can see the new behaviour of a replica receiving concurrent resource operations. In this case, when *user2* receives the delete operation of file A, the versions are compared and the operation is applied; however, when the delayed create operations finally arrive *user1* and *user2*, they are ignored due to their deprecated versions.

Document Operations over Modified Resources

Concurrent resource operations may also lead to a lost of *intention* in remote text operations, as up to this point textual operations had no information regarding the version of the resource they belonged to. This would cause that, if the delivery of a text operation to a replica was delayed for a period of time long enough that would allow for its target resource to be deleted and a new one created in its place, when the text operation finally arrived the replica it would be applied to the new resource. This would lead to different replicas having different views of the same resource version.

To avoid this issue, text operations will also carry a version-vector informing the receiver of the version of the resource to which the operation was originally applied to. In these cases the operation will only be applied if the comparison, between versions of source and destination resources, is a match. If the received operation has a lower version, the operation is ignored; On the other hand, if the received operation refers to a version greater than the one that currently exists in the replica, its is kept on hold until the new version of the resource arrives.

3.3.7 Enforcement of the VFC Model

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.

```
CHECK_VALUE()
 1. value := 0;
 2. foreach op in zone.getOperations() do
 3.     value := value + op.getLength();
 4. if value > zone.maxValue() then
 5.     SEND(zone.getOperations());
```

Figure 3.13: Checking *value* constraint

Multiple programmers working simultaneously in the same project, results in a constant change in the project's dependency structure. 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. This way, it would be computationally very heavy to keep an always up-to-date representation of the dependencies between all the artefacts of a Java project. Instead, 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 recalculation of dependencies.*

VFC Constraints

In our architecture, *consistency zones* act as containers of text operations. Each user has a number of *consistency zones* associated to him, and every time a text operation arrives the server, the *consistency manager* determines in which zone it should be stored.

A *consistency zone* has three types of constraints associated to it, when any of these constraints are violated, all operations stored in that zone are sent to the user. These constraints are:

- ***Time*** (θ): A *consistency zone* has a timer assigned to it, whenever the timer expires all operations in that zone are dispatched. This timer defines the maximum amount of time a user can stay without being informed of the changes occurring in a given zone.
- ***Sequence*** (σ): The sequence constraint acts like a counter of the number of operations that occurred in a particular *consistency zone* since it was last triggered. This constraint ensures that, if a large number of operations arrive to the server in a short period of time, the divergence between the replica of the client and server is never too great.
- ***Value*** (ν): The value constraint represents a limit to how much a client can diverge from the server replica, not simply based on the number of operations but on the intrinsic value and outcome associated to each operation. In our specific architecture, the value of an operation is directly proportional to the length, in characters, of its content (see Figure 3.13).

Incoming Operation

As text operations arrive the server-side, they will eventually be directed to the *Consistency Manager*. As an operation reaches the manager, its impact on the work of every user of a VFC session is determined.

```
INCOMING_OPERATION(USER, OP)
 1. realImpact := NONE
 2. foreach pivot in ConsistencyManager.getPivots(USER) do
 3.     foreach artefact in ConsistencyManager.getArtefacts(OP) do
 4.         impact := pivot.getImpact(artefact);
 5.         realImpact := max(impact, realImpact);
 6. zone := ConsistencyManager.getZone(realImpact);
 7. INSERT_OP(USER, zone, OP);
```

Figure 3.14: Measuring impact of an operation

```
INSERT_OP(ZONE, OP)
 1. region.add(OP);
 2. CHECK_SEQUENCE();
 3. CHECK_VALUE();
```

Figure 3.15: Adding a new operation to a consistency zone

As a single operation may affect more than one project artefact, the impact of an operation is determined by the impact of changing the artefact with the higher level of dependency over the user's pivot. Also, if a user has more than one pivot declared, the impact of the operation is measured for each pivot; and the higher level of impact among the various pivots is considered to be the real impact over the user's work (see Figure 3.14).

After the level of impact has been determined for a given user, the *Consistency Manager* translates the measure of impact into a specific consistency zone, and assigns the received operation to it. The insertion of an operation in a consistency zone is followed by a series of check operations, to determine if the new operation causes the threshold of any of the zone constraints to be surpassed (see Figure 3.15).

Changes in Pivots

A user is able to change its pivots in two different ways: explicit or implicitly. The first is achieved by manually adding a new pivot to the project, or removing a previously added pivot; the second happens whenever a user starts performing changes in a project artefact that does not match his previous point of edition. For each user, there exists only one implicit pivot, and when the user changes his point of edition, the implicit pivot is replaced by the new one.

Regardless of what might have caused it, a change in one of the user's pivots demands a re-adjustment of all the consistency zones of that user. This might cause that unsent operations that have been assigned to a consistency zone in the past, might now belong to a new consistency zone. Hence, aside from rearranging the consistency zones of a user, we must also recalculate the impact of all pending operations (see Figure 3.16).

```
NEW_PIVOT(USER)
 1. foreach zone in ConsistencyManager.getZones(USER) do
 2.     foreach op in zone.getOperations() do
 3.         zone.remove(op);
 4.         INCOMING_OPERATION(USER, op);
```

Figure 3.16: Repositioning pending operations after a pivot change

Chapter 4

Solution Implementation

This chapter covers the fundamental implementation details of our solution. We will start by explaining the adaptation of the already existing *VFC* algorithm to the new context of distributed collaboration software development (Section 4.1), followed by the structure of VFC server, along we some of its particularities (Section 4.2); and later we will describe the client's implementation, referring some of the most interesting aspects, and the features developed to provide awareness over external changes (Section 4.3).

Our solution was originally developed in the *Galileo* release of Eclipse, and was latter tested (with no need for additional adaptations) in the *Helios* and *Indigo*[1] releases.

4.1 Eclipse Adaptation

Having identified the key aspects to be adapted (Session 3.2.1), the next step is to apply them to the context of a specific *Integrated Development Environment* (IDE). Having in mind factors like, portability, extensibility and community activity (and having preference for an open-source tool), the IDE of choice was *Eclipse*.

In this section we describe the steps taken towards the adaptation of the *Eclipse IDE* to support distributed collaboration based on *VFC*.

4.1.1 Eclipse IDE

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE). It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins, other programming languages including C, C++ and Perl. For purposes of this document we will rely solely on its core language: *Java*.

One of our major concerns was to be able to complement the IDE with a collaborative feature in a way that would not involve changing the core of the application. Luckily in Eclipse this can be easily ensured, as it provides an extensible plug-in system, allowing a developed plug-in to virtually run on any Eclipse IDE, regardless of its version or the Operative System behind it.

[1] Eclipse - http://www.eclipse.org/

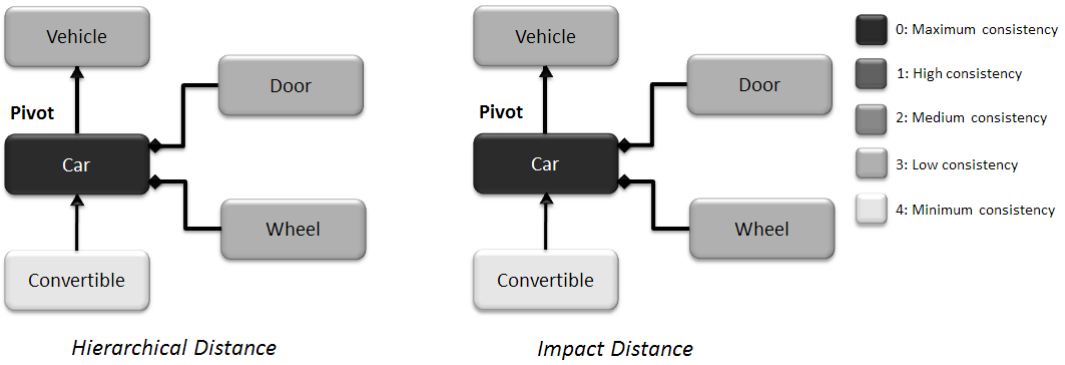

Figure 4.1: Vehicle project structure - Hierarchical distance matching Impact distance in a scenario with low granularity

4.1.2 Eclipse Java Project Structure

The structure of an Eclipse Java Project is fairly similar to the native Java project's structure, where programming logic and behaviour are encapsulated in objects designated as *Classes* which in turn contain fields and attributes (referencing other objects) and subroutines, designated as *methods*. These *Classes* support an inter-class relationship which can be either compositional or hierarchical. Sets of *Classes* are organized into namespaces through means of *Packages*. All the components of the Java language from package to method, and every element that might compose a method, are referred by Eclipse as *Java Elements*. Also, the Eclipse Project Structure augments this scope by including the concept of *Resources*. A *Resource* can represent either a folder or file, which can be any type of system file (xml, text document, an image, a library...).

4.1.3 Positioning and Dependencies in a Java Project

When a programmer is working on a Class, the Eclipse Editor has no perception of the specific Java element which the programmer is currently editing, the only information collected is related to the position in the document of each document change. Hence, for each inserted character we must identify the Java element corresponding to that position in that particular document; the analogous operation is performed in the case of the deletion of characters, with the particularity that in the event of a block delete (the deletion of more than one character in the same operation) we must determine all the affected elements within the deleted region. By accessing the Eclipse core compilation structure we can obtained the Java elements associated to each document position.

Every time a change occurs in a given Java element, the distance between this element and the element associated to a user's pivot must be calculated, and then translated into a *Consistency Zone*. The distance between a pair of Java elements must be measured based, not on the hierarchical class structure of a project, but on the *impact* that a change on a particular element might have on the other. Should the Java elements granularity be set solely to classes, the difference between hierarchical distance and impact would certainly be thin. As we can observe from Figure 4.1, depicting a very simplified version of the structure of a vehicle, the user with a pivot on the class *Car* demands its super class (and the types that compose its class) to be placed in a high consistency zone, and its subclass in a very low consistency zone, given that changes on the subclass *Convertible* have no impact on *Car*. In this case, as the hierarchical class relation grows stronger, either by inheritance or composition, so does the impact of

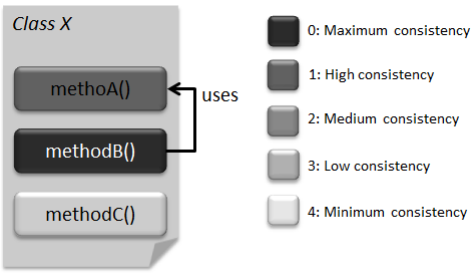

Figure 4.2: Dependencies within a Class (pivot in *methodB*)

changes on those elements.

The case completely changes as we increase granularity, in order to add methods into the equation. With this new level of granularity, the degree of required consistency within a given class is no longer homogeneous, and might vary from method to method. Therefore the hierarchical distance no longer applies, as it cannot infer relationships between Java elements inside a class. For instance, in Figure 4.2 we have an example of a class with three declared methods inside (A, B, and C) and somewhere inside the execution flow of *methodB* there is an invocation to *methodA*. In this case the user with a pivot set on *methodB* will likely be more interested in changes that occur within *methodA* than changes in any other part of the class, mainly because a change in *methodA*'s signature will have a critical impact on the correct behaviour of *methodB*.

A similar example but with increasing degree of complexity is exemplified in Figure 4.3. Where simply using hierarchical distance would result in identifying two completely disassociated classes, where the highest consistency zone would be surrounding every element within *ClassX*, while *ClassY* would be placed in the lowest consistency zone. However when we introduce a finer granularity we can see that these two seemingly unrelated classes are affiliated by a reference from *methodB* (the pivot) to *methodJ*; assigning to *methodJ* a considerable level of consistency.

