SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:1293–1311
Published online 18 August 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.937

Using frameworks to develop a distributed conferencing system: an experience report

Fabio Calefato and Filippo Lanubile*, †

Dipartimento di Informatica, University of Bari, Bari, Italy

SUMMARY

Application frameworks are a powerful means to reduce software development costs while improving quality. However, at the same time they are difficult to select and understand, as well as hard to learn, use, and debug effectively and efficiently. In this paper we report the story of eConference, a distributed conferencing system that was developed as part of a broader research effort. Here we discuss the lessons learned from the evolution of our conferencing tool over four generations, which have been necessary to find good frameworks and build a flexible distributed tool. Copyright © 2009 John Wiley & Sons, Ltd.

Received 16 October 2008; Revised 29 June 2009; Accepted 9 July 2009

KEY WORDS: software evolution; software reuse; application framework; communication framework; collaborative system

1. INTRODUCTION

In the last decade, being a good programmer has shifted more and more from only mastering programming languages to mastering application frameworks as well. Good programmers must able to create complex architectures by first selecting appropriate application frameworks, or building new frameworks on their own if no other matches the desired criteria, and then integrating them properly.

Application frameworks are usually built using object-oriented programming languages [1] and defined as a set of cooperating classes that make up a reusable design for a specific software

*Correspondence to: Filippo Lanubile, Dipartimento di Informatica, University of Bari, Bari, Italy.
†E-mail: lanubile@di.uniba.it

Contract/grant sponsor: Eclipse Innovation Award
Contract/grant sponsor: MiUR-Italy

[2,3]. Here, by using the expression application frameworks (or simply frameworks) we mean both closed-source Commercial Off-The-Shelf (COTS) and free open source software (FOSS) components. Open source products are usually considered to be completely different from closed source, commercial ones. However, in practice, they are both treated as closed source in the same way because, although the source code is available for FOSS frameworks, adopters seldom look at it [4].

Developers realize the full benefits of leveraging application frameworks only after several iterations. Indeed, although frameworks are a powerful means to reduce the software development costs while improving the quality, they are at the same time difficult to select and understand, as well as hard to learn, use, and integrate effectively and efficiently [5,6].

Although there is no commonly accepted method for frameworks selection [7,8], all the methods share some key steps, such as searching for candidates based on some established evaluation criteria or conducting pilot studies that apply those selected to develop representative prototype applications. However, the suitability of a framework for a particular application may not be apparent until the learning curve has flattened, which often occurs on successive projects that use the framework, since application developers can take months to become highly productive with frameworks on their own [9]. In addition, application development is often based on multiple frameworks that have to be integrated with one another. The integration, however, can lead to serious problems, since frameworks are generally designed for extension and not for integration [10].

In this paper we describe the evolution of eConference [11], a text-based, distributed conferencing system that has been developed as a part of a broader research effort with the aim to support the interaction of *ad hoc*, goal-oriented workgroups, which need low-cost administration infrastructure just to complete the task at hand [12,13].

When developing distributed conferencing systems, complexity and usability are the major problems [14]. People need to focus on the content of their meeting, not on the meeting tool itself, and thus, features have to be chosen carefully to maximize the tool effectiveness while minimizing the complexity. However, the scope of a distributed conferencing system goes beyond the supporting users' activities during the meeting itself, focusing also on to facilitating the arrangement and setup of meetings. An important challenge that we faced while building our distributed conferencing system was keeping the administration cost at a minimum, thus making the tool not only easy to use but also to set up and maintain.

Here, we discuss the lessons learned from the evolution of the eConference tool over four generations, which have been necessary to find good frameworks and build a flexible distributed conferencing tool. Through the years, we first changed the underlying communication framework from the JXTA P2P platform (ver. 1) to the eXtensible Messaging and Presence Protocol (XMPP) client/server protocol (ver. 2), which has proved to be a more robust and reliable solution to develop an extensible tool for distributed meetings. Then, eConference evolved from a conferencing system to a pure-plugin collaborative framework, built on top of the Eclipse rich client platform (RCP) (ver. 3). Finally, in the latest version, eConference has reached communication protocol independence through the Eclipse Communication Framework (ECF) (ver. 4).

The remainder of this paper is structured as follows. In Section 2 we briefly describe the eConference tool and its core features. In Section 3 we present each of the four versions of eConference, motivating the design choices and discussing the problems encountered during its development. In Sections 4 and 5 we discuss the open issues and the lessons learned. Finally, in Section 6 conclusions are drawn.

DOI: 10.1002/spe

2. TOOL DESCRIPTION

eConference is a text-based distributed meeting system. The primary functionality provided by the tool is a closed group chat, augmented with agenda, meeting minutes editing, and typing awareness capabilities. Around this basic functionality, other features have been built to help organizers to control the discussion during distributed meetings. Indeed, eConference is structured to accommodate the needs of a meeting without becoming an unconstrained online chat discussion. The inceptive idea behind the eConference is to reduce the need for face-to-face meetings, using a simple collaboration tool that minimizes potential technical problems and decreases the time that it would take to learn it.

