IT UNIVERSITY OF COPENHAGEN

Software Development Group

ABC4GSD: DESIGN AND IMPLEMENTATION OF AN INFRASTRUCTURE FOR SUPPORTING ACTIVITY-BASED COMPUTING PARADIGM IN GLOBAL SOFTWARE DEVELOPMENT

Author:
PAOLO TELL

Author:
MUHAMMAD ALI BABAR

Version	Date	Description
1.0	110603	Creation date
1.1	110604	Structure defined, inserted initial content
1.2	110606	Structure changed, future work should be more design consideration
1.3	110610	First complete draft
1.4	110615	Improved 5.5 data model
2.0	110805	0mq introduced
3.0	111202	New structure

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It has become a common practice to develop all or part of software with teams distributed across multiple sites either within the same country or in different countries. Developing software in a geographically distributed arrangement is known as Global Software Engineering (GSE) or Global Software Development (GSD) [19]. Companies have been increasingly adopting the GSD paradigm by moving their software development to places considered to be lower cost destinations or to destinations where the required skills are more readily available [13]. As a result of the growing adoption of GSD, more and more software development projects are being undertaken by following some kind of GSD model that has several unique aspects. GSD projects are usually characterized by geographical, temporal, cultural and linguistic distances [36]. These distances usually lead to a significantly increased complexity for project teams that are expected to face several kinds of new challenges [21][13]. Therefore, one of the key challenges of GSD projects is to provide distributed software development teams with appropriate tools to help them to effectively and efficiently perform their software engineering tasks, which are likely to require an increased degree of communication, coordination and collaboration among a project's team members.

Given the critical role and importance of adequate tooling support for GSD teams, the Software Engineering (SE) community has developed dozens of tools [30][11][53][47]. Some researchers have provided a summary of technologies and tools being used by GSD teams [34]; others have proposed frameworks for categorizing and comparing specialized technologies (e.g. the coordination pyramid [42]) according to which coordination tools can be placed into three categories: communication (e.g. Skype and email), artifact management (e.g. code repositories) and task management (e.g. project planning tools and workflow systems). Authors have emphasized the importance of integrating tools from three categories for supporting *continuous coordination* [42].

We are currently undertaking a large scale literature review study to systematically mapping the technological support reported in scientific publications. Our preliminary analysis of the data extracted from the reviewed studies shows that a significant number of publications describe tools developed to enhance particular development processes (e.g., inspection

meetings in [31]; requirement management in [29]) or to incorporate information from different sources into monitoring applications (e.g. T3 presented in [51] or FastDash described in [8]). However, one of the main challenges that has been identified in the context of GSD is the lack of integration among different kinds of tools that are likely to be used by GSD teams [25].

We assert that the Activity-Based Computing (ABC) paradigm can be leveraged to provide a flexible and easily adaptable infrastructure for building and using activity-aware tools for GSD teams [49]. Our effort to introduce the ABC principles in the context of GSD has been motivated by its successful adoption in hospitals for supporting nomadic and distributed users in handling concurrent activities for intensive collaboration [7]. We argue that an ABC based infrastructure can aggregate different resources required for a particular software development activity, support multi-tasking among different activities, allocation and reallocation of the required resources to different activities as well as create, maintain and support awareness at multiple levels.

The main goal of this paper is to report and discuss some of the significant requirements that an infrastructure has to fulfill to successfully support ABC in the context of GSD. By surveying the literature, we have identified the requirements that a system for supporting GSD teams needs to consider and address. The reported requirements have been identified based on a systematic analysis of a selected set of papers on tooling support for distributed teams, our previous work on GSD related topics [1][4][23][3], and our observations and experiences of studying globally distributed software development teams. We will also describe how the current desktop metaphor limitations prevent existent technologies to truly support GSD practitioners. After describing the principles underpinning the ABC paradigm detailed in [7], we will use a scenario to explain how a system based on the ABC paradigm can effectively and efficiently support GSD practitioners by fulfilling the identified requirements. These requirements are mainly expected to benefit us in providing a roadmap to build such an infrastructure. However, we also hope to encourage the community to use the requirements for their own purposes and to leverage the features provided by the infrastructure when developing new tools or tailoring existing one.

The reminder of this paper is organized as follows: Section **??** gives an overview of the current technologies supporting GSD teams and describes the ABC paradigm and its principles. The system requirements are identified in Section **??** and a scenario describing the benefit of introducing a system based on the ABC paradigm in the GSD context is described against the previously identified requirements in Section **??**. Finally, Section **??** and **??** conclude the paper.

Chapter 2

Background and Motivation

In this section we will discuss the background and motivation of the work reported in this paper by providing the context in which we are identifying the requirements for introducing ABC in GSD. We also describe the theoretical background underpinning the ABC paradigm and its principles.

GSD has become an established software development paradigm. An increasing number of companies are getting their software developed by geographically distributed teams or relocating their software development centers at different geographical locations to gain several perceived benefits such as access to a larger number of skilled personnel at relatively low cost or reduced development time as a result of 'follow the sun' strategy [19]. Adequate tooling support is essential to successfully support GSD teams [30]. The Software Engineering (SE) community has developed dozens of tools for supporting distributed development teams and many GSD projects have adopted tools developed and used by the Computer Supported Cooperative Work (CSCW) and Open Source Software (OSS) communities (e.g. [30][11][53][47]).

The importance of providing adequate tooling support for GSD teams has resulted in the development of several dozen of research and commercial tools. Researchers have reported extensive surveys of tools for GSD teams [53]. An overview of commercial and open source technologies and tools for distributed teams has been reported in [41]. Lanubile et al. have reported an overview of collaboration tools for global software engineering in [30]. Sarma et al. [42] have reviewed coordination technologies and proposed a framework to categorize them; a classification driven by the software life cycle has also been proposed [34]. Furthermore, some researchers have reported systematic literature reviews; for example modeling tools were analyzed in [11] and, the authors of [47] systematically identified and reported awareness support for GSD.

A significant number of publications describe tools developed to enhance particular development processes (e.g. inspection meetings in [31]; requirement management in [29]) or to incorporate information from different sources into monitoring applications (e.g. T3

presented in [51] or FastDash described in [8]). However, one of the main challenges that has been identified in the context of GSD is the lack of integration [25]. Throughout the software life cycle many diverse tools are used and the improvement of one of them can hardly have a broad enough impact to truly lessen the GSD related issues. To provide a better integration, two different approaches have been followed: one operates at the model level, whereas the other at the application level. The idea behind the former is to provide a common abstraction layer through which applications can communicate and have a common understanding about the artifacts handled. Examples of such approach are represented by SoftDock [48], which allows to collaboratively model software components in UML or ADAMS [14] that helps improve project and artifact management by providing a fine-grained versioning system that is able to enhance traceability. Similar to the tools developed for the improvement of one particular process, these solutions provide support only to one specific aspect of working environment. They are unable to sufficiently support coordination, collaboration, communication and awareness, which are considered some of the most significant GSD challenges to be addressed. The application level integration approaches usually offer solutions to make diverse set of tools work together by providing a common way of interacting through an infrastructure or middleware. These kinds of solutions can help improve a working environment through an interaction mechanism to support different enhancements. An interesting example based on a client/server architecture has been proposed in [33]. The reported middleware was developed to enhance CASE tools for providing awareness information via an event mechanism. There are also reported solutions based on a peer-to-peer architecture in [40] and [35]. The architecture presented in the former is used to enhance collaboration in a distributed environment, whereas the authors of the latter describe an environment able to provide applications integration.

Collaborative desktop environments (CDE) like these are not new concepts; Booch and Brown defined them as *"a virtual space wherein all the stakeholders of a project even if distributed by time or distance may negotiate, brainstorm, discuss, share knowledge, and generally labor together to carry out some task, most often to create an executable deliverable and its supporting artifacts."* [10]. However, collaboration is just one of the aspects that have been identified as a key challenge to be addressed in the context of GSD. Together with communication and coordination, collaboration forms the so-called *3C*s model. Initially identified for groupware technologies [16], the model has been widely accepted to provide classifications of tools in the SE context (e.g. [47]).