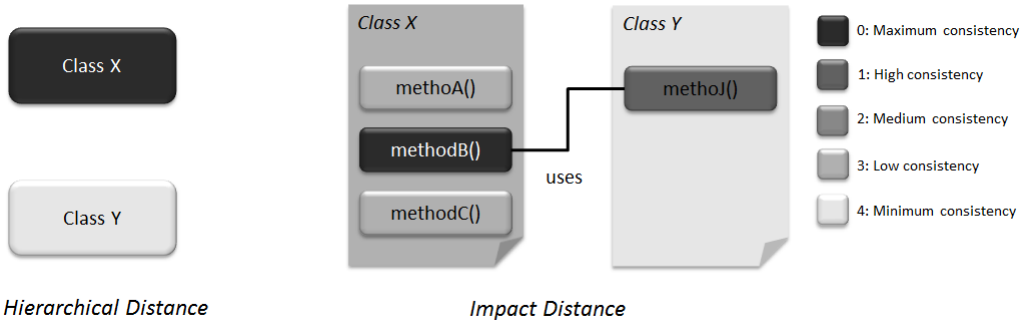

Figure 4.3: Dependencies between Classes (pivot in *methodB*) - Comparison between Hierarchical Distance and Impact Distance

On the other hand, observing the project's structure with a lower granularity can also help us determine relationships among elements that would have been otherwise overlooked. Introducing *Packages* to the calculus of impact allow us to increase the degree of consistency over elements that, while not directly affecting the focus of the user's work, might be useful in order to ensure an overall consistency of the module the user is working on.

Hence, to pragmatically calculate the dependencies among *Java elements*, in the context of collaborative software development, we need to parse the project structure into four elements, which we will

Pivot \ Changed Artefact	Method	Class	Package	File
Method	**1) Is the same method;** **2) Is a used method;** **Else:** Get *pivot's* class and treat as class/method relation;	**1) Is a Used class;** **Else:** Get pivot's class and treat as a class/class relation;	Get *pivot's* package and treat as a package/package relation;	No relation
Class	Get *artefact's* class and treat as a class/class relation;	**1) Is the same class;** **2) Is an Inner class** **2) Is an Interface;** **3) Is a Superclass;** **4) Is a Used class;** **5) Is a Subclass;** **Else:** Get *pivot's* package na threat as a package/class relation;	Get *pivot's* package and treat as a package/package relation;	No relation
Package	Get *artefact's* package and treat as a package/ package relation;	Get *artefact's* package and treat as a package/package relation;	**1) Is the same package;**	No relation
File	No relation	No relation	No relation	No realtion

Figure 4.4: Map of dependencies between a pivot and a changed artefact

from now on refer to as *artefacts*. *Artefacts* are: *Packages*, *Classes*, *Methods* and *Files*, where the latter represents non-java files. Further decomposition of the project's structure would have been possible but, as we will observe, the various available relationships between these four elements already provide more than enough complexity to determine a realistic notion of impact.

As the reader may observe from 4.4, at this stage our solution does not yet support dependencies involving non-Java elements. This, however, falls beyond the scope of this dissertation and, as such, all non-Java elements will have associated the lowest consistency level available.

Due to the high level of inter-artefact relationships, the use of a tree-based structure to represent all available dependencies no longer suffices. It becomes necessary to create a graph of dependencies able to, for each *artefact* type, indicate the multiple possible associations with every other artefact:

- *Package*: Must be able to reference all of classes contained in that given package;
- *Class*: Must be able reference all the classes that compose it, its superclass, all its subclasses, its innerclasses and implemented interfaces; along with all the methods declared inside.
- *Method*: Must be able to reference all the class types used, all methods used, and the owner class.

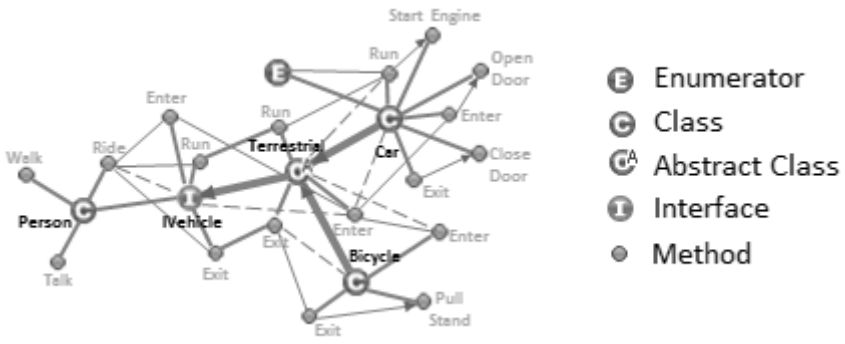

Figure 4.5: Example of a dependency graph between a class *Person* and a module *Vehicle*

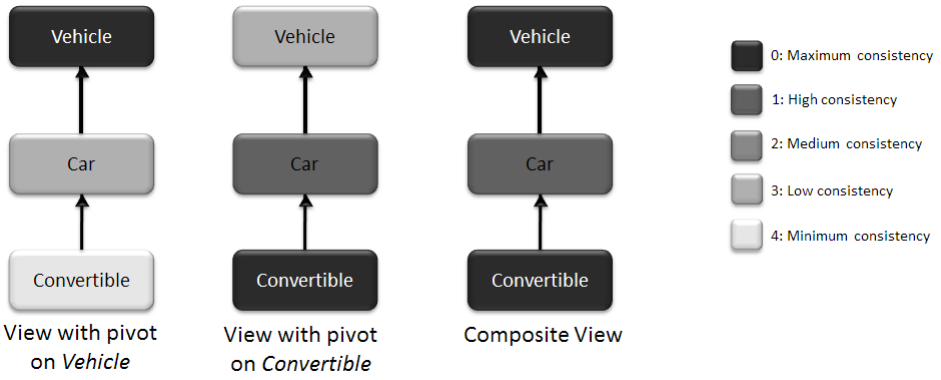

Figure 4.6: User with multiple pivots

In Figure 4.5 we illustrate how this dependency graph would look like with a graphical representation. In this case we have the example of a very small project containing a class *Person* which can interact with two types of vehicles: *Car* and *Bicycle*, where the thickness of each line represents the strength of the dependency. In the example not all dependencies are depicted so that it can become more comprehensible.

The measure of impact gains yet another level of complexity, as we allow for programmers to specify multiple zones of interest. Allowing multiple *pivots* will not only create additional consistency zones centered around each pivot, but may also lead to overlapping consistency zones; as the same artefact may be, at the same time, the superclass of a *pivot'ed* class, and the subclass of another. In these cases, every time an artefact is associated with more than one consistency zone for the same user, it will behave as if it was solely associated with the highest of the consistency zones (see Figure 4.6).

Even though the concept of consistency zones associated with each artefact persists, in practice, some adaptations are vital. In the scope of Java project's, artefacts are constantly shifting between consistency zones, either triggered by frequent changes to the projects structure or through the addition and removal of pivots. Consequentially, it is no longer practical the notion of a consistency zone associated to a given artefact. Instead, consistency zones now act as "containers" of changes. For each change that occurs within an artefact, the impact of the change is measured based on its relation with a user's pivot for the current state of the project's structure. The impact of these changes are then mapped into consistency zones based on several impact thresholds. Thus, each time a consistency constraint is exceeded, all operations accumulated in the zone, up to that point, are propagated.

The available consistency zones present themselves as following:

- *Consistency Zone 0*: This is the higher level of consistency, assigned to changes which will most definitively affect the programmer's task and might even conflict with it. For example, changes within the same method the programmer is working on, or changes to the same line;
- *Consistency Zone 1*: This level of consistency is reserved for changes that might have a critical impact on the programmer's work. For example, changing the interface of the class in which the programmer has a focus on.
- *Consistency Zone 2*: Concerns changes that, while not critical, can directly or indirectly affect the programmer's current task. For instance, changes within the same class or the superclass of the programmers focus.

- *Consistency Zone 3*: Changes less likely to affect the programmer's work, but that are hierarchically close, go into this category. For example, changes within a subclass or a class from the same package.

- *Consistency Zone 4*: This is the lowest degree of consistency, and is assigned to changes that are completely irrelevant to the programmer's task. For example, changes in a non-Java file.

4.1.4 Eclipse Compilable States

As seen in Section 3.2.3, in order to ensure that a programmer always has it own compilable version of the project, while continuing to receive operations from all participants, we must not apply incoming remote operations directly to the programmer's workspace. Instead, external changes must be kept on hold until it is certain that they will lead the project to a compilable state.

At the same time it will also be an interesting feature enabling the programmer to, at any time, visualize pending changes which have not yet been applied. This would enrich the programmer's level of awareness over specific changes, as well as it would ease the resolution of eventual conflicts. Further details about compilable state detection and behaviour are presented in Section 4.3.

4.2 Server

The server implementation follows the same model described in Section 3.3.3, only the original *Server* and *Client Connection* components were joined into a single module. Also, in order to detach the *Server Manager* component from any dependency with the Eclipse framework, we created an additional module (*Resources Module*) to handle concurrency among Eclipse Resources.

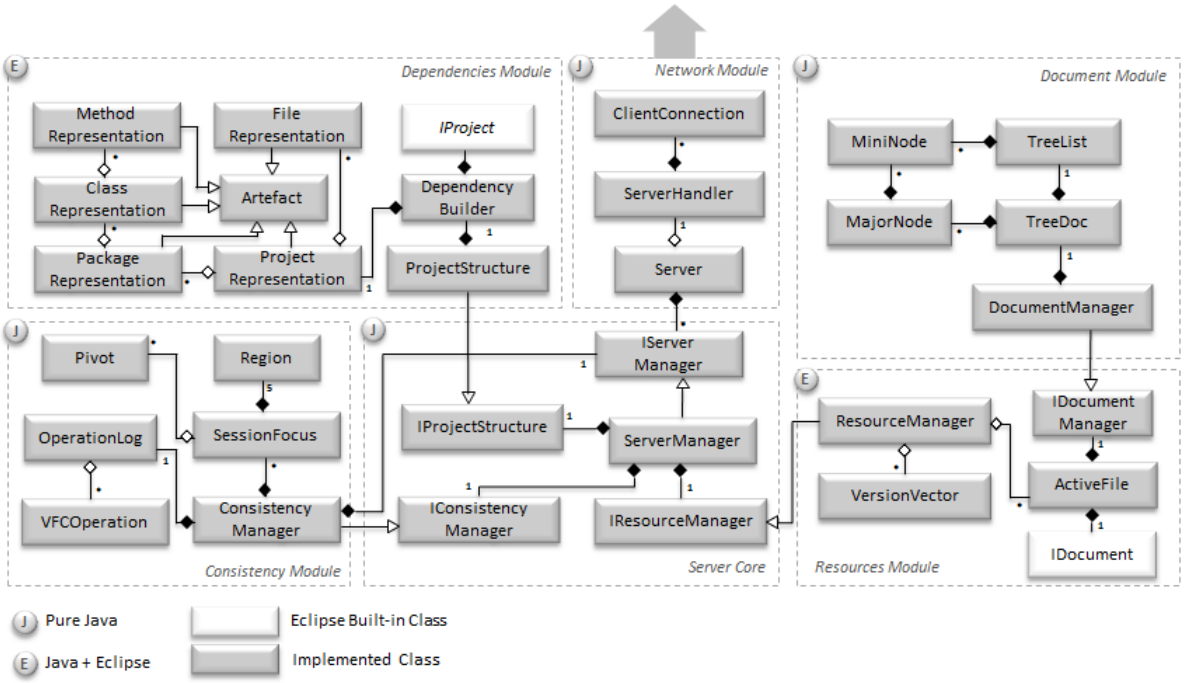

Figure 4.7: VFC Server Class Diagram

The result, as we can see from Figure 4.7, is four modules written in pure Java, and only two modules dependent of some specificities of the Eclipse framework:

- *Dependencies Module* is in charge of managing the multiple dependencies among the artefacts of a project and, as such, demands an inspection of the Eclipse project structure; it is also this module that, periodically called by the *Server Core*, checks if the project has reached a compilable state.
- *Resources Module* is responsible for managing Eclipse Resources, and it is the module that forms a bridge, connecting the document structure of our architecture (*TreeDoc*) with the document associated with the Eclipse Editor.