The tool screen has six main areas: agenda, input panel, message board, hand-raising panel, edit panel, and presence panel (see Figure 1). The *agenda* indicates the status of the meeting, as well as the current item under discussion. The *input panel* enables participants to type and send statements during the discussion. The *message board* is the area where the meeting discussion takes place. The hand-raising panel is used to enable turn-based discussions. The *edit panel* is used to synthesize a summary of the discussion. The *presence panel* shows the participants currently logged in and the role they play. Finally, the *hand-raise panel* mimics the hand-raise social protocol that people use during real meetings to coordinate discussion and turn-taking. Compared with the real-life social protocol, the hand-raise feature of eConference also gives to the moderator the ability to preview queued questions.

The organization of a meeting in eConference follows a protocol inspired by the eWorkshop tool [15]. The *meeting organizer* is guided by a wizard through a few steps in order to:

1. define the main topic and the agenda of the meeting;
2. specify participants invited and their roles;
3. schedule the conference and training sessions, if necessary.

Among the participants invited, the meeting organizer has to select who will act as the moderator and the scribe. The *moderator* is supposed to facilitate the meeting and has control over the participants, whereas the *scribe* captures and summarizes the discussion in the edit panel. Thus, the content of the panel becomes the first draft of the minutes of the meeting. The role of the scribe is flexible in that the participant who is selected as the scribe can change over time and there can be more than one scribe at a time. Finally, some participants may also be invited as *observers*, who will attend the meeting, but they will not be able to actively contribute to the discussion.

3. TOOL EVOLUTION

In accordance with the experience we are reporting here, previous work has pointed that several iterations are needed to get a collaborative system right (e.g. [16]). Figure 2 shows the timeline of the four released versions of eConference. In the evolution of our tool, changes affected both the underlying communication framework and the graphical layer, and, in some cases, the impact of the changes was so high that, rather than porting the code base to a new framework, we decided to redesign the application and build it again from the ground up.

When developing our prototype, we focused on the basic features for supporting smooth discussion and facilitating the meeting creation and execution, so as to maximize the tool effectiveness

DOI: 10.1002/spe

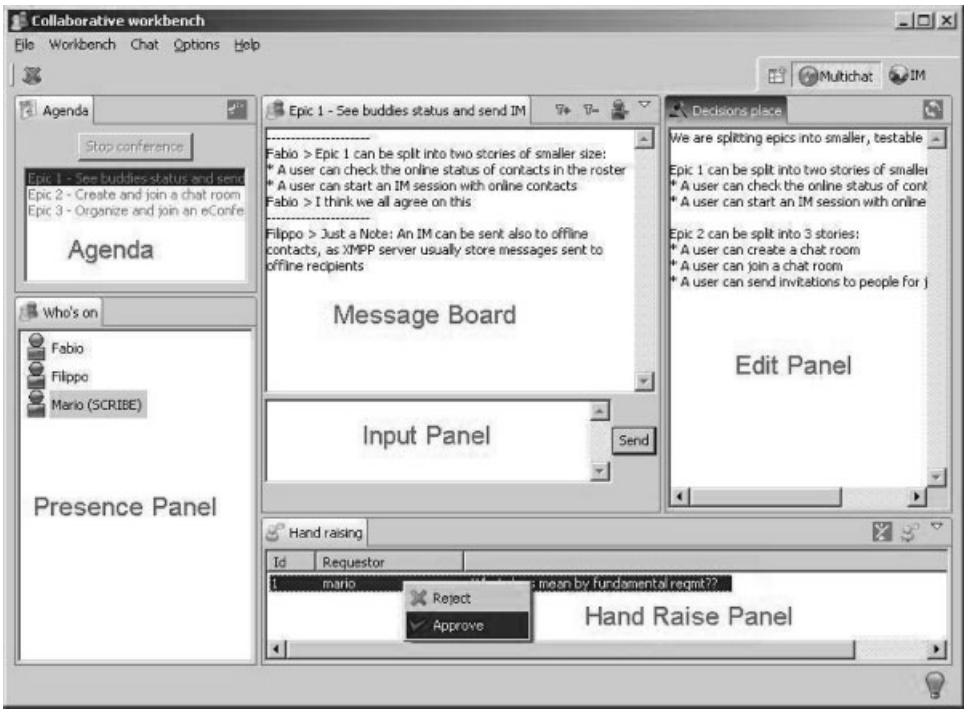

Figure 1. A screenshot of eConference as of ver. 3.0.