Despite the development of a significant number of tools, GSD practitioners have been reporting significant challenges that can be ascribed to lack of sufficient support for collaboration, coordination, and communication. In order to address these challenges, several efforts have been dedicated to enhance Integrated Development Environments (IDEs), which can be easily augmented with plug-in based solutions such as Eclipse (e.g. IBM Jazz [24], Mylyn[1] [26]) or Visual Studio (e.g. FASTDash [8]). One of the most successful examples of such tools is Jazz [24]; it consists of a set of plug-ins for Eclipse to provide status- and task- related awareness about team members. The most notable visual element is the Jazz band: a plug-in that enhances Eclipse by providing a list of people participating in the project inside the IDE. The IBM Rational Jazz supports the whole system; a platform that provides a binding service layer to allow information sharing throughout the linked components. Though there have been several successful reports of integrating and using different GSD tools developed as plug-in, most of them have been designed to address one particular challenge faced by GSD teams. Moreover, these solutions are usually designed for certain applications and may not interoperate with other applications. Furthermore, it is not pos-

[1] Originally called Mylar

sible to use such application in ways not foreseen when designed; thus, denying the benefits that could be gained by approaching GSD problems as a whole related system to support.

To concretely improve the 3Cs in GSD, we assert that the current desktop based metaphor may not be suitable for designing tools. Rather, the file- and application- based paradigm is outdated and does not provide any built-in support for coordinating, communicating or collaborating more effectively when working in teams. It was developed to support tasks performed by a single user.

We believe that there is a need of an infrastructure that can support collaboration, communication and coordination upfront; an infrastructure able to enhance practitioners' usage experience without restricting them by any means. Such an infrastructure should enable tools to adapt to the activities to be performed by GSD team members. The GSD team members should be able to aggregate all the required resources (e.g. artifacts, contacts, and applications) for a particular development activity such as designing a software in a collaborative arrangement, be able to switch between activities and hand-over activities to other team members potentially geographically distant, allocate and reallocate resources for different activities as well as be aware of other members' working status. Such an infrastructure should allow existing and new tools to be activity-centric.

Our approach aims to leverage the Activity-Based Computing (ABC) metaphor to provide a loosely coupled infrastructure that can spread the required information among GSD teams through an event-driven system [49].

2.1 Activity-Based Computing (ABC)

The concept of activity is the core of the Activity-Based Computing (ABC) initially introduced by Norman in the 90s. In his book [37], he states that *"[...] the basic idea is simple; make it possible to have all the material needed for an activity ready at hand, available with little or no mental overhead."*. Thus, the core concept of ABC is to provide an automatic, seamless and non-intrusive support for activities.

Bardram has successfully demonstrated the applicability of the ABC paradigm for supporting different collaborative activities in hospitals [7]. He developed a framework that provides a robust replacement of the application-oriented computing paradigm by focusing on the principles of roaming, adaptation, sharing and awareness. While the activities in hospitals differs from the ones in software engineering as the mobility is a fundamental characteristic to be tackled in hospitals, we assert that the six core principles of the ABC paradigm [7] can provide a solid foundation for building an infrastructure that can help address many of the existing 3Cs related challenges. Following are the six principles of the ABC paradigm.

I **Activity-centered** - An activity is an entity intended to collect all information needed to allow a user carrying out a specific task; they are the core blocks of the metaphor.

II **Activity suspension and resumption** - This principle is meant to support the users' nature of alternating between the different tasks they may be involved in.

III **Activity roaming** - The system needs to grant a persistent model of the activities to allow the resumption of an activity on different workstations connected to the system.

IV **Activity sharing** - Due to the distributed nature of the system, activities are shared among users participating in a specific activity granting an easy way to support collaboration.

V **Activity adaptation** - An activity is able to resume itself on different devices, adapting to the available resources.

VI **Context-awareness** - An activity is able to resume itself adapting and adjusting according to the context it is resumed in.

Chapter 3

Scenario analysis

To better explain the potential benefits that the introduction of the ABC paradigm would bring to the GSD environment, we outline a concrete scenario that tackles the requirements enlisted in Section **??** by leveraging the principles described in Section 2.1. The GSD scenario described below considers a working environment where the ABC principles have been introduced (*principle: I*). The collaborative environment used is based on a centralized platform (*unity*) where all information are stored and handled through different services deployed in a cloud-enabled infrastructure (*scalability*). The infrastructure is designed to permit heterogeneous applications to be hooked to the system (*flexibility*) by implementing a specific interface that allows them to feed and be fed by information flowing through the provided event based system. The interface requires the implementation of some methods, which allow the system to automatically introduce the ABC paradigm (*intuitivity*). Finally, the number and type of events a single application subscribes to is a decision completely left to tool developers (*flexibility*).

This scenario is based on a hypothetical software development project undertaken by a company which follows the GSD paradigm. The company's headquarter is located in Copenhagen close to the stakeholders. The company has two development center in Zurich and Bangalore; whereas, the quality assurance (QA) center in New York. Considering the month of June, the time zone differences with respect to the headquarter are +3:30 for the center in Bangalore, +0:00 for the one in Zurich and -6:00 for New York. Beyond being engaged in GSD, the company is adopting a slightly modified version of the distributed Scrum methodology described by Schwaber [43]. Thus, they hold scrum of scrums meetings on a three week bases (i.e. Monday, Wednesday and Friday at 14:30 Danish time) instead of performing them daily as the normal Scrum methodology requires. Further, the standup meetings are held locally at 10:00 for all departments apart from Copenhagen in which it is done at 17:00. Peter is the team leader and works in the Danish branch with Ali, a developer; Paolo, another developer, works in the Swiss department; Aufeef is the leader of the group in Bangalore; and Jakob is the manager of the QA center in New York.

It is Thursday afternoon and Peter is having a meeting with the stakeholders to discuss

new functionalities; Ali is designing interfaces that will be needed soon and Paolo is fixing some functionality issues (yes, bugs). Once the meeting is finished around 17:00, Peter and Ali hold the standup meeting in which they decide that a new activity has to be created to discuss and work on the newly identified requirements. After creating the activity as a sub-activity of the main project, Peter links the meeting minute and other documentation artifacts as well as the two developers (i.e., Ali and Paolo) to the activity (*coordination*). Being online, Ali and Paolo are immediately informed through a notification (*awareness*) that a new activity involving them has been created. Paolo decides to postpone any action not to loose his concentration on the current bug he is working on, whereas Ali starts working on it after the meeting by opening the IDE with the project related source code and the modeling tool with the UML diagram of the project. After sharing both of them with all the others participants of the activity (*principle: IV*), Ali starts designing and creating some object skeletons. Once the bug is fixed, Paolo checks his activity list as he remembers the one previously ignored and accepts it. The bug fixing activity is suspended and the new one resumed (*principles: II - III - V; intuitivity*) by automatically visualizing all the people involved, the related artifacts and the applications (i.e. modeling tool and IDE). After checking the meeting minutes he decides to discuss with Ali via the embedded messaging system to catch up with the work Ali is doing (*communication*). After some brief explanation they agree on synchronizing the modeling tool and start a voice communication to engage a closer collaboration (*principle: IV; collaboration*). Finally, before finishing the day they include Aufeef and Jakob in the activity (*coordination*).

It is the early Friday in Bangalore when Aufeef arrives at the office; he logs into the system and discovers that he has been allocated to a new activity; after resuming the activity (*principles: II - III - V*) his contact list is populated with the people involved in it and sees that Ali and Paolo are also involved but offline (*awareness*). By reading the logs and studying the source code he realizes that there is a conflict in the interfaces that was not identified by the others; during the standup meeting with his group they agree on the need of discussing the issue during the next scrum of scrums meeting as its modification involves a wide variety of artifacts. Therefore, Aufeef and his team continue working on other project related activities. It is around 18:00 and the scrum of scrums is about to begin, Aufeef goes in the meeting room. Once he logs into the meeting room workstation, his last activity is resumed loading all the related artifacts and, each application, is adjusted to fit in the new environment (*principles: II - III - V*). The meeting starts and Aufeef explains the discovered issue by starting a synchronous sharing of the needed applications (*principles: IV - V*) to support his presentation. During this meeting other activities to support the QA department are created. They collect the test procedures that the participants pointed out to be crucial and that represent known risks.