Each module of the server is completely transparent to the implementation of the remaining modules, as all interactions between modules are based on interfaces. This provides the solution with a higher level of adaptability and customization, where an entire module can be replaced without affecting the rest of the solution (for as long as the interface of the module is abided to). The instantiation of each of these modules is only performed at runtime, based on a configuration file; this allows for module switches to be performed without the need to recompile the entire project.

4.2.1 Queueing of Asynchronous Operations

As the *Server Core* module works as the single point of communication between modules, it is easy to be identified as point of bottleneck in the architecture's performance; where modules running in different threads would become locked due to concurrent accesses to the Core module. However, communication in the direction of the *Server Core* only occurs on two distinct occasions: when the *Consistency Module* propagates pending operations from a zone that exceeds a consistency constraint, and when the *Network Module* receives operations from a client and sends them to the Server Manager. In any of this two events, the invocations can be easily converted into asynchronous calls. Hence, to increase the performance of the server architecture, an *InputOperationQueue* was created. This way, when requests arrive the *Server Core* they are queued and treated in a FIFO order, thus allowing for the remaining modules to continue their work (Figure 4.8).

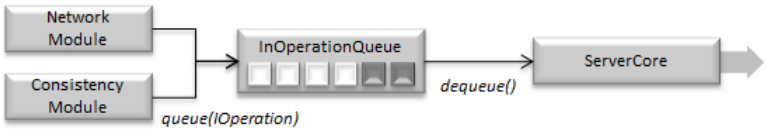

Figure 4.8: Server InputOperationQueue

4.2.2 Lazy-Loaded Documents

Depending on its nature, in time a Java project might grow to have a great amount of Java files associated to it. This way, it would be a very heavy procedure to create a *TreeDoc* structure for every single file at the beginning of a VFC session; also, having these structures running at the same time would demand high memory requirements. This problem affects the server as much as any client joining a session. In our solution, we developed a *Lazy-Load* approach where all documents start with no structure associated. As text operations arrive, if the targeted document does not yet have a correspondent *TreeDoc*, it is immediately created.

4.2.3 Customization

To provide the final user with a higher level of customization over our solution, several properties have been made customizable through means of a configuration file. The most important two being:

- Consistency protocol: which allows to change between *Vector-Field Consistency* and *Maximum Consistency*, where the latter causes changes received by the server to be immediately propagated to all the clients. This property will prove extremely useful in the evaluation chapter (see chapter 5).
- Vector-Field Configuration: which allows the consistency constraints of each consistency zone to be fully customized. Thus allowing the user to tune its server instance to fit the particularities of each project.

Several instances of the server can exist at the same time with different consistency configurations.

4.3 Client

In this section we will start by describing some of the more relevant aspects of the client's architecture, and then we will present some characteristics of the new interface provided by the *VFC-IDE Plugin*.

The client implementation follows a structure similar to the one described in the architecture chapter (see Section 3.3.4) where, much like its server counterpart, the few differences have in view a higher level of portability and detachment from an Eclipse-dependent implementation. The *Client* and the *Server Connection* components where unified in a single module and, to isolate the *Core* module from the Eclipse framework, an additional *Resources Module* was created.

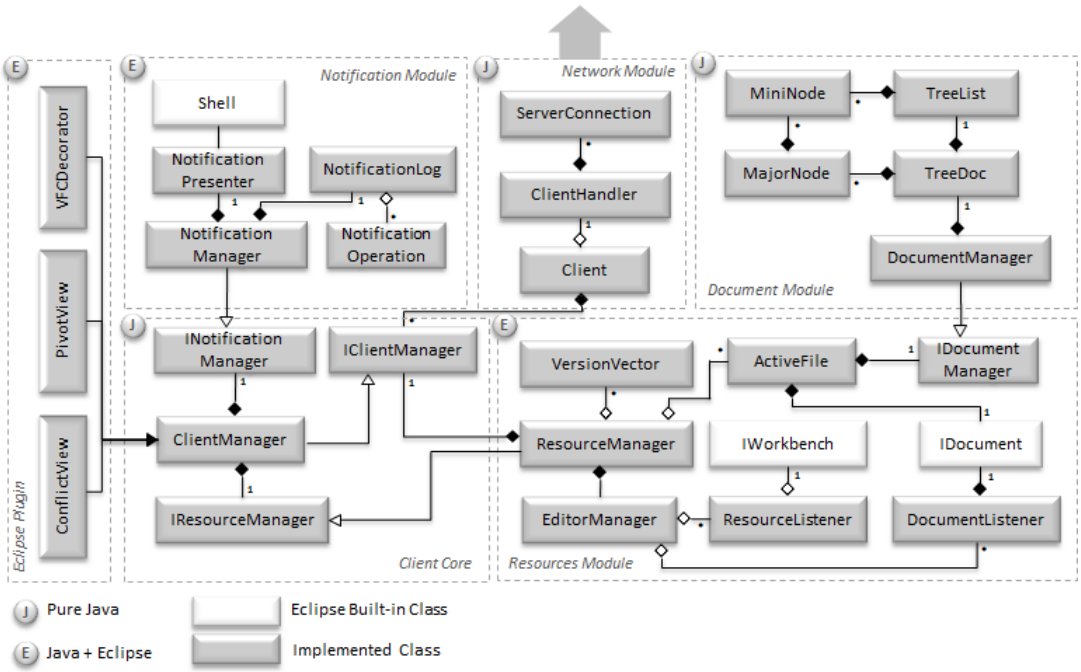

Figure 4.9: VFC Client Class Diagram

In Figure 4.9 we can see the five modules comprising the client-side implementation, and an additional module, *Eclipse Plugin*, which is responsible for all available user interfaces in our solution. The client-side implementation introduces a new module: *Notification Module*, and an extension of the *Resources Module* which already existed on the server-side.

4.3.1 Intercepting Changes in the Project

The Eclipse framework already provides several mechanisms to detect changes in the various resources. Such mechanisms are able to capture these change events before and after they are applied, and were crucial in the development of our solution.

In the *Resources Module* we can notice the existence of class *DocumentListener*, this is a class that extends a listener from the Eclipse framework which, at the beginning of each VFC session, must be associated with every document of the project. Through this listener where are able to be informed of local changes to a particular file, and accordingly update the *TreeDoc* structure and notify the *Client Manager* (see Figure 4.10). Also, whenever a user tries to close an editor with unsaved changes, the normal behaviour of the Eclipse framework would be to ask the user if he wanted to save the pending changes. However, having a user close an editor and not saving changes (that were already sent to the server), would cause inconsistencies not only between the *TreeDoc* and its associated document, but also between this replica and the server. To avoid this situation, we now intersect *editor close* events belonging to a VFC session, and always save the pending changes.

As important as changes within a document, are changes that create or destroy resources. The class *ResourceListener* implements the Eclipse's *IResourceChangeListener*, and can be associated with the Eclipse workbench to listen for changes affecting resources. Whenever a resource change occur, the *ResourceManager* immediately informs the *ClientManager*, who in turn will dispatch a resource operation to the *NetworkModule*.

In the Eclipse framework there are three distinct ways of changing a resource: create, delete and move; As for our solution, we only work with create and delete operations, and treat the *move* event as a composition of the first two. This choice is based on the premise that moving Java files, will sometimes lead to changes in the document's content that the document listener is not able to capture. For example, moving a Java file to a different folder causes the Eclipse framework to automatically update the package of the class inside that file. As such operations do not trigger the *DocumentListener*, the document's

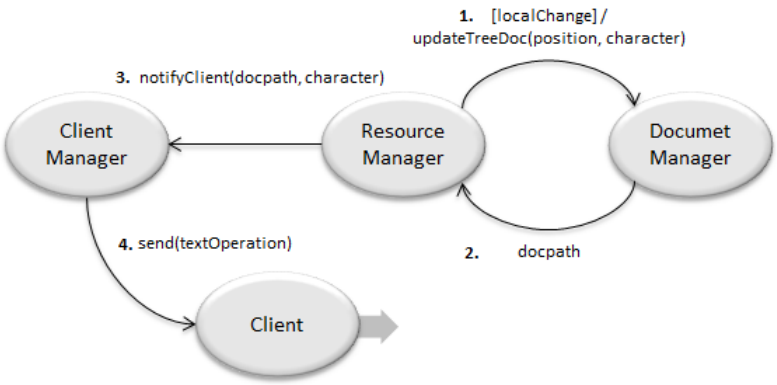

Figure 4.10: Reacting to a local change

TreeDoc would diverge from the actual contents of the file.

Finally, a resource listener is triggered by changes in the project structure, and these changes do not always refer to a single resource change. For instance, the act of moving a folder 'A' with two files 'Y' and 'Z', triggers a resource event which contains: move 'A', move 'Y' and move 'Z'. In this situations we, do not only decompose the *move* operations, as we reorder the list of atomic events and pack them in a single resource operation.

The order of the atomic events is:

1. Creation of Folders (multiple folder operations are ordered with ascending length of their path);
2. Creation of Files (in the new path);
3. Deletion of Files (from the old path);
4. Deletion of Folders (multiple folder operations are ordered with descending length of their path);

Note that conflicts due to concurrent operations are solved via the use of *version-vectors*, as seen in Section 3.3.6.

4.3.2 Joining a VFC Session

The Eclipse framework, and in particular the *Plug-in Developers* version, provides several extensions to the Eclipse interface, these extensions can go from the addition of new options in context menus, to the creation of new views and perspectives to the Eclipse IDE. Throughout the development of our solution several extensions to the interface were created, in order to ease the usage of the VFC-IDE Plug-in.

The most relevant is probably the addition of a VFC menu, which allows for a user to make a VFC session out of any project inhabiting its workspace (see Figure 4.11). This same menu, can also allow for a user to join an already-running VFC session at any given time.