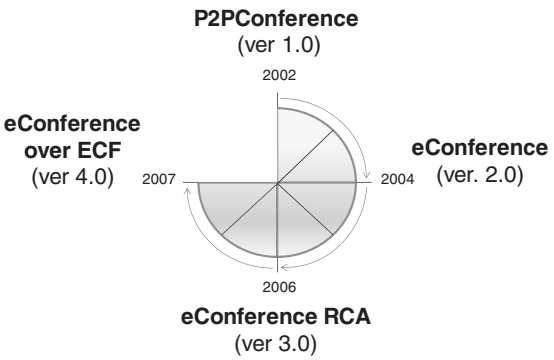

Figure 2. The four versions of the eConference tool.

while minimizing the complexity. Besides, our design and framework selection process was driven by the requirement of keeping the administration cost at a minimum, in order to make the tool not only easy to use, but also to set up and maintain.

The first version of our tool, also known as P2PConference [17], was released in 2002 and was developed using JXTA [18], for the communication infrastructure, and the Swing toolkit, for

DOI: 10.1002/spe

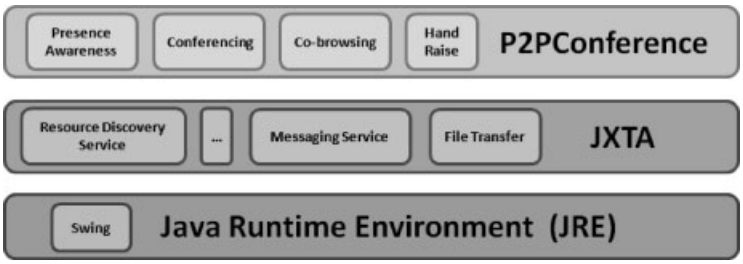

Figure 3. The architecture of P2PConference (ver. 1.0) released in 2002.

designing the user interface. Figure 3 shows the orthogonal architecture of the P2PConference system [19]. Layers represent the frameworks used as the main building blocks of the system and dependencies are implicitly represented by the stacking order. In particular, we followed a relaxed approach to layering [20, pp. 45–46], as each layer may use the services of all the layers below it, and not only to the next lower layer. Within each layer, the inner boxes represent the services or functionality provided to the upper layers. The project was active during the year 2003, but it was completely discontinued in 2004. Project JXTA is an open-source effort that provides a general-purpose, language-independent middleware for building P2P applications [21]. It defines an XML-based suite of protocols that build a virtual overlay network on top of the existing physical network, with its own addressing and routing mechanisms. The building blocks of the JXTA network are rendezvous and relay peers, also referred to as super peers, which deal, respectively, with the resources discovery and message routing. Other than communication features, P2PConference also offered file transfer, available out of the box from JXTA, as well as browser-sharing capability.

The second version of our tool, renamed eConference [22,23], was released in 2004 and replaced the JXTA P2P platform with the XMPP client/server protocol as communication framework, without changing the user interface (see Figure 4). The Jabber project was started in 1999 to create an open alternative to closed instant messaging (IM) and presence services. In 2002 the Jabber Software Foundation contributed the Jabber core XML streaming protocols to the Internet Engineering Task Force (IETF) as XMPP. XMPP was finally approved in early 2004 (RFC 3920–3923) [24] and now it is being used to build not only a large and open IM network, but also mainly develop a wide range of XML-based applications from network-management systems to online gaming networks, content syndication, and remote instrument monitoring [25]. To develop eConference ver. 2 we chose SMACK [26], the most used Java-based XMPP library, because of its elegant event-based programming interface and its full compliancy with the protocol specifications. Among the features available in the P2PConference, only co-browsing and file sharing features could not be easily migrated to work with XMPP, hence we decided not to port them in eConference ver. 2.0. Instead, presence awareness was supported out of the box by XMPP, and hence we removed the ad hoc protocol developed to implement it in JXTA.

In the third version, released in 2006, eConference was redesigned from scratch to evolve from a conferencing system to a collaborative platform, built on top of the pure-plugin Eclipse RCP [27] (Figure 5). The user interface was also rewritten, abandoning the Swing toolkit in favor of SWT. Although mostly known as a powerful Java IDE, recently Eclipse has become a universal plugin platform for creating other extensible rich client applications. In traditional plugin architectures, plugins are mere add-ons that extend the functionality of a host application, that is, binary

DOI: 10.1002/spe

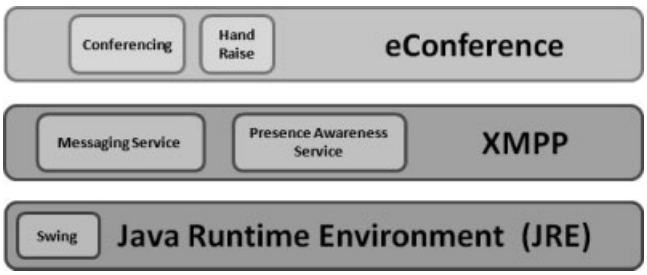

Figure 4. The architecture of eConference (ver. 2.0) released in 2004.