The development continued smoothly and the teams succeeded in meeting the deadlines. Jakob is preparing the presentation and the demo for the sprint meeting. He creates his own activity to arrange the work as it best suits him and, once finished, he shares the activity with Peter so that he will be able to see all the artifacts (*principle: IV*) and present it to the stakeholders in the Danish branch.

In a working environment lacking any support for computer-mediated teamwork this scenario would have entailed different procedures and tools to achieve a similar result:

- Peter would have used a project planning tool to manage the new task by assigning the two developers and the related digital resources to it;
- Ali, Paolo and Aufeef would have had checked the project planner for updates or

eventually would have received a notification email to realize the presence of a new task;

- to check the related digital artifacts the developers would have needed to manually access some sort of repository or versioning system;
- to communicate with Paolo, Ali would have had to find Paolo's contact and would have required a specific tool to engage the communication, potentially one for the chat and one for the voice communication; this applies also for all the other communication activities described in the scenario;
- it is difficult to imagine that a single application sharing would have been supported and a whole desktop sharing would have been the only alternative;
- to give the presentation Ali and Peter would have had to perform one of the following: bring their laptop to connect to the projector; save the presentation on a removable disk; manually access some remote repository to access the previously stored presentation or use some Active Directory technology;
- a cumbersome procedure, similar to the one just described, would have been followed to perform the scrum of scrums and sprint meetings.

Chapter 4

Requirements

In this section, we discuss the high level requirements that have emerged from a survey of the literature. Most of the reviewed papers were selected from the primary studies identified while performing a large scale mapping study of the technologies used for GSD; we also selected some papers from outside of the pool of the mapping study papers. The identified requirements have been classified into two main categories: *system quality requirements* and *system requirements for computer-mediated teamwork*. This categorization does not entail requirements or functionalities that have been identified as important for supporting software engineering practices (e.g., knowledge management and risk management) as the purpose of this classification is to clearly differentiate the architecturally significant requirements (i.e., system quality requirements) from those related to the features considered important for supporting computer-medicated teamwork; both kinds of requirements are important for designing and developing an infrastructure that can effectively and efficiently support GSD teams.

4.1 System quality requirements

We discuss the high-level system quality requirements in the following paragraphs. These requirements are based on the data extracted from the reviewed paper; therefore, they may not capture all aspects of a system for supporting GSD teams. However, our purpose is to gather those quality requirements that have been reported as critical in most of the reviewed papers as such requirements should be important for designing technological support for GSD practitioners.

Many have pointed out the concept of **unity** as a central element to succeed when performing a task in a distributed environment. The authors of [9], while analyzing the consequences of geographical distribution for software quality, conclude that it is more important to be organizationally compact rather than geographically local. Furthermore, after analyzing nine GSD projects Herbsleb et al. conclude that one of the lessons learned was

the necessity to *"[...] set up a single 'virtual site' to the extent it is possible"* [22]. In [39], the authors report the need of a single software configuration management environment; Sinha et al. assert that distributed teams can greatly benefit from an environment where the support is guided by uniform mechanisms [44]. Finally, the presence of a conjoined system is identified as a core system property for a GSD environment in [35].

Based on our literature survey, we can conclude that **flexibility** should be a critically important design consideration. Maalej [32] reported that introducing new tools within an industrial context is a challenge; in [2], the authors suggest as a guideline the provision of full autonomy of sites in terms of tools and software process models. However, this degree of flexibility is recognized by many in a less rigid manner by arguing that it has to be provided at the application level. To provide highly customizable systems that support different ways of presenting and personalizing the content as well as to build tools for extensibility, interoperability and flexibility are some of the guidelines suggested in [10][27][12].

From our analysis of the literature, it is clear that there needs to be balance between flexibility and unity. It is going to be a challenge to design a technological support which is not only able to enforce uniformity all over the shared virtual space used by the GSD members to collaborate but also provides flexibility and adaptability required by different individuals and groups performing various activities and tasks of GSD.

Principle I of ABC requires a unifying concept underlying the metaphor: the activity. Therefore, by providing a system which is activity-centered would support a uniform way of interacting with the workstation. Moreover, flexibility is also a characteristic of ABC as it allows users to suspend/resume activities (principle: II) based on their convenience. The ABC principles also require activities to be able to adapt to the environment and the system in which they are resumed (principles: V - VI). Furthermore, our envisioned infrastructure is agnostic to any specific application or operative system and provides APIs to leverage diverse applications.

Additionally, a distributed system that may be used concurrently by many users needs to properly address **scalability**. Researchers have identified this requirement as a crucial quality attribute [38] and an important property of the system [35]. A system's performance should not degrade if it has to support an unpredictable, rapidly changeable number of connected users. In fact, in the GSD setting entire development groups starting their work shift may log into the system at the same time and vice versa.

By fulfilling principle I of ABC, scalability is introduced at two different levels. On the one side, it allows an end user to organize his/her work into well defined activities; on the other side, it introduces a clustering mechanism driven by the concept of activity that can take advantages of features provided by current technologies (e.g. cloud computing) by dividing, for example, computational needs in well known units identified by the activities.

Finally, a shared workspace has to be as neat as possible; that means it should not introduce an additional degree of complexity in the way the end users perform their tasks; the mechanisms introduced need to be seamless and intuitive [10]. In short, an important requirement for the system is **intuitivity**.

When introducing the ABC [37], Norman clearly argued that the main idea behind his vision was to release an end user from mental overheads; the ABC metaphor supports an interaction that allows an end users to resume work and alternate between tasks with one click in a potentially very intuitive way.

4.2 System requirements for computer-mediated teamwork

The requirements presented in this section differ from the aforementioned ones as these requirements are meant to capture the gap between co-located and distributed environments. The following requirements identify the types of features that are considered critically important for supporting teamwork performed by geographically separated members.

There are many challenges that are faced by geographically distributed members when working together such as time-zone difference and lack of physical presence. That is why enhancing **collaboration** is considered quite important. Collaborative support is a core requirement that should have a built-in support in a technology supporting GSD teams. A system needs to bring improvements to the shared virtual space providing support to the way GSD team members interact synchronously and asynchronously. According to the authors of [12], these collaborative features should: *"avoid embedding or enforcing a rigid notion of what the correct development process should be [...]; provide a flexible collection of ways that developers can collaborate [...]; be configurable and extensible by the developers themselves"*.

Two of the ABC principles (i.e., II and III) allow practitioners to move their work setting from their personal machine to another system, for example inside a meeting room, without having to move any physical device and restart the cooperative work with the remote team-members located across the globe. That means collaboration should be a basic feature provided by a GSD support system rather than a property that needs to be added through some external application.

Coordination represents another key requirement, it defines the support that is provided by a system to improve the way GSD team members can manage their work and themselves with respect to the others. Many people have identified **coordination** as an improvement area or a challenge to be addressed in a GSD environment (e.g. [25], [22]). Considering that projects are handled in different ways [18], instead of enforcing best practices the coordination support offered by a system should facilitate activities such as work product sharing [46], delegation of work to colleagues [22].

Having activities interconnections, from a managerial point of view, allows the decomposition of a project in sub-activities giving the ability to have a thorough vision of activities and assets allocated to them boosting coordination. Moreover, individual coordination would also be improved by introducing the central concept of activity (principle: I), letting the end users organize their work into activities.

In a distributed environments where face-to-face interactions are not possible, **communication** has always been identified as a very challenging aspect [20][45]. In [54], the lack of informal communication is considered one of the main reason for complications in the distributed context: requirement management, stakeholder negotiation, hand over of tasks are just some of the many areas in which, due to communication flaws, issues may quickly arise [10][28]. Therefore, a GSD support technology should provide both synchronous and asynchronous communication features without enforcing a unique way of communicating but allowing different mechanisms to realize it in a fast and intuitive manner.

The principle I of the ABC entails linking people to activities. That means knowing a priori which are the team members connected to an activity allows a smooth improvement in communication by, for example, enabling the automatic population of a contact list while switching between activities.

Dourish and Belloti [15] defined **awareness** as *"an understanding of the activities of others, which provides a context for your own activity"*. In a context where physical presence is denied, awareness is able to facilitate collaboration, communication and coordination [47]. The importance of this requirement can be seen in almost every piece of work involving a distributed context: from a simple notification message to a whole system providing monitoring functionalities (e.g. Augur [17]). However, there are not many work which identify awareness as a concrete requirement to be tackled throughout a system.