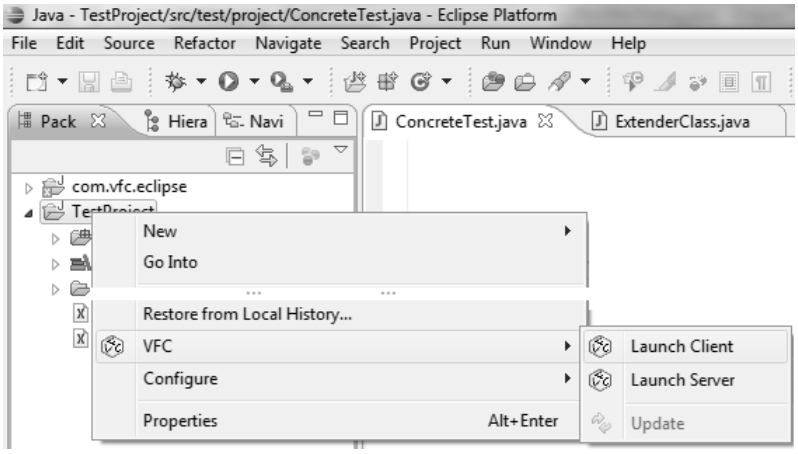

Figure 4.11: Starting a VFC session

4.3.3 Real-Time vs User-Choice

As multiple programmers are editing the same project, a single user's replica of the project is constantly being updated with external changes. Despite of how desirable it might be to have an up-to-date view of all the changes occurring in the project, such feature would probably prevent the user from ever having a compilable version of the project; thus reducing his chances of being able to successfully compile and test his own changes.

To surpass this issue, we provide two distinct modes of edition:

- The first is the *Real-time* mode, in which external changes, arriving the replica, are immediately applied to the user's workspace.
- To the second mode we call *User-choice*, and it will allow the user to work with some degree of isolation. In this case, external changes arriving the client-side are kept in a temporary version of the correspondent file, thus not affecting the compilable state of the local project. These pending changes can latter be applied to the workspace when the user has confirmation that they will lead to a compilable state. This feature is achieved by assigning to each file being edited two *TreeDocs*, one containing only local changes performed by the user, and the other containing local changes and all changes received from the server.

Of course that if, in *User-Choice* mode, a user cannot directly see external changes until they are compilable, there is little or no gain in relation to other *Software Configuration Management* techniques such as *version control* (see Section 2.1). Hence, we developed two additional features that enable the user to know what changes are being made in the project, while not directly affecting his work.

The first was the creation of *decorators*, that vary based on the status of each specific file. As it can be seen from Figure 4.12, whenever a local file receives external changes that are not immediately applied, the icon of the correspondent file in the project tree is decorated with a star in its right upper corner. The same mechanism is used to notify the user of a file with conflicts (appearing as a red circle), and to inform that the file has a new compilable version (appearing as a yellow circle, with a check sign in the middle).

The second feature, allows for the user to compare its local version of a file with the version that contains changes from the server, that were not yet accepted. To achieve this, we added a new menu option; whenever a user right-clicks a file or an editor window that has pending changes, a "VFC" menu

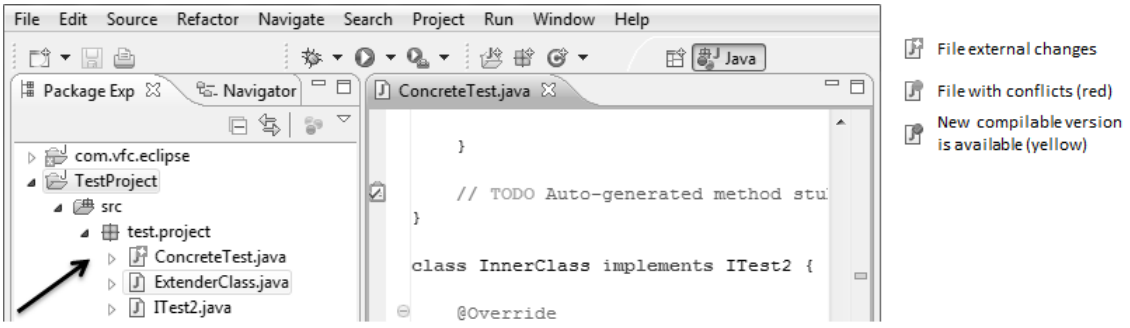

Figure 4.12: Local file differs from the server version

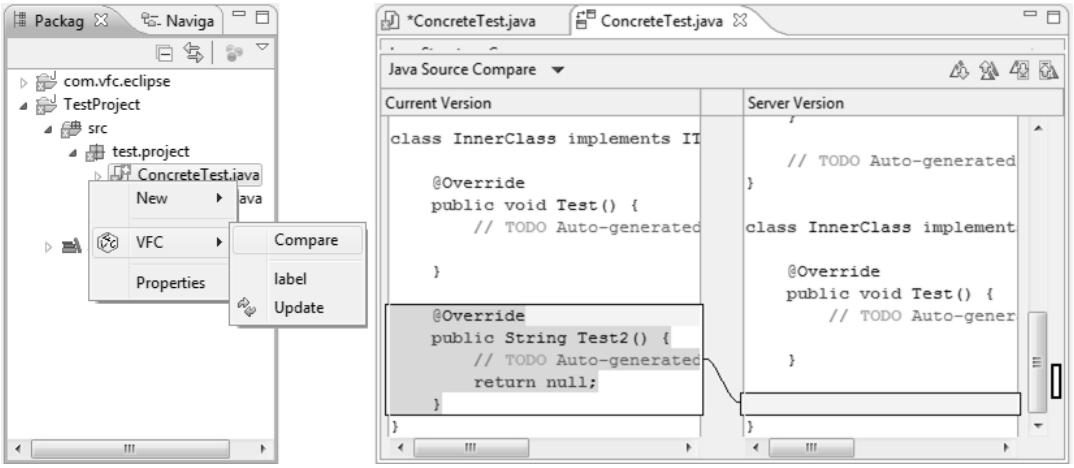

Figure 4.13: Comparing workspace with server version

appears with the option *'Compare'* (see Figure 4.13). If the user selects this option, a new window appears containing a *'Diff'* view of the changes between the two versions.

Additionally, a user does not have to wait for the project to become compilable in order to update its workspace with external changes; or otherwise conflicts would never be solved. In Figure 4.13 we can notice the existence of an *Update* option (in the VFC context menu). This option allows for the selected file to be updated with all the pending changes received from the server up to that point.

4.3.4 Detecting Compilable States

In order to detect compilable states of the project, the *ServerManager* periodically queries the *Dependencies Module*, which in turn checks if the Eclipse project has any compilation errors. If the project has changed since its last stable version, and there are no errors, then a new compilable version of the project exists. In these cases, an *Update Action* (see Section 3.3.2) is broadcast to all clients.

When a client receives the update message, the *'compilable' decorator* is applied to all files and folders of the project. Then, the user might chose to immediately update the project to the most recent compilable version, or to continue his work and apply the changes some time in the future. The process of updating the project is the exact same of updating a single file, only it is performed by right-clicking on the project's root folder.

An update message contains solely the number 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. To provide a safe-guard for delayed operations, for every new operation that arrives the client-side, its operation number is check. If it is equal or lower than the last state accepted by the user, then it is immediately applied to the workspace.

4.3.5 Conflict Detection

In our solution we assume that a conflict occurs, 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

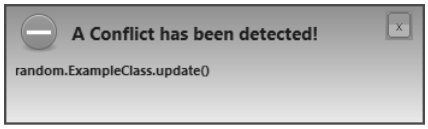

Figure 4.14: Notifying conflicts

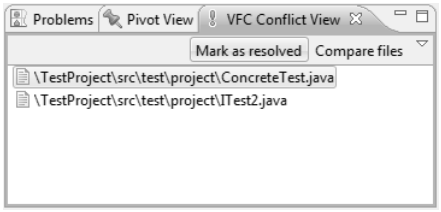

Figure 4.15: Managing conflicts

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 and sent to the conflicting users, the file icon associated with the document with conflicting changes is decorated with a conflict icon (red circle). Additionally, a dialog window fades-in at the lower right corner of the screen, alerting the user. Figure 4.14 shows the example of a conflict occurring in method *update()* of the class *ExampleClass*.

To provide the user with a general overview of all conflicting files, a new view was created: *Conflict View* (Figure 4.15). This view presents all the files with unresolved conflicts that exist in the project. After a user manually solves a conflict, he can mark that conflict as *resolved*, thus removing it from the *Conflict View* and also removing its conflict *decorator*.

These mechanisms allow users to be aware of possible conflicting actions in *near real-time*, thus preventing continuous conflicting actions to be performed and only detected in a later stage of the project.

4.3.6 Dialog Notification

Conflicts are but one of the three 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.

Even though the *Eclipse for Plug-in Developers* provides a series of extensions to the Eclipse UI, none of them had the characteristics we desired in the translation of a notification event. Hence, in this particular case, we made use of the Eclipse's GUI framework to create our own generic notification dialog. Which will be customized for every particular kind of event.

When the notification arrives the client's side it can be translated into three forms of pop-up dialogs:

- *Information Dialog*: Informs 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 amount of time;
- *Warning Dialog*: Alerts 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*: Appears whenever conflicts are detected. Its colour is red and only fades-out after the user clicks on the dialog.

4.3.7 Pivot in the IDE

As it was previously mentioned, there are two ways to declare a *pivot*: implicitly and explicitly. While the implicit way is completely transparent to the user and requires no interface, the same cannot be said for the explicit form.

To aid the users in the creation and destruction of pivots, we developed an additional context menu. Using our plug-in, when a user right-clicks in any position of an opened document, one of the available options is the addition of a new pivot (see Figure 4.16). 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 contains that position, and the *artefact* is sent to the *ConsistencyManager*.

Additionally, to inform the user of all the existing pivots in a document, we add *markers* to the line where the pivot was created (visible on the second panel of Figure 4.16). Finally, to visualize all explicit pivots set in the project, and to allow their removal, a *view* similar to the one displaying the project conflicts (Figure 4.15) was also created.

a) Adding a new *pivot* explicitly b) *Pivot* marker

Figure 4.16: Adding explicit *pivots*

Chapter 5

Evaluation

This chapter presents the evaluation of the VFC-IDE Plugin, and will be divided in three distinct sections: In the first section we will present a qualitative evaluation, discussing the level of success of the adaptation of the VFC algorithm to the context of collaborative software development; The second section is where the quantitative evaluation will take place, presenting statistics over the usage of network resources as well as server resources; In the third and last section a comparative evaluation will be made between the results obtained for the VFC algorithm and a solution using Maximum Consistency (MC).

5.1 Qualitative Evaluation

The goal of this dissertation 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.

In order to measure the quality of the implemented solution, we will monitor the behaviour of the algorithm and check if it is indeed capable of managing and compressing the operations propagated, based on impact metrics. We will test the enforcement of constraints, firstly on documents of a Java project which are not Java files (such as text documents), and later on actual Java files. Also, we will analyse the conflict detection feature, along with the developed mechanisms for aiding conflict resolution.

5.1.1 Absence of Structure

As it has been previously referred, the VFC-IDE Plugin determines regions of consistency based on the dependency between Java artefacts. However, not all files composing a Java project are actual Java artefacts. For instance, XML files are present in the majority of projects, as well as properties files; These files can hardly be associated with classes or methods and, as such, it falls beyond the capacities of our plugin to measure their dependencies with other files. Hence, remote changes to files that do not represent Java objects are assigned to the lowest region of consistency.

There are, however, two exceptions to these cases:

- Conflicts are still detected for concurrent changes to the same line, and assigned the highest level of consistency;
- Concurrent changes within the same file are treated as changes within the same class.

5.1.2 Constraint Enforcements

In the VFC model, remote operations are propagated to a given user whenever one of three constraints (Time, Sequence, Value) is reached, the following examples show how the actual behaviour of the algorithm is in accordance to the behaviour specified by the theory.

Example 1: Continuous Consistency

This example demonstrates the enforcement of the VFC consistency constraints in a simple Java project.