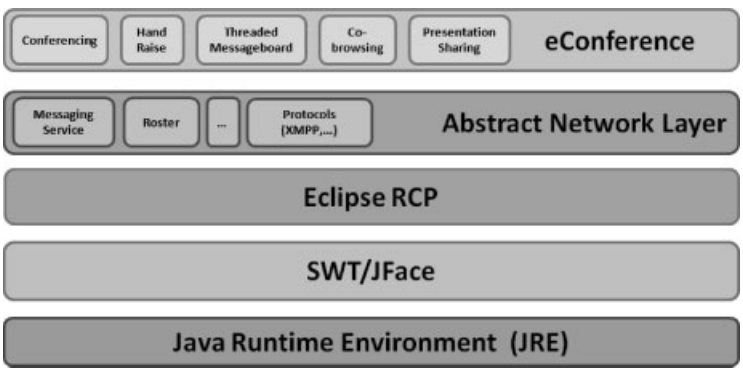

Figure 5. The architecture of eConference RCA (ver. 3.0) released in 2006.

components not compiled into the application, but linked via well-defined interfaces and callbacks. Instead, in pure-plugin systems, plugins become the building blocks of the architecture, as almost everything is a plugin and, consequently, the host application becomes a runtime engine with no inherent end-user functionalities, each of which is obtained through a federation of plugins orchestrated by the engine [28].

Being pure-plugin systems themselves, rich client applications built on Eclipse RCP are fully extensible by architectural design. In eConference ver. 3 we developed our application focusing on its core functionalities (i.e. communicating), expecting to develop all the 'extra features' (e.g. presentation sharing, co-browsing) as plugins. Thus, our tool inherited all the capabilities of the RCP, in terms of extensibility, as well as the well-known benefits from the Eclipse world, such as views, perspectives, and update manager. The experience gained in developing the first two versions of our prototype helped us in identifying the basic features that a communication protocol must provide to work with our tool. Thus, in our rich client application we developed an abstract network layer that exposed the core communication features, which had to be mapped onto concrete network backends, such as XMPP for which we used again the SMACK library because we were already familiar with its event-based model that easily fits into our architecture.

Finally, in 2007 we started to work on the latest (fourth) version of our tool. We decided to replace the abstract network layer existing in ver. 3 in order to be relieved from the burden of

DOI: 10.1002/spe

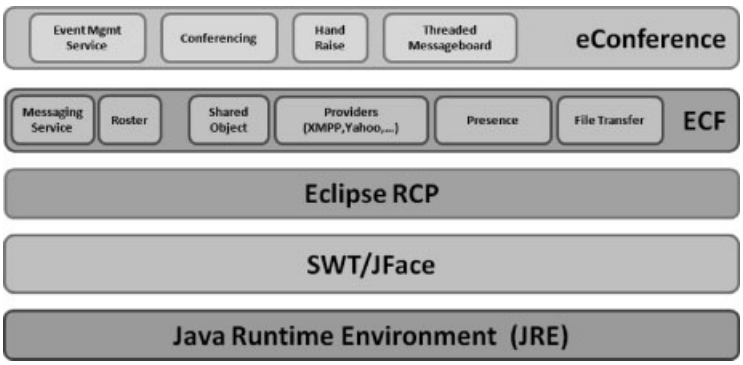

Figure 6. The architecture of eConference over ECF (ver. 4.0) released in 2007.

maintaining it and cope with future evolutions. In eConference ver. 4 (see Figure 6) we reached communication protocol independence through the adoption of the ECF. ECF is an internal project of the Eclipse foundation meant to provide rich client applications with an abstract communication layer and typical collaborative services and features (e.g. IM, white-boarding, file transfer), bundled as a set of reusable plugins [29]. Such components include core Application Programming Interface (API) definitions, graphical user interface widgets, and interfaces for specific network protocols. The ECF core includes an extensible framework, the shared object API, which provides a way for sharing data at the application level, without having to bother with protocol-specific details. Two other notable components available in the ECF include the presence API, which handles the presence events, and the File Transfer API, for sharing content between remote users. All these APIs provide a high-level abstraction layer that promises to enable ECF-based applications to support multiple protocols wholesome, ignoring all the implementation details, which are transparently handled by the underlying framework. ECF, in fact, already provides the implementations (called providers) of abstract interfaces for the most used communication protocols (e.g. MSN/SIMPLE, Yahoo, Skype, XMPP).

In the following sections, we motivate our design choices and discuss the problems encountered during the development of each of the four versions of the eConference tool.