Unifying all information related to an activity (principle: I), allows a system to benefit from these centralized information. Therefore, we believe that the ABC paradigm can help make awareness a basic feature of a GSD support infrastructure.

Chapter 5

Solution - Design Part

In this section, we provide a detailed account of the theoretical concepts, architectural elements, and some of the implementation decisions underpinning our infrastructure for supporting the ABC paradigm in GSD. Following are the key elements of our infrastructure described in this section. The prototype built as a proof of concept for demonstrating the utility of our infrastructure will be described in Chapter 6.

- The current state of the concrete *data model* used to represent activities is described and all the rational behind its design explained; moreover, the design iterations that led us to the definition of the model current version are described in Appendix .
- the mechanism we introduced in the *persistency layer* that allowed us to perform rapid prototyping is presented;
- the high level *infrastructure* we designed to support the concepts of ABC is described in the third section; and,
- the event based *communication protocol* that enables the exchange of information throughout the whole system is presented in the last section.

insert correct appendix

5.1 Data model to support ABC

Collaboration and sharing are two of the paramount characteristics that need to be provided to support GSD [50], which have been proven to be effectively yield by some of the systems based on ABC (e.g., the ABC framework [7]). The ABC paradigm has been successfully implemented in different contexts (e.g., healthcare in [6]) for supporting users in different ways: from providing personal workspace organization (e.g., Gnome Shell[1]); to offering new desktop metaphors (e.g., Giornata [52]); and, new collaborative desktop environments (e.g., [5]). However, to support software engineering activities a WYSIWIS[2]-based

[1] http://live.gnome.org/GnomeShell
[2] WYSIWIS: What You See Is What I See.

approach can hardly be applied. The authors of [55] detected and categorized various different ways (namely patterns) in which knowledge workers using Lotus Activities were defining activities. Differently from processes and related workflows, activities have unclearer boundaries; thus, are harder to define. Nevertheless, they noticed that users, autonomously, after appropriating the system, created activities identified by a clear goal and with a well-defined endpoint; still, using the concept of activity to support their needs at various levels of abstraction. In the software engineering context, practitioners need to be able to independently manage applications and handle artifacts without necessarily having these being part of a common representation of activities shared by its participants. If not engaged in a concrete synchronous collaboration, users require activities representation to be personalized and able to fit their needs.

We believe that a system integrating the ABC principles, to be effective and used in the context of software engineering, must not deprive users from the freedom and ease of use proper of the currently used file- and application- based paradigm. The data model supporting *ABC4GSD*, tailoring these preliminary analysis of the context, is depicted in the UML diagram presented in Figure 5.1 and detailed in the following.

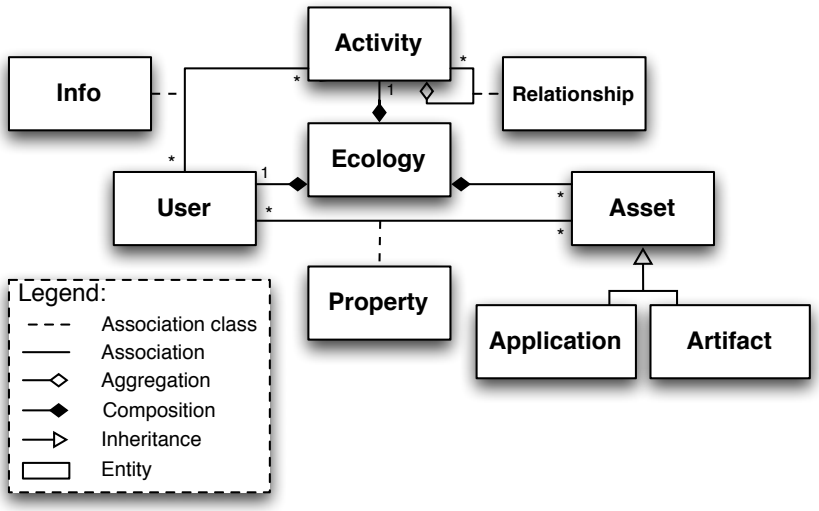


Figure 5.1: Data model

The central element of the model is the *Ecology* entity that, by indirectly linking activities to users, keeps track of all assets associated to an activity by each user. From the diagram we can see how every ecology links one activity to one of its participants and to all assets used (i.e., applications and artifacts). Through the Ecology entity, the necessary flexibility previously described is obtained.

The *Activity* entity captures the core concept of the ABC paradigm and contains information shared by all participants like its name and state (i.e., initialized, terminated, ongoing).

To properly provide a structured way of managing projects and supporting coordination, activities interrelations are an important aspect that the infrastructure needs to facilitate. The *Relationship* element has been introduced to model activities connections and represent

the extension point for future expansions towards this direction.

Participants are modeled through the *User* entity, by collecting all details needed to identify a physical person interacting with the system. An association class between User and Activity is provided to capture all information necessary to specify for example the individual role or status of each participant in each activity.

Asset is the superclass mapping digital artifacts and applications common information. Properties and unique information regarding assets connected to users are encoded in the *Property* association class, further supporting the flexibility initially discussed.

The *Artifact* entity collects all digital artifacts used in activities (e.g., UML models, source code, etc.). Whereas, the *Application* element, stores the 'need' that applications have to support. To give an example, an application that needs to handle pdf artifacts would be represented by an identification string (e.g., 'pdfViewer'), which at every client-side can be flexibly bound to a specific application (e.g., Skim for workstations using MacOS or Foxit Reader for those running Windows systems). Therefore, every 'need' is mapped at the client-side via a configuration file.

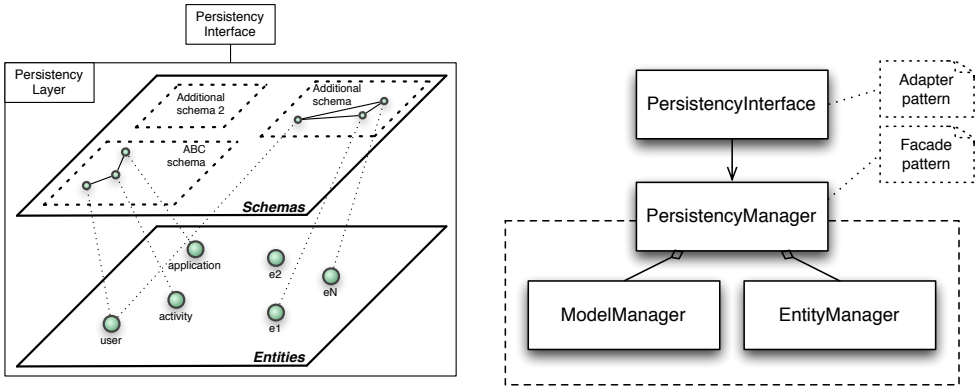
Finally, it has to be noticed that, by using inheritance, a behavior similar to the proxy pattern is introduced at the database level. In fact, to map this model in a relational database one of the solution –the most similar to object oriented design– would entail a unique primary key for the Asset table, which would differ from the one used for either the Application or the Artifact. This allows the same asset to be linked by different activities -through ecologies- and users yet maintaining unique properties. Application and Artifact elements are thus implicitly wrapped by elements of the *Asset* entity. On the other hand, from a code prospective this behavior would need to be explicitly implemented by using for example a proxy pattern.

5.2 Persistency layer

During the design of the server side of the infrastructure we decided to provide a flexible solution; a solution in which the server would have been as independent as possible from the concrete data model. During the prototyping activities we underwent, different data models to support the ABC paradigm have been experimented and server flexibility was a necessity. A solution in which the data model was hardcoded both in the server and the client side was too rigid for supporting the often changing data model. To overcome this issue we decided to devise a more sophisticated design where the information persistency, still handled by the remote server, would have been independent from the ABC concepts. Still maintaining the idea of having a persistent representation of the data, two distinct levels were provided for the entities on the one hand and for the relationships on the other. Moreover, an additional layer wrapping them was designed to transparently handle this separation. A graphical representation of the persistency layer is provided in Figure 5.2 and in the following described.