In this scenario, there are four programmers working collaboratively in a project which currently only has two classes, each class with a single method. As we case see from Figure 5.1, all users are located in different positions of the document: user A is editing the given method, user B is editing a method that uses user A's method, user C is editing an unrelated method in the same class as A and B, and user D is editing a different class. This example will be performed using **real-time edition** in order to ease the perception of arriving operations (notice that real-time edition still enforces VFC, but it applies incoming changes to the programmers workspace as soon as they arrive, see Section 4.3.3).

Now lets assume user A *pastes* a line (single operation) at the end of the method he is currently editing: "*System.out.println("Done.");*" (Figure 5.1). What happens is that the operation originated from user A is nearly instantly propagated to user B, given that user A changes are located in a method used by user B; hence having the second highest region of consistency. In this particular case the propagation of operations is triggered by the *Value* constraint, this happens because a relatively large amount of characters (28 characters) are contained in the user A's operation (Figure 5.2). After a few seconds, the *Time* constraint is triggered for user C, which becomes consistent with A and B (Figure 5.3). Finally, after a significant amount of time, the *Time* constraint is triggered for user D, thus placing all replicas in the same consistent state.

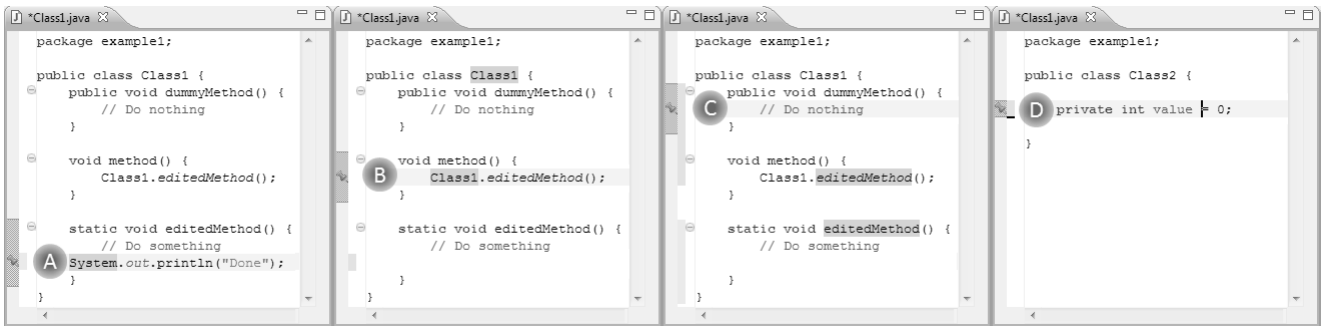

Figure 5.1: Example 1 - Insertion by user A.

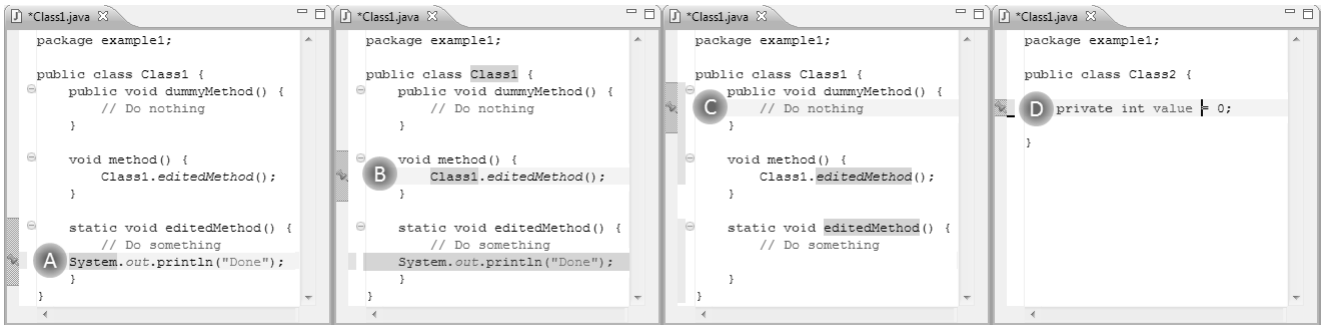

Figure 5.2: Example 2 - Value constraint triggered for user B.

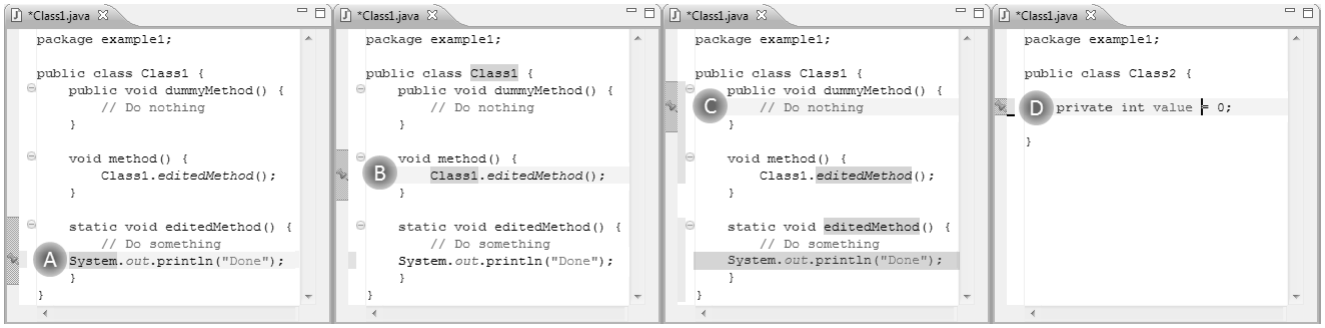

Figure 5.3: Example 3 - Time constraint triggered for user C.

In this particular case only *Time* and *Value* constraints were triggered. However, let's test a scenario in which user A inserts the new line not with a *paste* operation but by typing each character individually. In this situation, right after the insertion of the 10th character, the *Sequence* constraint is reached, and the first 10 operations are sent to user B. The *Sequence* constraint is to be triggered once again after the insertion of the 20th character, and the remaining 8 characters, of the original 28 characters inserted by user A, are only propagated to user B after the *Time* constraint is exceeded. The behaviour of user C e D remains the same.

Example 2: Pivot Usage

In order to test the influence of explicit pivots, in the consistency regions of a user, we performed a test very similar to the first example. However, in this new scenario user D added an explicit pivot to the method being edited by user B. As a result, right after the insertion of the new line by user A (Figure 5.1), user D *Value* constraint is triggered (as well as user's B constraint). As stated previously, after a few seconds user C becomes consistent with the replicas of the remaining users.

As observed, the usage of explicit pivots can present itself very useful when a user wants to ensure a high level of consistency in regions he is not currently editing.

In conclusion, the scenarios presented up to this point have shown how incoming remote operations result in the reaching of different types of constraint limits, depending on how distant these operations are from the user's point of edition. Additionally we demonstrated the difference between the three types of constraints (*Time, Sequence, Value*) that comprise the VFC algorithm. And finally, we evidenced the benefits that can be obtained from using additional explicit pivots.

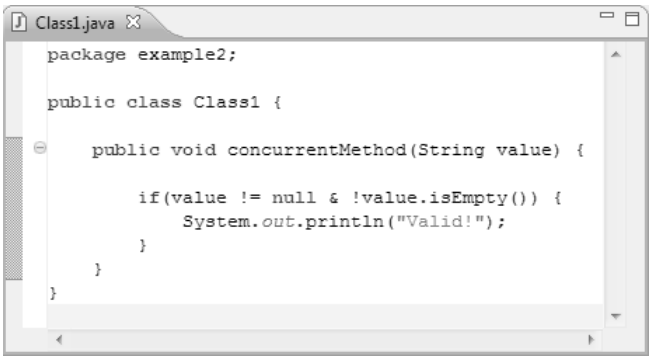

Figure 5.4: Method using binary *and* instead of logic *and*.

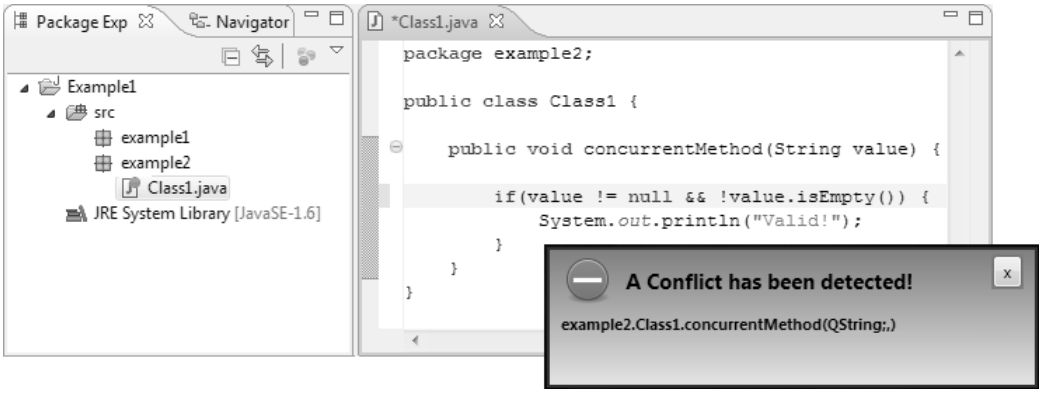

Figure 5.5: Conflict notification arriving user.

5.1.3 Conflict Detection

In the following scenario we wish to demonstrate the behaviour of the developed plugin during the arrival of conflicting operations to the server. For the purposes of this test, and to demonstrate our solution in the way that it is meant to be used, we turned on the **user-time edition** mode, this way, operations arriving the user are not automatically applied to the workspace until the user explicitly requests it (see Section 4.3.3).

There are two users A and B editing a project which recently reached a compilable state. However, as it can be seen in Figure 5.4, the *if* block in method *concurrentMethod()* has a misspelled *and* condition which might lead to an incorrect behaviour. At a given point in time, user A identifies the mistake and corrects it, coincidently, around the same time user B also attempts to correct the same mistake. These actions cause the server to arrive to an uncompilable state where the *if* condition now has: **if(value != null &&& value.isEmpty())**.

When the actions of user B arrive the server, the server detects that the two users are editing the same line and immediately dispatches the conflicting operations (associated with the region of greater consistency) along with a conflict notification to both users (Figure 5.5). After receiving the notification, user B compares its workspace with the pending changes not yet applied (Figure 5.6). After identifying the conflict, it performs an *update* action, which applies the pending incoming changes, and then manually resolves the conflict. Soon after the correction of the conflict, the server detects a new compilable version of the project, and notifies all users of the session (Figure 5.7).

Hence, using the VFC-IDE plugin, the users benefited from a higher degree of awareness which enabled

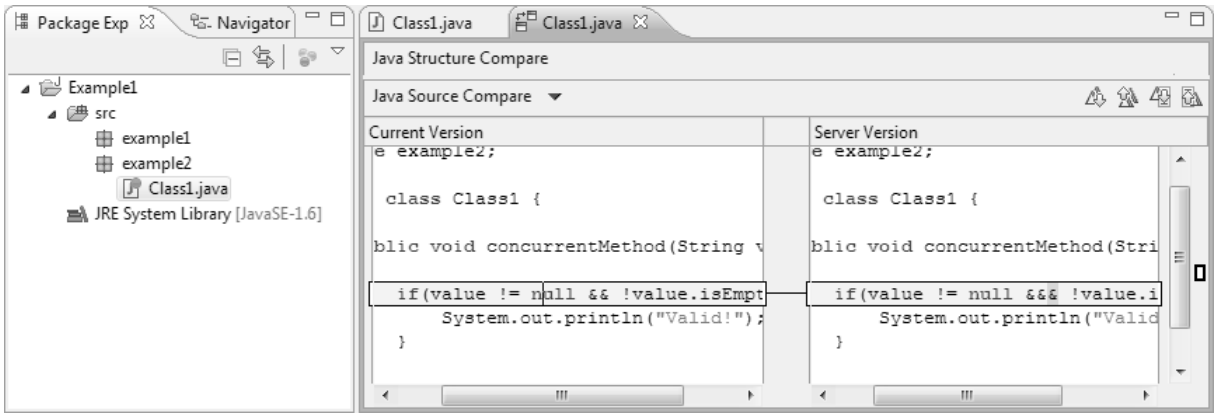

Figure 5.6: Comparing workspace with remote operations.