3.1. P2PConference (v 1.0): problems with JXTA

In the first version of our tool, the choice of using a fully decentralized, P2P approach stemmed from our intent of building a distributed meeting system easy to use and set up, with administration costs kept at a minimum. JXTA seemed a promising technology because, by exploiting its virtual network, we aimed at using the existing resources that exist on the edge of the internet infrastructure (e.g. bandwidth, storage space of the PC running the tool). No central server to maintain and no single point of failure is what the platform promised. JXTA did not deliver all of its promises though[‡].

[‡]All the experiences reported and judgments expressed here refer to versions of the platform previous to JXTA 2.3.

DOI: 10.1002/spe

Table I. Impact of JXTA platform changes [30].

Version	Release date	Estimated change impact (compared with previous release)	Actual change impact
1.0 build 49b	2002/02/08	Low	Non-breaking
1.0 build 65e	2002/07/08	None	Non-breaking
1.0 final	2002/09/24	None	Non-breaking
2.0	2003/03/01	High	Breaking
2.1	2003/06/09	Low	Breaking
2.1.1	2003/09/16	None	Non-breaking
2.2	2003/12/15	Medium	Breaking
2.2.1	2004/03/15	Medium	Breaking

None: No changes to API, bug fixes, other improvements; Low: New APIs; Medium: New APIs, APIs changes (deprecations, methods/classes removed, signature changes); High: New APIs, APIs and Protocol changes (no backward compatibility).

In the following, we illustrate the problems encountered while developing using JXTA, distinguishing between architectural and functional issues, respectively.

3.1.1. Low-level unstable API and complexity

Table I lists the eight different releases of the platform used for the development of P2PConference, showing also the estimated and actual change impacts. Depending on whether or not they are backwards compatible, API changes can be classified as breaking or non-breaking [31]. A breaking change is not backwards compatible and may either cause the application to fail to compile and link, or even to behave differently at runtime in the worst case. A non-breaking change is backwards compatible, such as the addition of a new functionality, a performance optimization, or an error removal.

One of the main disadvantages of JXTA was its overly unstable API, which did not properly hide low-level (i.e. implementations) details, thus making developers subject to frequent changes. A low-level API was probably considered as a means to build a general-purpose middleware and grant flexibility to developers, but it ended up adding a considerable amount of extra code and complexity.

Our initial feeling about the complexity and instability of the API was later confirmed by the creation of the JXTA Abstraction Layer, a community project launched at the end of 2002 with the goal of providing an immutable, high-level API meant to hide away from programmers all the useless implementation details for the most commonly used JXTA primitives.

Until the release of the first usable version of the P2PConference, there were two major updates released (ver. 1.0 build 65e and ver. 1.0 final), consisting of only API additions or bug fixes that did not break our code. Since then, however, the changes in the release of JXTA 2.0 were breaking, with a high impact on our code due to large protocol changes. Afterwards, three of the following four releases had significant changes and a considerable impact on the developers.

Sometimes, as in the case of release 2.1, although the impact on developers was assessed as low (i.e. non-breaking), there were some platform behavior incompatibilities at runtime, which actually obliged us to update the code. Indeed, as soon as the super peers that build the overlay network were updated to the latest release, we used to experience erratic behaviors (e.g. failure of resource

DOI: 10.1002/spe

Table II. Alternative JXTA pipe services evaluated.

Pipe service	Since	Type	Needs a server for group communication	Reliability ensured
Unicast	v 1.0	1-to-1	Yes	No
Secure	v 1.0	1-to-1	Yes	No
Propagate	v 1.0	1-to-M	No	No
Bidirectional	v 1.0	1-to-1	Yes	Yes (v2.3+)
JXTA socket	v 2.0	1-to-1	Yes	Yes

discoveries, high rate of lost messages). Thus, even though our code was not broken, not upgrading to the latest release meant a lack of interoperability, that is, we could not properly use fundamental services like routing or discovery, and run our system over the Internet, in a truly distributed mode, but could run it only in our local area network.

JXTA was not only complex for developers but also for end users. The first time a JXTA peer was started and each time the network configuration changed, the user had to manually set up the platform through the JXTA configurator. This was an overwhelmingly complex wizard where a plethora of options had to be set, not only about the network configuration (e.g. behind a firewall or not) but also about the JXTA network itself (e.g. the peer is a rendezvous or relay peer). However, since JXTA 2.0, the community felt the need to bypass the manual configuration and make it fully automatic. Until JXTA 2.2.1, automatic configuration was not sophisticated, as it simply tried to skip manual configuration using common template configurations, which however did not always work well without manual tuning.

3.1.2. Lack of a reliable group-messaging service

The main issue that forced us to abandon the P2P platform was the inadequacy of the JXTA messaging service. In JXTA, the fundamental abstraction used for inter-peer communication is the pipe, a virtual channel consisting of an input and an output end. JXTA offered different alternatives to implement group communication in our prototype (see Table II). As the release of JXTA 1.0, the JXTA core protocol specification defines three kinds of core pipes: unicast, secure, and propagate pipes.