Schemas are the bounding components of this mechanism; they can be uploaded at runtime by the clients and, upon a validation performed by the remote server, referenced by any other client. Each schema contains the following information:

- the list of entities composing the schema;
- the list of attributes each entity contains and their type;
- the list of relations of each entity and their nature (e.g., one to many, etc).



(a) (b)

Figure 5.2: Persistency layer.

Entities and their attributes are stored in the lowest layer; whereas, at the schema level, relationships between entities are handled and enforced. Therefore, data models are inserted as schemas.

Given this design of the persistency layer, the automatic generation of new data models is supported as well as the possibility to easily modify the existing ones. Clients can opt whether to connect to existing schemas getting advantages from the information shared within them or to define their own new schemas enabling particular behaviors to support specific needs (e.g., synchronous communication). Finally, no restriction is imposed on the number of schemas a client can connect to.

think of a better example

Furthermore, by using this approach there are no constrains on the type of persistency mechanism used: relational databases or noSQL ones could be used as well as hybrid solutions, where for example the Ecology and the Property entities could be identified by a concatenation of ids referenced (e.g., *activityId:userId:assetId* for the ecologies and *userId:assetId* for the properties). This design decision was also made to support future improvements aimed at addressing performance issues that the data layer might have to ensure as the potential load of requests and the amount of generated information can rapidly increase; thus, we felt the need of addressing this challenge at an early stage of the design.

Finally, the *Persistency Interface* unites the two layers exposing proper APIs to interact with the *Persistency Layer*. The language used to query the data model has been designed ad-hoc and will be hereafter briefly introduced with the subscription mechanism. The template of all queries follow this structure:

```
1 <schema>.<entity>.<ent_id>.<field>.<operation>.<value>
```

Therefore, a message like the following,

```
1 abc.user.164225212.state.=.11103
```

would be sent to the remote server to inform it about the change of a user state. The massage starts with the declaration of the schema to be used followed by the entity name, the entity id, the entity field, the operation type and the value to be assigned; it literally means that in the schema named 'abc' the state of the user identified by the id 164225212 will be changed to 11103 (i.e., disconnected). After performing the requested query, the remote server checks for all the subscription received and, if anyone matches, the exact

message is sent to the subscriber. Through this mechanism it is possible to allow the automatic adaptation of applications in response to events propagated by the remote server. The middleware flexibility, in fact, does not constrain the developer by any means and, in Chapter 6, we will show how an IDE like Eclipse can be easily enhanced by leveraging the middleware with small coding effort.

5.3 High level architecture

The architecture of the *ABC4GSD* middleware is based on the client-server architectural style that, at the client-side, contains an additional component to increase the control over communications by making the stream of messages converge to a single element also on the client-side (i.e., the middleware). The main components and communication channels of the architecture are depicted in Figure 5.3 and hereafter described.

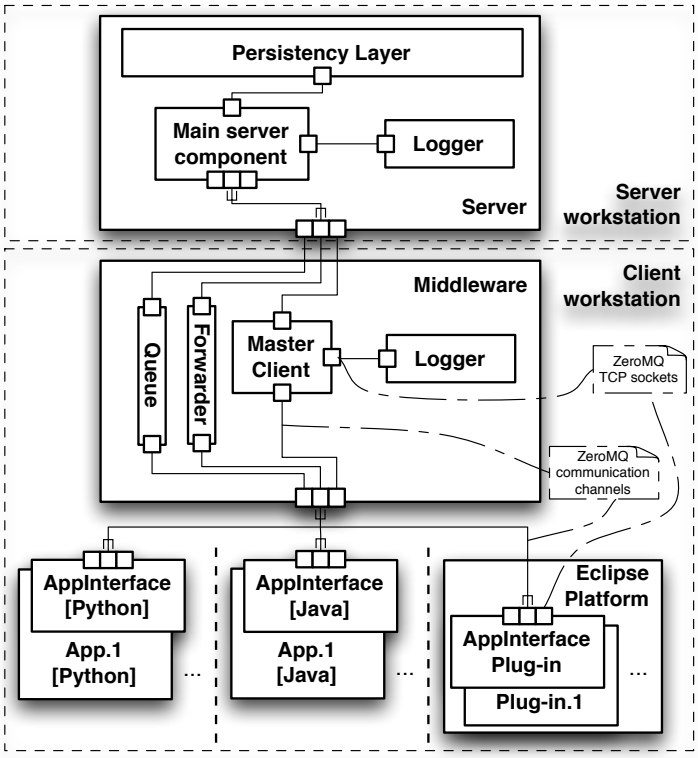


Figure 5.3: High level architecture components.

The *Persistency Layer*, whose design was covered in section 5.2), is the component responsible for ensuring the persistency and consistency of the data; it is accountable for:

- managing schemas;
- validating requests against the schemas stored to control if well formed and adhering to the schemas;

- evaluating requests and generating responses for senders;
- identifying matching subscriptions.

This component has been made completely independent and detached from the others to increase modifiability of the server-side and to allow further extensions for better supporting availability and scalability of the overall system. A logging component has also been included for control purposes in the case of malfunctions.

The *Main server component* is the element communicating with the client-side of the middleware; it is responsible for generating the necessary events to ensure the correct communication of the information throughout the client machines. The main operations that are performed when a message is received are: check and authenticate the sender; forward requests to the Persistency Layer; send responses; and, propagate information to potential subscribers.

Together with the *Forwarder*, the *Queue* element acts as a broker between the remote server and the client applications by dispatching messages generated locally and directed to the remote server and vice versa. They are responsible for handling the most intense communication streams: the publishing of events by the server (Queue); and, the requests client applications direct to the server (Forwarder). The *Master Client* is the main component in each client workstation, its purpose is to handle authentication and enforce the basic logic to support the ABC concepts of suspension and resumption. These three elements compose the main middleware through which client applications communicate with the server and vice versa acting as a middleman. Through this design we are able to monitor and control all the communication generated by each client.

The purpose of the *AppInterface* is to provide a common way to interact between the system and the applications linked to the activities; the client applications (see Chapter 6) have to adhere to this specific interface to benefit from the middleware. This component, besides providing the functionalities to send/receive messages and subscribe/unsubscribe to events, defines a list of ad-hoc methods that have to be implemented for having both the *Client Master* control some behavior of each application and, the applications, react to messages coming from the *Client Master* and *Queue* component. Following these methods are enlisted and explained:

- **killOperation** allows the application to perform operations before being terminated by the Client Master;
- **suspendOperation** allows the application to perform operations before being suspended. Through this hook all the needed information to restore itself can be sent to the remote server;
- **resumeOperation** allows the application to perform operations before being resumed. Symmetrical to the suspendOperation one, this method is the hook to allow the application to restore itself consistently by retrieving information from the server;
- **personalHandler**: the hook to intercept the system messages and react as needed. The communication handled with this method does not include the response to requests. It is limited to local messages from the Client Master only, used for example to send to applications the suspend and resume signals; and, to the events published by the remote server via the Queue component.

We can see from the diagram that three versions of the AppInterface are provided depending on the type of application that wants to leverage the functionalities provided by

the middleware. Starting from the the left: a python version, a java version and one that has been developed as an Eclipse plug-in. This last version is integrated in the Eclipse platform and exposes an extension point for other plug-ins to be extended. Finally, also at the client-side a *Logger* has been introduced for controlling the middleware activity.

Through this architecture, all additional client components are treated as independent application, i.e., concrete external processes. This design decision has been taken to maximize the decoupling of each component allowing the just in time (JIT) insertion or substitution of any element with any other one.

The communication infrastructure between both client and server elements has been implemented using ZeroMQ[3]: a messaging library, which allows to design a complex communication system through intuitive APIs also providing messaging patterns like publish-subscribe or request-reply. Each connection line of Figure 5.3 identifies a ZeroMQ communication channel (lines ending with a fork are used to simplify the diagram and should be read as three independent channels); and, each square, a ZeroMQ TCP socket. As we can see, each communication type (i.e., event publishing, individual requests, middleware communication) has been kept isolated and a dedicated channel has been provided.