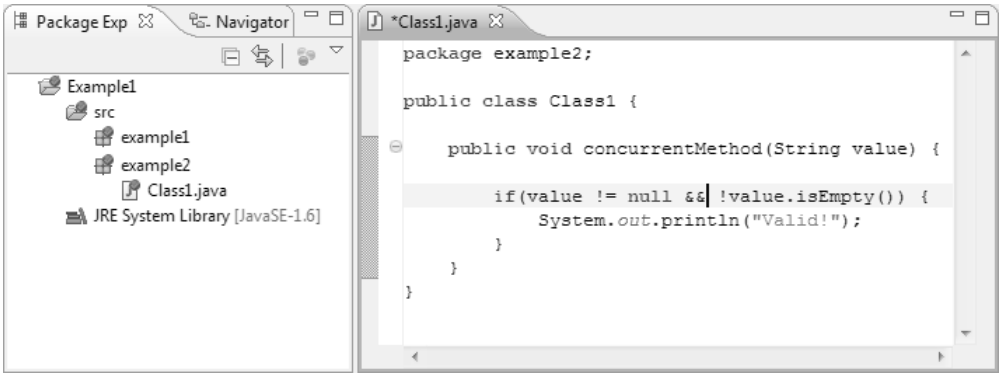

Figure 5.7: Conflict resolved, project arrives compilable state.

them to immediately detect conflicting operations. Additionally, the features provided by our solution allowed for a quicker identification, and resolution, of the actual conflict.

5.2 Quantitative Evaluation

In this section we will show the results obtained from the usage of the developed solution. With these results we were able to retrieve information providing a notion of the evolution of server's and client's system resources, as well as the overall bandwidth savings achieved through the VFC algorithm (when in comparison to the MC alternative). The results gathered with the MC aproach were inferred via an extra module, implemented in our plugin, which emulated the behaviour of an application with no consistency management.

The Eclipse plugin was tested in two distinct scopes. In the first, we used a bot developed specifically for this purpose (to which we will now refer to as *IntelliBot*), and which was designed to simulate the behaviour of a real software programmer. The *IntelliBot* is capable of performing valid semantic changes to a Java project, assuring that the changes lead the project to a compilable state (see Section 5.2.1). Tests using the *IntelliBot* were performed with one VFC-Server instance and a variable number of VFC-Client instances spread across two different machines; both machines using *Galileu* versions of Eclipse. 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*).

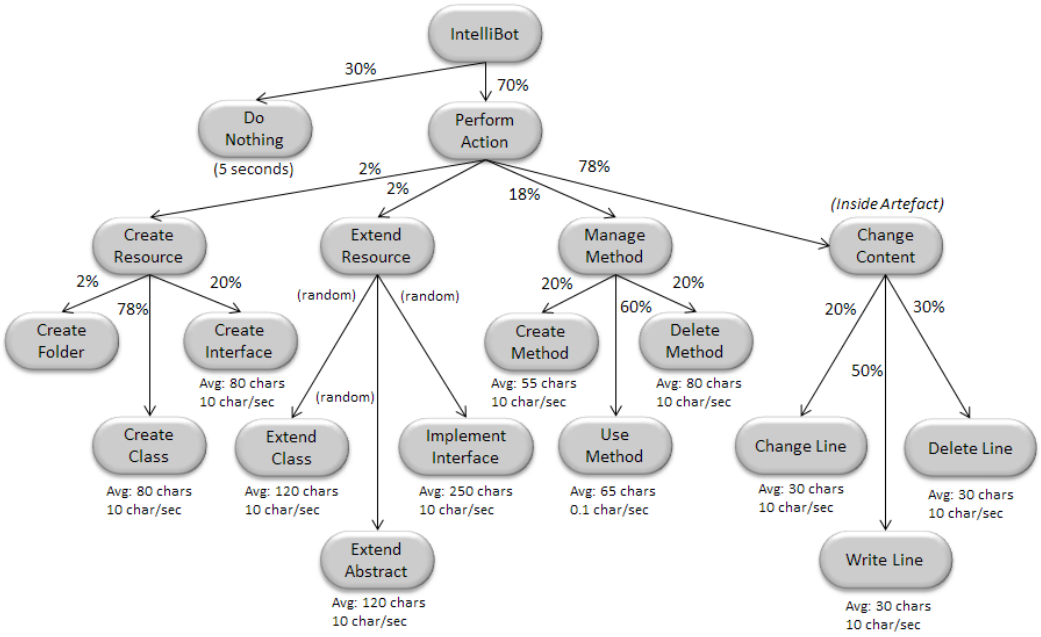

Figure 5.8: Behaviour Tree of IntelliBot.

5.2.1 IntelliBot

The *IntelliBot* is an additional Eclipse plugin developed specifically for this project, its goal is to simulate (in a very simplistic fashion) the programming patterns of a real software developer. Knowing that a bot only capable of performing blind updates to random positions in Java files would hardly provide realistic statistical data, we enriched the plugin with the capacity to infer a Java Project structure. Through the analysis of the Java artefacts composing a project, the *IntelliBot* is able to perform realistic changes, such as:

- Create Java resources (classes and packages);
- Extend classes and interfaces;
- Add method invocations to a given method;
- Delete methods from classes;
- Add and remove attributes of a class;
- Add, remove and change Java comments.

The content of all changes is designed in a way that assures that the project arrives to a compilable state. Also, the behaviour of our bot is parametrized by means of a behaviour tree (decision tree), which is described in detail in Figure 5.8, and which can be easily rebalanced to fit our needs.

Furthermore, to better simulate software development in the real world, where programmers usually perform their work with a certain degree of isolation, we associated each session of the *IntelliBot* with a given package. After being assigned a "root package", the majority (**not all**) of the *IntelliBot*'s updates will be performed within packages and sub-packages of its root. Note that, our bot will still use methods and objects from packages outside its root, it will just not directly modify their implementations in the majority of cases.

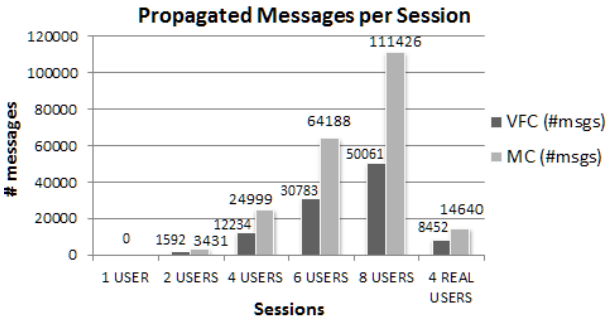

Figure 5.9: Total propagated operations per session.

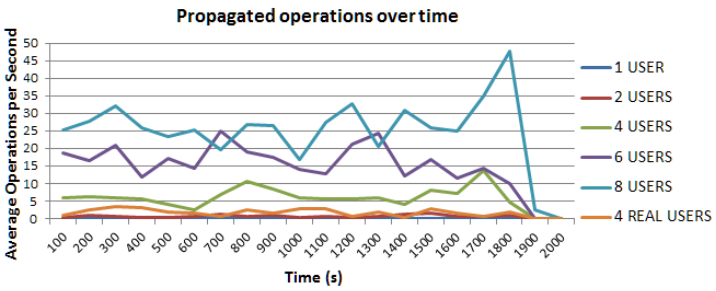

Figure 5.10: Evolution of average operations size over time.

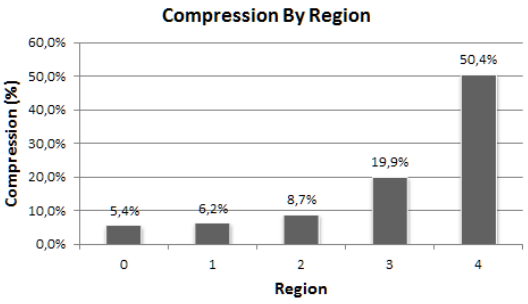

Figure 5.11: Op. compression per region.

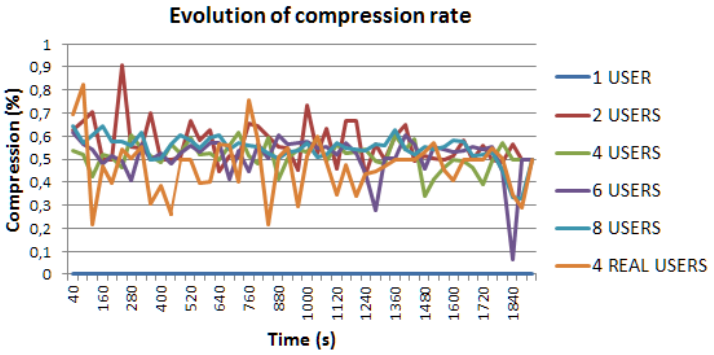

Figure 5.12: Evolution of average compression rate over time.

Finally, it is worth referring that, even though it tries to mimic the programmer's behaviour, the *IntelliBot* has a degree of activity (measured in operations per second) far greater than that of a normal programmer. The bot was intentionally parametrized this way in order to increase the level of stress on the server's side.

5.2.2 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 5.9).

Since consistency management, and consequential message compression, is only performed and measured at the server side, when a session only has one user there is no need to propagate the incoming messages; thus, in these cases there are no server outward messages.

An interesting aspect to be addressed is that, even though the compression rate is fairly significant, it seems to be almost completely unaffected by the number of participants of a session; in Figure 5.9 we can notice a very slight increase of the compression rate as the number of participants in a session grows (less than 1% per participant). Also, by studying Figure 5.12, we can conclude that the average compression rate seems to vary very little in time, neither having an increasing nor decreasing pattern as time advances.

The compression algorithm is able to detect linear inserts, or deletes, in the document and grouping the sequential operations into a single operation; thus saving in contextual information, that would otherwise

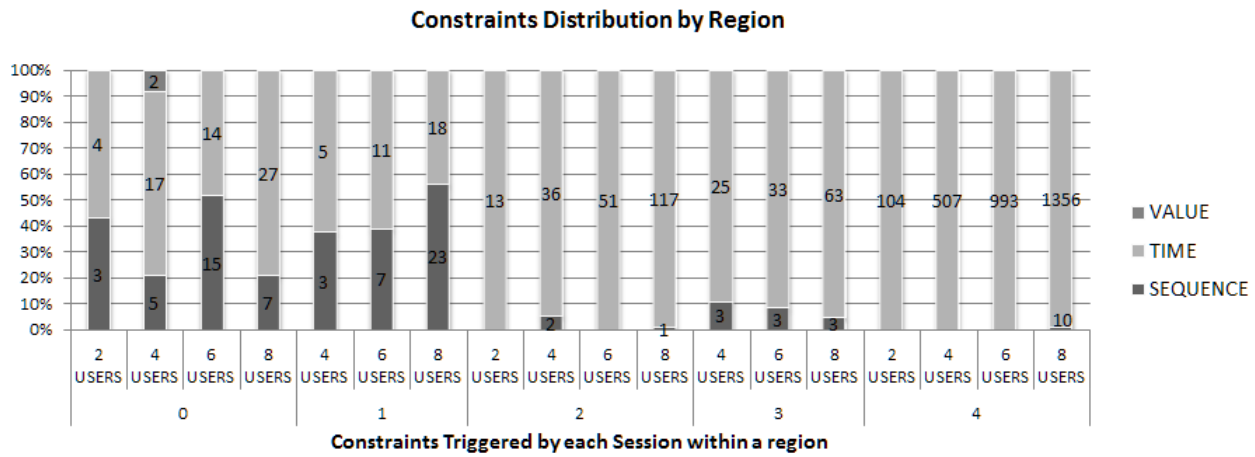

Figure 5.13: Constraints distribution per region.