Unicast and secure pipes serve for one-to-one communication, connecting two peers in unicast mode, whereas propagate pipes operate in one-to-many mode, leveraging either IP multicast on the subnet or rendezvous peers. All types of core pipes are not reliable by definition and thus, they cannot guarantee ordered message delivery. We also considered a non-core pipe services, namely bidirectional pipes and JXTA Sockets, whose purpose is to provide bidirectional communication. Bidirectional pipes were available since JXTA 1.0, but became reliable only since the release of JXTA 2.3, when we had already discontinued the prototype development and maintenance. JXTA sockets, available only since ver. 2.0, are fundamentally a reimplementation of the standard Java socket API upon the JXTA pipe infrastructure, and thus reliable by design.

We chose to use the propagate pipe service in our prototype because its one-to-many communication mode was the most apt for implementing group communication in our decentralized system. Despite the fact that reliability was not ensured, propagate pipe was actually the only practical

DOI: 10.1002/spe

solution, as all the other communication services were meant for point-to-point communication. Indeed, the use of any one-to-one service would have entailed the need to set up in the peer group, a super peer that behaved very similar to a server (i.e. receive a message from a peer, then route it to all other known peers). This solution would have defeated any motivation for experimenting a P2P approach, as it would have been equivalent to using a traditional client/server solution, but on a P2P platform and with much more complexity.

Unfortunately, in our experience propagate pipes and discovery on rendezvous peers proved to be too much unreliable, unless all the peers were in the same subnet, using multicast. Instead, when peers were dispersed over the Internet, the results were discouraging, showing a high message drop rate and a low resource discovery recall.

Benchmarking JXTA is a challenging task, far beyond the scope of this paper. Although we have not executed any test, other research studies have somewhat confirmed the problems of the JXTA messaging architecture. In particular, the results of previous works (e.g. [32–35]) show a high variance due to the several platform releases and the many different network settings and peer configurations to be taken into account. In general, findings confirmed that the pipe services have high latency values due to the verbosity of XML messages, and behave reliably in local networks, with a lower message drop rate. In addition, test settings for benchmarking propagate pipe scalability in [33,34] were not representative of our case because those tests were performed considering only one sender and an increasing number of receivers (i.e. 1, 2, 4, and 8 peers), without taking into account the realistic case of multiple senders and receivers in a large peer group over the Internet.

3.2. eConference (v 2.0): the adoption of Jabber/XMPP

JXTA was released in 2001. After having developed with it for over a year and a half, our feeling was that it had been released in a yet too-early stage to become the reference platform for P2P. The discontinuation of the process to standardize the JXTA protocol can be seen as a reflection of its initial lack of maturity. A first draft was submitted to IETF in June 2002, but it naturally expired at the end of 2004, since IETF had previously declined to start a working group, referring JXTA to their sister organization IRTF as a part of the peer-to-peer working group [36]. In addition, JXTA messaging service proved to be inadequate for developing a fully decentralized meeting system. Considering the several issues we encountered during the development of P2PConference, we decided to port the tool onto a different communication platform: Jabber/XMPP.

Compared with JXTA, XMPP offered us three clear advantages. First, XMPP provides by design a robust, extensible, and scalable architecture for near real-time presence, messaging, and structured data exchange. The second advantage is simplicity. XMPP has been conceived to delegate complexities to the servers as much as possible, so that developers can be focused on the application logic, and the clients can stay lightweight and simple. Furthermore, the intrinsic extensibility of XMPP allows leveraging the existing services (e.g. multi-user chat) and also adding extra features (e.g. agenda, hand raise). Third, the IETF standardization of the core XMPP protocols has generated a plethora of high-level XMPP libraries, available for a number of programming languages. XMPP programmers need not even know the protocol details, as all the raw XML exchanges are hidden by the use of any of these APIs.

At a first glance, compared with our previous P2P solution, choosing XMPP might look somewhat contradictory. However, its architecture is not purely client/server, but a hybrid one, very similar to that of e-mail. XMPP entities are identified by a unique Jabber ID, which is all that is needed in

DOI: 10.1002/spe

order to exchange messages. The XMPP network is formed by hundreds of public servers, which are all interconnected to form the XMPP federation. Thus, using the XMPP federation allowed us to develop a meeting system with a client/server-like approach, but without abandoning the initial goal of keeping at minimum the infrastructure costs (i.e. no central server to install and administer, and no infrastructure costs as in the case of P2P).