5.4 Communication mechanism

This section describes the element of the infrastructure that permits the flow of information described in the previous section: the communication mechanism. Through the infrastructure described above, messages are generated and dispatched to clients in order to enable a dense communication with each other. The protocol is based on a publish/subscribe architectural style where, brokered by the Activity Manager, each of the client applications independently notifies to the remote server which are the changes to be performed on the remote data layer as well as which events they are to be informed about.

Applications, accordingly to validated schemas, can generate messages to either modify entities fields or add/remove relationships between entities. Messages are sent by each application through the Activity Manager to the remote server which checks their consistency and propagate them to any client if subscriptions to the specific event are matched. The design of the protocol has been kept as simple as possible in order to be able to quickly test it and easily modify or upgrade it. Following a practical example will describe its basic functioning.

Let's consider a scenario involving multiple clients using an application named *Contact List* in order to track and interact with users belonging to an activity; for this purpose, users are identified by a trivial entity containing only the id of the user, its name and status.

When the Activity Manager executes the Contact List, the method resumeOperation described in section 5.3 is called. Figure 5.4 shows a code snippet of a possible implementation of the method where it can be seen how the provided interface allows an application to easily subscribe to specific events (the insertion of a new user in the current activity in line 3, the changing of the name in line 6 and the modification of the state in line 8) as well as associate a method to be called when such events are received from the remote server (line 4, 7 and 9).

Further, during the login operation of a client, a message similar to the following

```
1 abc.user.164225212.state.=.11102
```

[3] http://www.zeromq.org

```
1 def resumeOperation(self):
2     ...
3     self.subscribe('abc.activity.%s.user.+' % \
4                    (self._actId, ), self.update)
5     for id in self.getUsers():
6         self.subscribe('abc.user.%s.name' % (id,), \
7                        self.update)
8         self.subscribe('abc.user.%s.state' % (id,), \
9                        self.update)
```

Figure 5.4: ContactList::resumeOperation()

would be sent to the remote server to inform it about the change of a user state. The massage starts with the declaration of the schema to be used followed by the entity name, the entity id, the entity field, the operation type and the value to be assigned; it literally means that in the schema named 'abc' the state of the user identified by the id 164225212 will be changed to 11102 (i.e. connected). After performing the requested query, the remote server checks for all the subscription received and, if anyone matches, the exact message is sent to the subscriber. Finally, in the case of the Contact List application described above, the method *update* (Figure 5.4 - line 9) would be called passing the message as parameter.

Therefore, through this mechanism it is possible to allow the automatic adaptation of applications in response to events propagated by the remote server. The infrastructure flexibility in fact does not constrain the developer by any means and in the next section we will show how a graphical user interface (GUI) using this infrastructure can be easily build on top of it from scratch with small coding effort. A graphical interface able to improve awareness for collaborators working in a common activity by leveraging the shared information handled and propagated by the system.

Chapter 6

Solution - Applications

Bibliography

[1] M. Ali Babar. A framework for groupware-supported software architecture evaluation process in global software development. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.

[2] L. Aversano, A. De Lucia, M. Gaeta, P. Ritrovato, S. Stefanucci, and M. Luisa Villani. Managing coordination and cooperation in distributed software processes: the GENESIS environment. *Software Process: Improvement and Practice*, 2004.

[3] M. Babar and J. Verner. Groupware Requirements for Supporting Software Architecture Evaluation Process. In *International Workshop on Distributed Software Development*, 2005.

[4] M. A. Babar, B. Kitchenham, and R. Jeffery. Comparing distributed and face-to-face meetings for software architecture evaluation: A controlled experiment. *Empirical Softw. Engg.*, 2008.

[5] J. Bardram, J. Bunde-Pedersen, and M. Soegaard. Support for activity-based computing in a personal computing operating system. *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2006.

[6] J. Bardram and H. Christensen. Real-time collaboration in activity-based architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, 2004.

[7] J. E. Bardram. Activity-based computing for medical work in hospitals. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2009.

[8] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. FASTDash: a visual dashboard for fostering awareness in software teams. *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2007.

[9] C. Bird, N. Nagappan, P. Devanbu, and H. Gall. Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista. *Communications of the ACM*, 2009.

[10] G. Booch and A. Brown. Collaborative development environments. *Advances in Computers*, 2003.

[11] D. Budgen, A. Burn, O. Brereton, B. Kitchenham, and R. Pretorius. Empirical evidence about the UML: a systematic literature review. *Software: Practice and Experience*, 2010.

[12] L.-T. Cheng, C. R. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building Collaboration into IDEs. *Queue*, 2003.

[13] D. Damian and D. Moitra. Guest Editors' Introduction: Global Software Development: How Far Have We Come? *IEEE Software*, 2006.

[14] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Fine-grained management of software artefacts: the ADAMS system. *Software: Practice and Experience*, 2010.

[15] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. *CSCW '92 Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, 1992.

[16] C. A. Ellis, S. J. Gibbs, and G. Rein. Groupware: some issues and experiences. *Communications of the ACM*, 1991.

[17] J. Froehlich and P. Dourish. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004.

[18] R. Grinter, J. Herbsleb, and D. Perry. The geography of coordination: dealing with distance in R&D work. *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, 1999.

[19] J. Herbsleb. Global software engineering: The future of socio-technical coordination. *2007 Future of Software Engineering*, 2007.

[20] J. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *Software Engineering, IEEE Transactions on*, 2003.

[21] J. Herbsleb and D. Moitra. Global software development. *IEEE Software*, 2001.

[22] J. D. Herbsleb, D. J. Paulish, and M. Bass. Global software development at siemens: experience from nine projects. In *Proceedings*, 2005.

[23] E. Hossain, M. A. Babar, H.-y. Paik, and J. Verner. Risk Identification and Mitigation Processes for Using Scrum in Global Software Development: A Conceptual Framework. In *Proceedings*, 2009.

[24] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Introducing collaboration into an application development environment. *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, 2004.

[25] M. Jiménez, M. Piattini, and A. Vizcaíno. Challenges and improvements in distributed software development: a systematic review. *Advances in Software Engineering*, 2009.

[26] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006.

[27] H. Kienle and H. Muller. Requirements of Software Visualization Tools: A Literature Survey. *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, 2007.

[28] R. E. Kraut, D. Gergle, and S. R. Fussell. The use of visual information in shared visual spaces: informing the development of virtual co-presence. In *Proceedings*, 2002.

[29] M. Lang and J. Duggan. A Tool to Support Collaborative Software Requirements Management. *Requirements Engineering*, 2001.

[30] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno. Collaboration Tools for Global Software Engineering. *IEEE Software*, 2010.

[31] F. Lanubile, T. Mallardo, and F. Calefato. Tool support for geographically dispersed inspection teams. *Software Process: Improvement and Practice*, 2003.

[32] W. Maalej. Task-First or Context-First? Tool Integration Revisited. *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, 2009.

[33] M. Mangan, M. Borges, and C. Werner. A middleware to increase awareness in distributed software development workspaces. *Proceedings of the WebMedia & LA-Web 2004 Joint Conference 10th Brazilian Symposium on Multimedia and the Web 2nd Latin American Web Congress*, 2004.

[34] R. Martignoni. Global Sourcing of Software Development - A Review of Tools and Services. *Fourth IEEE International Conference on Global Software Engineering, 2009. ICGSE 2009.*, 2009.

[35] P. Mukherjee, A. Kovacevic, M. Benz, and A. Schürr. Towards a Peer-to-Peer Based Global Software Development Environment. In *Software Engineering*, 2008.

[36] J. Noll, S. Beecham, and I. Richardson. Global software development and collaboration: barriers and solutions. *ACM Inroads*, 2010.

[37] D. Norman. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution.* 1999.

[38] R. Palacio, A. Moran, V. Gonzalez, and A. Vizcaino. Collaborative Working Spheres as support for starting collaboration in distributed software development. *computer.org*, 2009.

[39] L. Pilatti, J. L. N. Audy, and R. Prikladnicki. Software configuration management over a global software development environment: lessons learned from a case study. In *Proceedings of the 2006 international workshop on Global software development for the practitioner*, 2006.

[40] M. Purvis, M. Purvis, and B. Savarimuthu. Facilitating collaboration in a distributed software development environment using P2P architecture. *5th International Workshop on Agents and Peer-to-Peer Computing, AP2PC 2006, May 9, 2006 - May 9, 2006*, 2008.