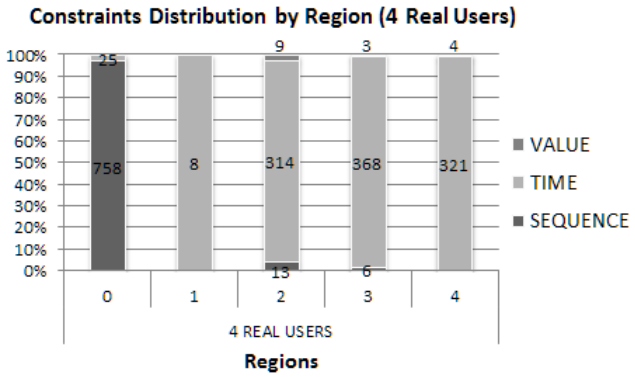

Figure 5.14: Constraints distribution per region, using real users.

be replicated in each individual operation (such as *DocPaths* and *version-vectors*). Additionally, the compression algorithm is able to detect operations that nullify each other, and prevents their sending altogether. 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. As the compression is essentially based on sequential insertions and operations that nullify each other, if a programmer writes code in a linear fashion (rarely changing its editing position), or if he performs undo operations regularly, the compression level is bound to increase.

Regarding consistency zones, by examining Figure 5.11 we can notice that, 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. Additionally, in Figures 5.13 and 5.14, which show how many times each type of constraint was triggered within each consistency region (for respectively, *IntelliBot* and a session with four real users), 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.

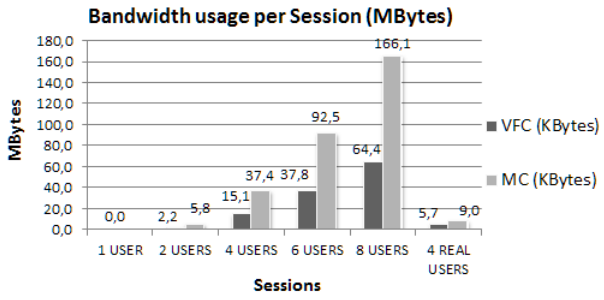

Figure 5.15: Total messages sent per session.

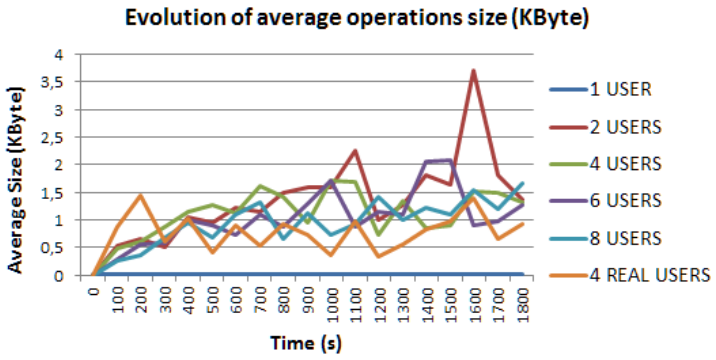

Figure 5.16: Total propagated messages per session.

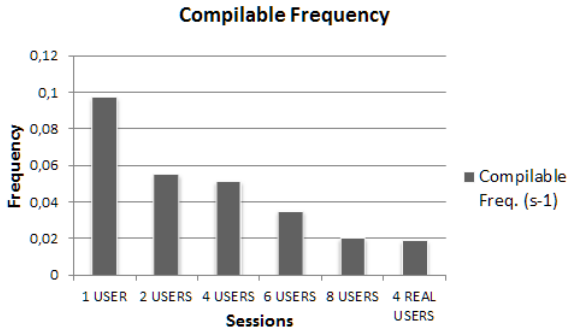

Figure 5.17: Frequency of compilable states detection.

5.2.3 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 5.2.2. As it can be seen in Figure 5.15, 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 3.3.5, 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. The later phenomenon can be observed in Figure 5.16, where 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*.

5.2.4 Compilable States

As we explained in Section 4.3.4, the VFC-Server is constantly looking for new compilable states of the shared project. By determining compile states during a project's development, the server is able to notify

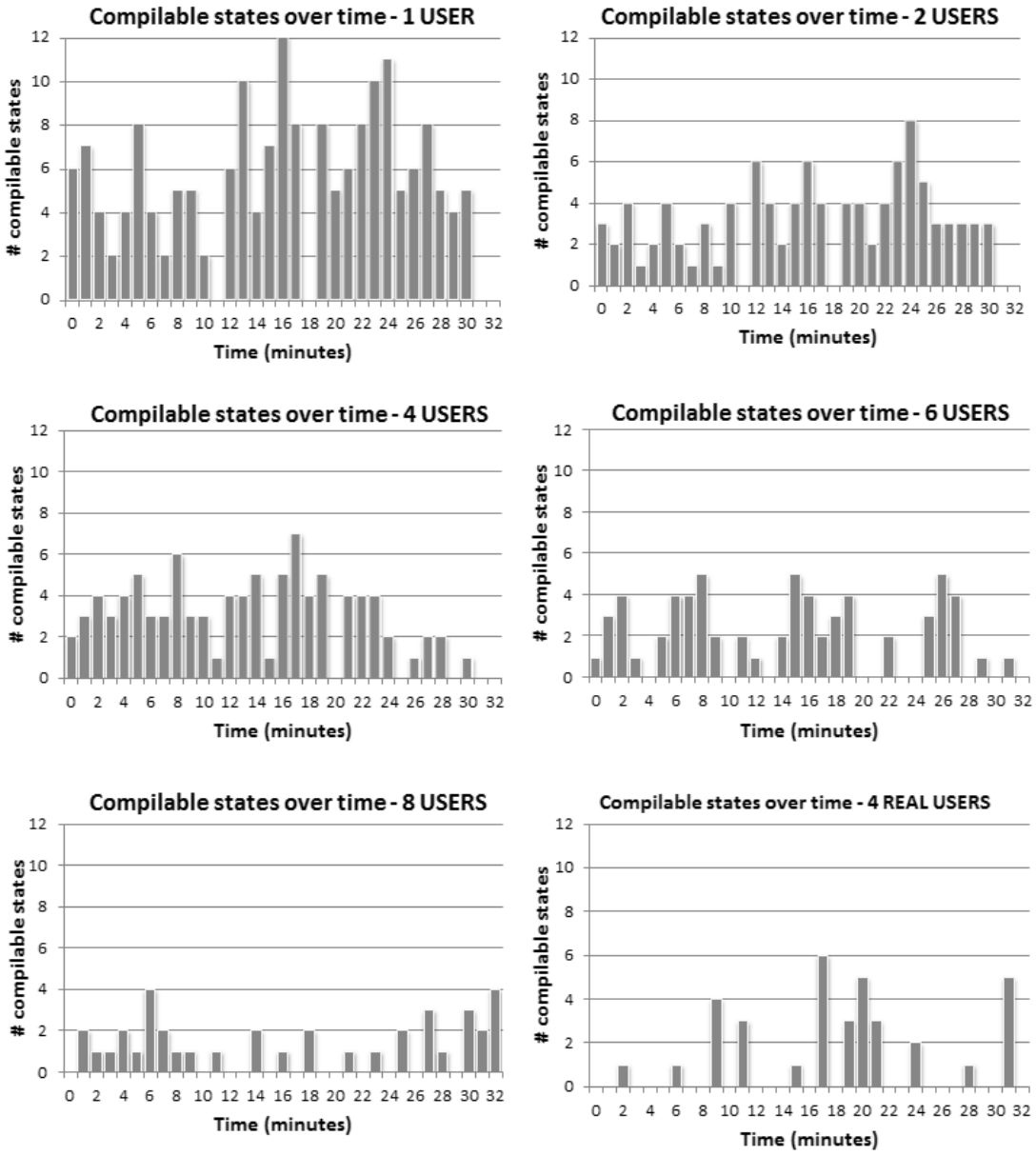

Figure 5.18: Compilable states over time.

the various participants of a VFC session that it is safe to apply external changes to their workspace. This being the case, it is crucial that the server has the capacity to identify these states regularly. In Figure 5.17 we captured the frequency with which the VFC-Server was capable of identifying compilable states of the project.

It is easy to observe that with an increasing number of participants the frequency of detected compilable states diminishes. This happens because, as the number of participants increases so does the number of operations being sent to the server, and as a result, the probability of a method or class being halfway through their edition at the moment the compilability-check runs becomes more and more elevated.

Figure 5.18 shows the number of compilable states detected by the server throughout sessions with different number of participants. 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. This, ultimately, means that the frequency with which compilable states are detected is almost completely dependent on the programmers behaviour. This last aspect is able to justify the lower frequency shown in the session with four real users.

Finally, we can conclude that the frequency with which compilable states are detected is satisfactory, even in sessions with a significant number of users; 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).

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.

5.2.5 System Resources

Additionally, using yet another Eclipse plugin, we were able to monitor the evolution of CPU and memory usage of our solution. In Figures 5.19, and 5.20 we can observe the resources used in the client and the server (respectively) of a VFC session with 8 users. 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 90 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 with 8 participants using *IntelliBot*, the rate by which operations are performed tends to exceed the rate by which they are dispatched through constraint violations, the messages will gradually accumulate until the activity decreases.

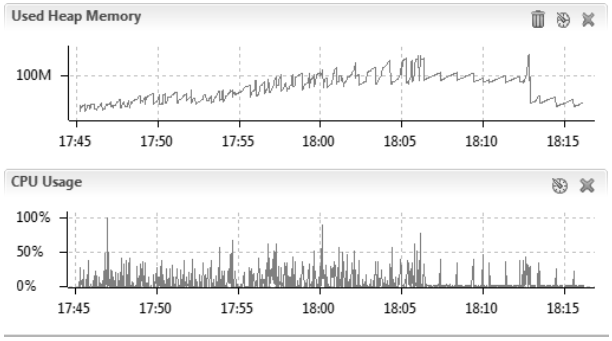

Figure 5.19: Usage of **client** resources over time.

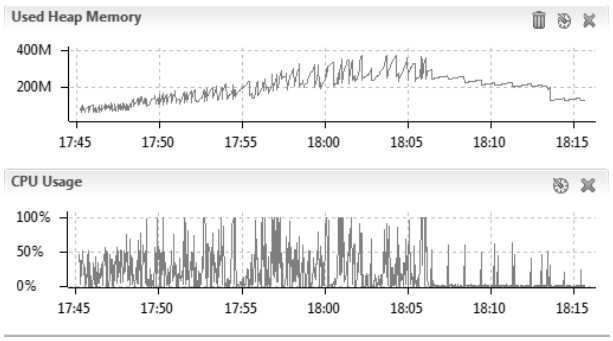

Figure 5.20: Usage of **server** resources over time.