3.3. eConference rich client application (v 3.0): toward a collaborative platform

The first time we used eConference was to organize and run 16 distributed requirements workshops, with the main intent of testing the tool itself [22]. This pilot study in the distributed Requirements Engineering scenario guided us in the design of the next version of our tool. We analyzed information from multiple sources to collect experience results, namely direct observations of the meetings, conversational logs, and questionnaires, which were then used to evolve the tool. Direct observation helped us to spot design flaws in the implementation of the hand raising and item-based message board features, also confirmed by the log analysis, whereas the feedback from the participants allowed us to obtain feature suggestions and enhancements.

When we developed eConference ver. 2, porting our tool from JXTA to XMPP, we lost some features (namely file- and browser-sharing), because they could not be easily adapted. From this drawback we realized that all the effort spent in adapting the tool to support another communication platform in future releases must be avoided. Furthermore, it is overly challenging to foresee all the possible features needed to make a meeting system flexible enough to be apt for all contexts. These concerns led us to think about how to make eConference more flexible. Our intention was to have a platform that offered as the core functionality a reliable, extensible, and scalable messaging framework, on which new collaborative features could be added as plugins. We also wanted to support multiple communication protocols through pluggable network backends, so as to have the possibility to add a new one at any time by writing only the specialized code for its integration. To support the composition of a larger system that is not pre-structured, or to extend it in ways that cannot be foreseen, an architecture that fully supports extensibility is needed. We thus decided to build the new prototype by exploiting Eclipse RCP. Among the steps we took during the evolution of our tool, the development of the third version was definitely the most 'revolutionary'. In fact, initial release apart, while the technology changes in the other releases did not alter the nature of the tool, with ver. 3 we shifted our focus from building a collaborative application to building a collaborative platform.

The eConference ver. 3 was developed incrementally, using a story-driven agile process [37]. We started building a feature (i.e. a collection of plugins in Eclipse terminology) to provide IM and presence awareness capabilities, which were available out of the box since they are both at the core of XMPP. Then, we extended the existing feature to implement multi-user chat for reliable group communication. Unlike presence and IM, multi-user chat is not a core functionality of XMPP. Instead, it is available as an XMPP Extension Proposal (XEP). The Jabber Software Foundation develops extensions to XMPP through a standards process centered on XEPs. The multi-user chat XEP [38] is the protocol extension proposed for managing chat rooms. Though not in the final stage yet, this draft is already supported by all the hundreds of public servers belonging to the XMPP federation. One limitation we found with the multi-user chat extension was that it did not handle typing awareness. We tackled this problem leveraging the intrinsic extensibility of XMPP and creating a custom typing notification, sent whenever a participant in the room starts to type.

DOI: 10.1002/spe

Finally, leveraging the functionality already provided by the multi-user chat feature, we developed new plugins for each view needed, namely the agenda, edit panel, and hand raising, so as to obtain the overall 'eConference feature'. Indeed, rather than an application, in eConference ver. 3 the conferencing became just a feature of our rich client application, with its own perspective. Similarly, to develop new features for browser- and presentation-sharing, we built onto the existing features and plugins, and created new perspectives to optimize the arrangements of the UI views. Finally, we added to the eConference feature the support for one-to-one private messaging and message threading.

3.4. eConference over ECF (v 4.0): toward protocol independence

Although designed to be independent of the network protocol and implemented using a pure-plugin architecture, eConference ver. 3 suffered from some architectural drawbacks, namely (1) a low-level, abstract network layer, too expensive to maintain on our own and (2) a burdensome publish/subscribe subsystem, in which every bundle implemented the observer pattern [2, pp. 293–303], without taking advantage of the Eclipse internal mechanism for appropriately handling events dispatching in a dynamic, pure-plugin environment [39].

While the second drawback was due to our initial inexperience with the development of the Eclipse RCP platform, the first one was imputable to a design choice of ours. Although mainly interested in XMPP, in the third generation of the eConference tool we designed and implemented an abstract network infrastructure layer to allow for the use of other communication protocols in the future, without a severe impact on the code base. Consequently, all the domain-specific features were built on such internal API. As a side effect, the low-level network layer had to be maintained in addition to the application itself, although our main intention was to focus the effort on the development of domain components.

3.4.1. Porting to ECF versus redeveloping

When we were starting the development of eConference ver. 4, we intended to have two alternative solutions, that is, either porting the earlier version to ECF or developing the new version from scratch. Initially, the preferred alternative was the porting because it was alleged to be faster and it would have allowed to retain a larger portion of the codebase we had already developed. Instead, a porting of eConference to ECF turned out to be infeasible for the proper use of ECF [40].

One of the aspects we overlooked when we decided to draw on ECF was that it is a 'vertical' communication middleware, since it does not come only with a set of network services. Instead, ECF already provides several out-of-the-box graphical components, along with the respective services (e.g. contacts roster, chat editors, and user account management), which can be embedded in any Eclipse-based application. In fact, between eConference ver. 3 and ECF there was a large overlapping among the basic communication features they both provided, in terms of APIs, visual components, and model objects. Integrating frameworks'-specific representations of the real-world components is a common problem, already acknowledged by previous research and referred to as the 'overlapping principle' [10]. This problem is generally tackled through multiple inheritances, which typically cause an increase in complexity, other than being unsupported by Java.