[41] J. Rodriguez, C. Ebert, and A. Vizcaino. Technologies and Tools for Distributed Teams. *Software, IEEE*, 2010.

[42] A. Sarma, D. Redmiles, and A. v. d. Hoek. Categorizing the Spectrum of Coordination Technology. *Computer*, 2010.

[43] K. Schwaber. *Agile Project Management with Scrum*. 2004.

[44] B. Sengupta. Enabling Collaboration in Distributed Requirements Management. *Software, IEEE*, 2006.

[45] B. Sengupta, S. Chandra, and V. Sinha. A research agenda for distributed software development. *Proceedings of the 28th international conference on Software engineering*, 2006.

[46] H. Spanjers, M. ter Huurne, B. Graaf, M. Lormans, D. Bendas, and R. van Solingen. Tool Support for Distributed Software Engineering. *Global Software Engineering, 2006. ICGSE '06. International Conference on*, 2006.

[47] I. Steinmacher, A. Chaves, and M. Gerosa. Awareness Support in Global Software Development: A Systematic Review Based on the 3C Collaboration Model. In *Collaboration and Technology*. 2010.

[48] J. Suzuki and Y. Yamamoto. Leveraging distributed software development. *Computer*, 1999.

[49] P. Tell and M. Ali Babar. Supporting Activity Based Computing Paradigm in Global Software Development. *Automated Software Engineering, 2011*, 2011.

[50] P. Tell and M. A. Babar. Requirements for an Infrastructure to Support Activity-Based Computing in Global Software Development. In *Global Software Engineering Workshop (ICGSEW), 2011 Sixth IEEE International Conference on*, 2011.

[51] V. Trapa and S. Rao. T3 - tool for monitoring agile development. *Agile Conference, 2006*, 2006.

[52] S. Voida, E. D. Mynatt, and W. K. Edwards. Re-framing the desktop interface around the activities of knowledge work. *Proceedings of the 21st annual ACM symposium on User interface software and technology*, 2008.

[53] J. Whitehead. Collaboration in software engineering: A roadmap. *2007 Future of Software Engineering*, 2007.

[54] S. Whittaker, D. Frohlich, and O. Daly-Jones. Informal workplace communication: what is it like and how might we support it? *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, 1994.

[55] S. Yarosh, T. Matthews, T. P. Moran, and B. Smith. What Is an Activity? Appropriating an Activity-Centric System. In *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction: Part II*, 2009.

.1 Design considerations

In the following section critical aspects of the infrastructure will be analyzed and design considerations given and explained. These aspects represent improvements and enhancement elements that are considered to be area that have to be tackled in the near future.

.1.1 Communication protocol

Figure 1 shows the current communication architecture. As it can be seen, a double channel is created for every couple of components, which will be hereafter described. Channels are created by the element containing the label (i.e. port number or generic identifier) on its side.

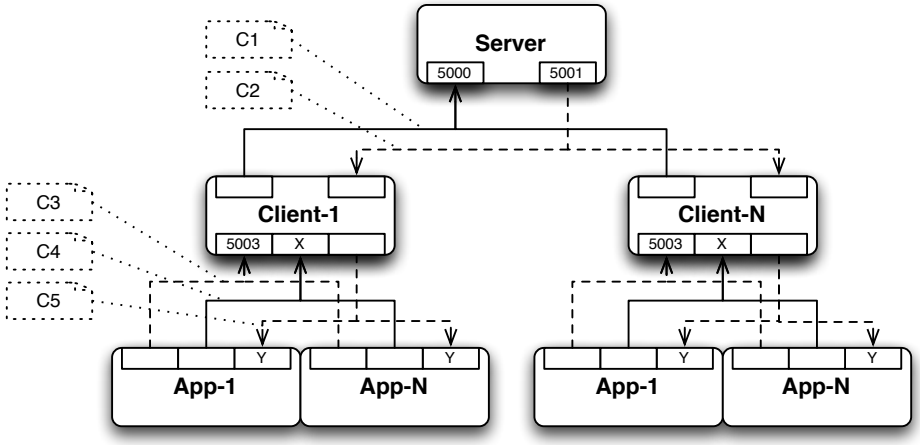


Figure 1: Communication Overview

The communications between the remote server and the local servers are both bound by the former and are defined a priori. This is done to allow the remote server to be hosted in secure networks like DMZ[1] where internal machine cannot connect to the outside. *C1* is the first socket created by the remote server, the entry point to the system. After establishing the connection, the channel is used by the clients to send messages to the remote server; whereas, *C2* is established by the server to propagate the information from the server to the clients. The former deals with the type of messages hereafter described. All messages are preceded by `CODE <long> FROM <client_id>`, where code is a unique identifier and client_id is a long identifying each client.

- `CONNECT <model> USER <name>`: used to connect to a model;
- `DISCONNECT <model>`: used to disconnect from a model;
- `STORE <filename>`: temporarily implemented to store the server state;
- `RESTORE <filename>`: temporarily implemented to restore the server state;
- `QUERY <model> <query>`: user to query a specific model. It's syntax will be covered in the following;
- `RUN <query>`: equivalent to the previous one, this command has been mainly implemented for debug purposes as it does not require to be connected to specific models.

[1] DMZ: demilitarized zone

As previously stated, the channel marked *C2* is used by the server to communicate to each client all the modifications that occurs. The messages sent through this channel contain the exact copy of the query that the server received.

Channels created to handle local communications between client and single applications are created both by the client element as well as by the *ABCAppInterface* element. The channel marked *C3* is used as entry point for establishing the remaining connections.

C4 is established by the client and is used by the applications to messages described in the following.

- `RESUME <activity_id>`: used to resume an activity;
- `SUSPEND <model>`: used to suspend an activity;
- `QUERY <query>`: used to send specific requests to the server via the client;
- `ABC RESUME COMPLETED`: used to synchronize the resumption of an activity;
- `ABC SUSPEND COMPLETED`: used to synchronize the suspension of an activity.

The channel marked *C5* is initialized by the single applications and is used to send messages from the client to the single applications. These are the messages handled:

- `<data_entity> CMD INIT`: sent to force the initialization of applications interested to specific data;
- `<data_entity> CMD SUSPEND`: sent to force the suspension of applications interested to specific data. Normally used by sending *all* as data_entity to suspend the whole activity;
- `UPDATE <query>`: used to propagate the updates coming from the remote server to the single applications.

The current implementation is obtained using basic TCP sockets, the communication protocol might benefit from the introduction of a more sophisticated and well established message queue system. In a concrete GSD scenario, the number of exchanged messages between the remote server and the local one can significantly increase. Currently, the most commonly known solutions are being reviewed: MSMQ, ActiveMQ, RabbitMQ and ZeroMQ; and, from benchmarks and descriptions ZeroMQ[2] seems to be the most appropriate considering its flexibility and performances.

.1.2 0mq

As suggested in the previous section, 0mq has been integrated to enhance the overall communication protocol by leveraging a well known and very actively developed messaging queue (MQ). Compared to traditional MQs that require a running daemon, 0mq is a library based system. The system provides, through the mean of a library, a framework built on top of standard sockets. Some designers of 0mq, when asked to introduce the system, describe it by saying that 0mq is simply 'sockets on steroids'. In fact the whole system was designed to support financial systems where a huge amount of data need to be handled and their design decisions were driven by the concept of making first of all a fast system and only secondly a reliable one. In terms of throughput, 0mq has no competitors and is has the

[2] http://www.zeromq.org

previously introduced advantage of being a library and not a running service. Further, the library is designed to ease development effort. 0mq provides support by implementing well established communication patterns that can be either directly used as they are or tweaked upon needs. Examples of the patterns 0mq provides are: publish/subscribe, request/reply, push/pull, etc..