5.3 Comparative Evaluation

Through the observation of the various tests performed in the previous section, 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. Moreover, we are able to naturally conclude that the gains increase even more significantly as we impose a greater delay between receiving and sending the updates at the server side, by reconfiguring the consistency zones constraints.

Consequentially, if the bandwidth usage is reduced, we can infer that, with the VFC approach, we are able to scale better and accommodate higher numbers of participants in a session. And also, we are capable of running a VFC session in an environment with high bandwidth constrictions.

Chapter 6

Conclusion

This dissertation 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 travelling 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.

Our architecture, and according implementation, was designed to adapt the existing concepts of the original VFC algorithm to the scope of collaborative edition of Java projects in the Eclipse IDE platform. Hence, we revised the notion of the user's location (*pivots*), along with the concept of distance between *pivots* and consistency zones. This was achieved through the implementation of a dynamic dependency-graph, capable of re-adjusting the relationship between the artefacts of a Java project as the project was being edited. Additionally, and to fit the purpose of concurrent document edition, we changed the update logic from *state-transfer* to *operation-transfer*. To achieve this, we implemented a *CRDT* structure, known as *TreeDoc*, capable of handling concurrency and ensuring compliance to and fulfilment of *intention*; and we extended it in a way that made it capable of precluding the need for causality support.

As to ensure functionality and usability, we enhanced our solution with two distinct modes of edition: one where external operations are immediately added to the programmer's workspace, and one where external operations are placed on-hold until they acquire explicit approval from the programmer; the later assures that the user is able to maintain his own compilable version of the project at all times. Along with the two edition modes, we developed a notification mechanism (embedded in the Eclipse GUI) which informs the programmers, in various ways, of significant changes within the project, such as compilable states and conflicts; and even means to aid the solving of the latter.

In conclusion, we verified three key aspects. Firstly, we were able to develop a solution for concurrent document edition with no loss of information or user *intention*; all through means of a plugin (compatible

between different Eclipse versions), and without changing the application's core. Secondly, we observed that higher compression ratios of the messages sent by the server are usually associated with zones with looser consistency constraints, and that they are mostly dependent on the behaviour of the programmers; being almost completely independent of the number of participants in a collaborative session. 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.

6.1 Future Work

Given the complexity of the VFC implementation, and due to restrictions in the scope of this work, we were unable to fully optimize some particular aspects of our solution. In this section we overview several enhancements and additional functionalities that should be considered in future iterations.

6.1.1 Detecting Unused Documents

As explained in Section 4.2.2, a lazy-load approach was taken into consideration when generating *TreeDoc* structures for each file of a VFC project. And although this solves the problem of having a big overhead at the beginning of a each session, as documents are being edited throughout a session, in time a large variety of files will have generated their correspondent *TreeDoc*, thus demanding for high memory requirements. Nonetheless, there might exist some files that were initially changed but have not been edited in a long time, and probably never will again if their development is finished.

Having such examples in mind, it would be interesting to develop a mechanism capable of detecting files which have had no changes for a significant period of time. In these cases we would assume that future changes were unlikely, and that it would be safe to destroy the document structure associated with the file. If, sometime in the future, this given file was to be subject to changes, it would behave as if it was its first time being edited, thus creating a new balanced *TreeDoc*. Such protocol assures, that during the lifetime of a VFC session, only the strictly necessary structures are kept in memory.

6.1.2 Backup Deleted Resources

In our solution, we track, detect and handle concurrency between resources (files and folders) through means of *version-vector* (see Section 3.3.6). This vector is able to disambiguate concurrent actions over a given resource, such as creating, deleting or moving a file. However, if a given programmer is editing a document and concurrently another programmer deletes this same document, the first programmer would lose all the work performed in that file up to that point. We propose a mechanism, which upon receiving a delete operation over a file with local changes, is able to prompt the user to choose among a variety of options, such as: "cancelling" the deletion (by re-creating the deleted file), create a local backup of the file (which is not shared among the participants), or discard the local changes.

6.1.3 Selective State-Updates

In Section 2.3.2 we mention the existence of two distinct approaches: *state-transfer* and *operation-transfer*, and more importantly we refer some alternative solutions which try to mix these two. Likewise, in the scope of our work, it would be interesting to create a "hybrid" approach capable of choosing between *state-transfer* and *operation-transfer*. For instance, in regions of looser consistency, instead of dispatching a large amount of text operations, we could simply dispatch a single message containing the state of the entire document (or class, or method); thus achieving significant bandwidth savings. This might, however, incur in a great level of complexity due to the *TreeDoc* associated we each document.

6.1.4 Revision-Based Update

One of the features implemented in our solution is the capacity for a programmer to explicitly determine when it is safe to include external operations to his workspace. Presently, the programmer is only given the choice to update his workspace to the last compilable state detected by the server (or to "blindly" include all remote changes to a specific file). However, it would be interesting to provide the programmer with the capacity to choose between the multiple compilable states which have been detected since the last time he updated its workspace. Additionally, it might also be useful to allow the programmer to only update his workspace with the changes from a specific user. The latter is specially useful if the two programmers' work is related.

6.1.5 Distributed VFC Management

Currently, the VFC algorithm is being enforced only by a single participant, the VFC-Server. This, however, incurs in a single point of failure, as well as an evident bottleneck to the system; which might result in degradation of performance as the number of participants of a session increase. This way, it would be ideal to adapt the VFC-IDE solution to a fully distributed model, where every participant would be able enforce the imposed consistency constraints, on a subset of the artefacts, over the remaining participants.

Bibliography

[1] D. Agrawal, A. El Abbadi, and A. K. Singh. Consistency and orderability: semantics-based correctness criteria for databases. *ACM Trans. Database Syst.*, 18:460–486, September 1993.

[2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society.

[3] G. Booch and A. W. Brown. Collaborative development environments. volume 59 of *Advances in Computers*, pages 1–27. Elsevier, 2003.

[4] P. Cederqvist. Version management with cvs. Sweden, 1993.

[5] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 289–300, New York, NY, USA, 2005. ACM.

[6] 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*, eclipse '03, pages 45–49, New York, NY, USA, 2003. ACM.

[7] M. Chu-Carroll and J. Wright. Supporting distributed collaboration through multidimensional software configuration management. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management*, volume 2649 of *Lecture Notes in Computer Science*, pages 40–53. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-39195-9_4.

[8] S. Dart. Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software configuration management*, SCM '91, pages 1–18, New York, NY, USA, 1991. ACM.

[9] 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*, GROUP '03, pages 105–114, New York, NY, USA, 2003. ACM.

[10] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[11] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407, New York, NY, USA, 1989. ACM.

[12] S. I. Feldman. Make a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[13] R. A. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5:5–4, 1992.

[14] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. pages 365–394, 1976.

[15] I. Greif, R. Seliger, and W. E. Weihl. Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 160–172, New York, NY, USA, 1986. ACM.

[16] R. E. Grinter. Using a configuration management tool to coordinate software development. In *Proceedings of conference on Organizational computing systems*, COCS '95, pages 168–177, New York, NY, USA, 1995. ACM.

[17] J. Grudin. Why cscw applications fail: problems in the design and evaluationof organizational interfaces. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, CSCW '88, pages 85–93, New York, NY, USA, 1988. ACM.

[18] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conways law and beyond. *IEEE Software*, 16:63–70, 1999.

[19] J. H. Howard and J. H. Howard. Reconcile user's guide. Technical report, 1999.

[20] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20:350–353, May 1977.

[21] 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*, CSCW '04, pages 21–24, New York, NY, USA, 2004. ACM.

[22] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 210–218, New York, NY, USA, 2001. ACM.

[23] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10:3–25, February 1992.

[24] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Trans. Database Syst.*, 19(4):586–625, 1994.

[25] T.-W. Kuo and A. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *IEEE Real-Time Systems Symposium'92*, pages 35–45, 1992.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[27] D. Li and R. Li. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 457–466, New York, NY, USA, 2004. ACM.

[28] R. S. Mars, M. Raynal, M. Raynal, M. Singhal, and M. Singhal. Logical time: A way to capture causality in distributed systems. Technical report, 1995.

[29] P. Molli, H. Skaf-molli, and C. Bouthier. State treemap: an awareness widget for multi-synchronous groupware. In *INTERNATIONAL WORKSHOP ON GROUPWARE*, pages 106–114, 2001.

[30] K. L. Morse. Interest management in large-scale distributed simulations, 1996.

[31] J. Munson and P. Dewan. A concurrency control framework for collaborative systems. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, CSCW '96, pages 278–287, New York, NY, USA, 1996. ACM.

[32] C. O'Reilly, P. Morrow, and D. Bustard. Improving conflict detection in optimistic concurrency control models. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management*, volume 2649 of *Lecture Notes in Computer Science*, pages 61–69. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-39195-9_14.

[33] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for p2p collaborative editing. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 259–268, New York, NY, USA, 2006. ACM.

[34] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9:240–247, May 1983.

[35] 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*, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[36] W. Reinhard, J. Schweitzer, G. Volksen, and M. Weber. Cscw tools: Concepts and architectures. *ACM Comput. Surv.*, 27(5):28–36, 1994.

[37] B. Richard, D. Mac Nioclais, and D. Chalon. Clique: A transparent, peer-to-peer replicated file system. In M.-S. Chen, P. Chrysanthis, M. Sloman, and A. Zaslavsky, editors, *Mobile Data Management*, volume 2574 of *Lecture Notes in Computer Science*, pages 351–355. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-36389-0_27.

[38] H.-G. Roh, J. Kim, and J. Lee. How to design optimistic operations for peer-to-peer replication. In *JCIS*, 2006.

[39] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001. ACM.

[40] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[41] 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*, pages 80–100, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[42] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: Raising awareness among configuration management workspaces. *Software Engineering, International Conference on*, 0:444, 2003.

[43] A. Serban. Visual sourcesafe 2005 software configuration management in practice. New York, NY, USA, 2007.

[44] M. Shapiro and N. Preguiça. Designing a commutative replicated data type. Technical report, Computer Science Dept: University of Copenhagen, 2007.

[45] M. Shapiro, N. Preguiça, and J. O'Brien. Rufis: mobile data sharing using a generic constraintoriented reconciler. In *Mobile Data Management*, pages 146–151, Berkeley, CA, USA, 2004. IEEE.

[46] H. Shen and C. Sun. Flexible notification for collaborative systems. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, CSCW '02, pages 77–86, New York, NY, USA, 2002. ACM.

[47] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11:92–99, January 1992.

[48] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68, New York, NY, USA, 1998. ACM.

[49] 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.*, 1(5):63–108, 1998.

[50] 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.-Hum. Interact.*, 5:63–108, March 1998.

[51] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.

[52] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4:180–209, June 1979.

[53] 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*, GROUP '05, pages 136–139, New York, NY, USA, 2005. ACM.

[54] S. Weiss, P. Urso, and P. Molli. Wooki: a p2p wiki-based collaborative writing tool. In *WISE'07: Proceedings of the 8th international conference on Web information systems engineering*, pages 503–512, Berlin, Heidelberg, 2007. Springer-Verlag.

[55] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, pages 404–412, Washington, DC, USA, 2009. IEEE Computer Society.

[56] 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*, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[57] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. *ACM Trans. Comput. Syst.*, 24(1):70–113, 2006.