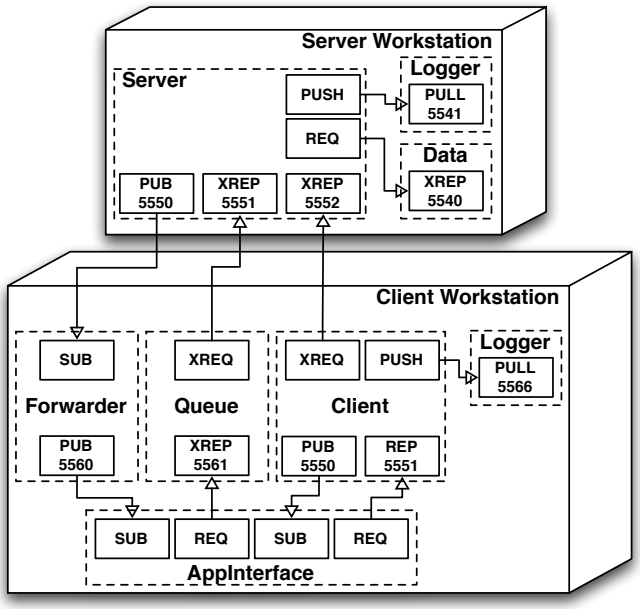


Figure 2: Communication overview after the integration of 0mq

Therefore, after deciding to integrate the library, redesigning the architecture to better leverage 0mq has been a natural design consideration. As we can see from Figure 2, the overall architecture has been made more modular and components less interdependent.

On the server side, we can see that two components have been isolated from the main application: the *Logger*, trivially providing logging functionalities; and, the *Data* component, in charge of the persistency layer. Thus, in the new implementation, the main server application is acting as a client in relation to the persistency layer; separation that would have been appropriate in the future if moving to a different paradigm (Section .1.4).

The client side has also been deeply modified. The main client application is now only responsible for making the whole system coherent; data requests as well as event notifications are detached from the main component and information handled by them flow independently from the application interfaces to the server and vice versa through the *Queue* and the *Forwarder* component respectively. Similarly to what has been done with regards to the server architecture, also the client one includes a separated local *Logger* component for carrying out logging functionalities.

.1.3 Data layer

The persistency layer, introduced in Section **??**, has been designed and implemented in order to provide a strong flexibility without imposing any design decision related to the

concrete choice of the storage system; as previously indicated, the separation between the entity layer on the one hand and the schema layer on the other encourages the use of non relational databases like the noSQL one. These databases are getting more and more appreciation for their capability of managing enormous amount of data without degrading performance. Therefore, noSQL databases like Cassandra may provide an interesting enhancement in terms of performance due to the structure of the data and the number of queries generated by the system in a concrete setting.

.1.4 Client-server architecture

The nature of the presented infrastructure has to be centralized. Nonetheless, the conventional client/server architecture might be easily replaced with more modern approaches like the service oriented or cloud computing one. In fact, the remote server represents a perfect candidate to be migrated to the cloud (depicted in Figure 3) in order to leverage the well known benefits that the cloud infrastructure can offer.

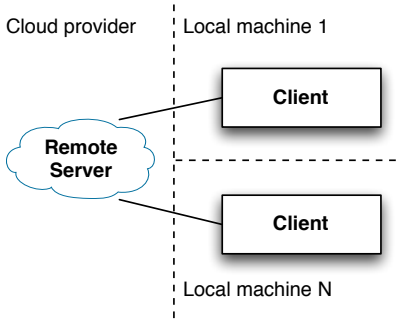


Figure 3: Possible elements distribution

.1.5 Applications

The current proof of concept is implemented through the use of ad-hoc applications developed to leverage the infrastructure, which cannot be comparable to any available tool as their purpose was limited to demonstrate the effectiveness of the infrastructure. Therefore, there is the need of the introduction of the *ABCAppInterface* into applications more used in the software engineering context. A plugin for an IDE like Eclipse is the first step envisioned towards a testable software engineering environment. Eclipse, in fact, offers APIs for that would allow an easy introduction of the interface; moreover, the leveraging of Eclipse workspaces seems to be a right approach to collect the needed information.

.1.6 Data model

The data model that has been described in Section **??**(IV-A) was sufficient for the purpose of the prototype. However, these considerations have been made while analyzing the outcomes:

- to support the roaming and sharing principles, the model is too rigid. An approach where, instead of storing the information about a concrete application, the infras-

tructure will save information related to the *need* that a potential application should support. Through this approach, there would be no constrain imposed on practitioners in terms of applications to be used for a specific activity; however, in this way the information stored would not necessarily be applicable to both softwares;

- thus, to support the previous point, every single workstation should contain a manifest-like file where *needs* are linked to specific applications that are able to support them;
- users part of an activity cannot have their own applications running inside the activity. The model does not support an application to be associated with a single/subset of users;
- the group entity does not provide any additional feature at the data level; thus, the composite pattern that it represent will be provided at the application level.

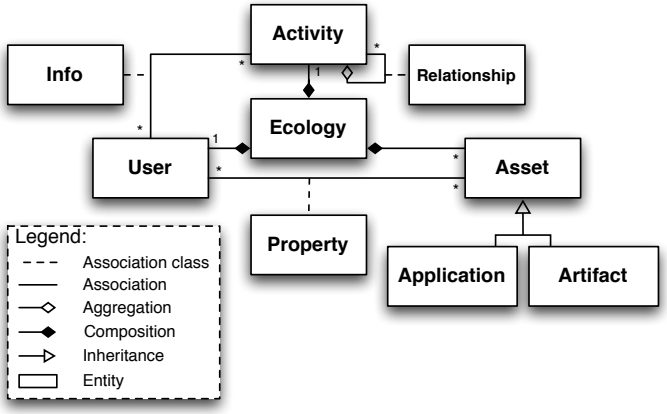


Figure 4: Possible new data model

Figure 4 shows the current idea that is being analyzed to support a more flexible model able to provide to activity participants the possibility to have personal applications and artifacts independently from other participants. This feature is of great importance in the context of software development compared to the healthcare environment where the applications ecology can be imposed and will be equal for every participant in the activity.

Therefore, the data model has been rearranged to be more user-centric. *Activity* is still the main entity; however, it is now connected to the rest of the model (except for *Relation*) through the *Ecology* entity, which collects the connections between users and assets (i.e. applications and artifacts) to each activity. Moreover, properties and unique information regarding assets connected to users are now encoded in the *Property* entity. Previously, for the applications only, such data where stored in the AppState entity. Now also artifacts can benefit from this by, for example, storing information related to each user on what is the preferred application to support it. Finally, the proxy pattern has been introduced to support the possibility of having the same asset having unique properties in different activities. Concrete assets are wrapped and identified by unique ids to permit their association with different properties.

.1.7 Query language

The language that is used to query the data model has been designed ad-hoc and will be hereafter described. However, improvements can be performed on the syntax mostly to support failure scenarios where illegal queries are sent.

Generally speaking, all the queries follow this structure:

```
1 <schema>.<entity>.<entity_id>.<field>.<operation>.<value>
```

Therefore, a message like the following,

```
1 abc.user.164225212.state.=.11102
```

would be sent to the remote server to inform it about the change of a user state. The massage starts with the declaration of the schema to be used followed by the entity name, the entity id, the entity field, the operation type and the value to be assigned; it literally means that in the schema named 'abc' the state of the user identified by the id 164225212 will be changed to 11102 (i.e. connected).

Moreover, the language supports also the generalization of the ids and the concatenation of commands. Therefore, a message like the following,

```
1 abc.artifact.[abc.artifact.[].state.==.<WIP>].location.=.{{./}}.&. \\
2 abc.application.[abc.application.[].state.==.11021].artifact.+. \\
3 [abc.artifact.[].state.==.<WIP>]
```

contains two commands concatenated by '*.&.*'; the former, means that the field *location* of all the artifacts having the *state* field equal to $<WIP>$ will be assigned to '*./*' ('{{' and '}}' are used to wrap values); the latter, adding to all the applications whose *state* field is 11021, artifacts having the *state* field equal to $<WIP>$.

Finally, the following commands are allowed:

```
1
2 +.<file_name>
3 -.<schema_name>
4 <schema_name>
5 <schema_name>.<entity>
6 <schema_name>.<entity>.+
7 <schema_name>.<entity>.-.<entity_id>
8 ...
9 #+<query>
10 #-<query>
```

Line 1 returns the lists of schemas present in the data layer; line 2 adds a new schema; line 3 removes a schema. Line 4 returns the list of entities; line 5 the list of entity ids; line 6 returns the id of the new entity and line 7 is used to remove the entity identified by the provided id. Line 9 and 10 are used to subscribe and unsubscribe to specific queries.