Thus, due to the larger than expected impact of ECF, the efforts of porting and redeveloping were almost equivalent, since only the user interface layer could be retained. Hence, we decided to rewrite

DOI: 10.1002/spe

Table III. Components required by eConference 3 and their support in ECF (only the major components are listed).

Available in eConference 3	Provided by ECF
Message board*	Partially
Roster view	Yes
Extension points API	Yes
Hand raising	No
White board	Yes
Conferencing events manager	No
Account creation/Login manager	Yes

*Does not support multiple discussion threads.

the application, building upon the ECF API and services, and reusing the existing user interface code when possible. Table III shows the main features and the associated graphic components available in eConference ver. 3, and whether they could be fully replaced by employing ECF. The fourth version of eConference was built just reusing the native ECF plugins when available out-of-the-box (e.g. contacts roster, whiteboard), extending those plugins which had only a partial support (e.g. threaded message board), and rebuilding the missing plugins upon ECF API (e.g. hand raising, event manager).

The development of eConference over ECF was guided by the application of both design and architectural patterns. The classic Model-View-Controller architectural pattern [20, pp. 125–143] was used to gain the separation of concerns between data manipulation and visualization. However, because eConference is a distributed application, some problems arose to keep synchronized each model of the eConference instances running across the Internet. Hence, the Proxy design pattern [2, pp. 207–217] was used in order to let local changes of a model be also consistently and transparently replicated on other remote instances' models through the ECF API. Besides, the observer design pattern [2] was used for creating listeners that react to event updates, both locally and remotely (e.g. participants joining or leaving a conference, changes to the agenda). The second architectural pattern applied was the Dependency Injection [41], which was used in conjunction with Model-View-Controller for further increasing object decoupling. Dependency Injection takes care of how to wire objects together. When an object needs to gain access to a service, this pattern can be utilized so that the object only needs to have a property for holding a reference to the desired service. As soon as an object is created, an external mechanism automatically injects a reference to an implementation of the service into the property.

4. OPEN ISSUES

While evolving eConference we found some issues that have been already acknowledged by previous research (e.g. [42]), but are still open to debate.

4.1. Complexity on server side versus extensibility on client side

In our experience XMPP proved to be more stable, easy-to-use, and reliable than JXTA. However, our preference for XMPP over JXTA is not based on a preference for the client/server paradigm

DOI: 10.1002/spe

over P2P. On the whole, XMPP is a good choice for applications that need a flexible messaging framework: although not fully extensible, in general the level of extensibility ensured by its XML-based protocols has proved sufficient to expand the multi-user chat capability and add the other extra functionality that we needed to build eConference.

Nevertheless, the main problem encountered stemmed from the limitations of the current multi-user chat implementation. In eConference vers. 2 and 3 a problem arose when synchronizing clients in case of latecomers or unintentional disconnections. The multi-user chat extension ensures persistence, delegating to servers the tasks of history logging and dispatching. Thus, in both cases, all the events are sent back to clients in order. However, all the custom notifications that we added, such as agenda items selection and edit panel updates, were logged in the history as if they were participants' utterances. That is, when clients were synchronized, servers did not send the current content of the edit panel or agenda all at once; instead, each and every change made was sent in chronological order. Furthermore, synchronization also included useless notifications, like speaking requests and typing awareness. As a quick fix, on the client side we could only prevent the notification of these events to be saved in the history, thus alleviating the issue by limiting the size of the history to be stored and propagated. However, according to the XMPP philosophy (i.e. moving the complexity away from the client), to completely overcome the synchronization problem, it should be tackled from the server side. Hence, to accomplish a comprehensive solution we should have either submitted an extension proposal for the existing multi-user chat extension, or written on top of it a new extension proposal for a 'structured multi-user chat' that handles history synchronization at a lower level. Writing a new extension proposal is a neat solution, in line with the XMPP philosophy, although it has a drawback in terms of the time required. To be accepted, any new extension proposal has to go through the XEP standards process, which involves a discussion on mailing list, formal review, voting by the Jabber Council, and, eventually, the approval as protocol extension. Thus, in the worst case, a new extension proposal submitted can be rejected at the end of the process, otherwise, in the best case, it will take several months and revisions before the draft becomes mature enough for public servers to implement it.

When building a communication-intensive tool, choosing between client/server and P2P communication frameworks is not just a matter of resource exploitation. Application protocol extensibility is another factor to consider carefully. Client/server frameworks, like XMPP, trade a reduced workload for developers, since the main effort is on the development of the client only, for a loss of extensibility. P2P frameworks, like JXTA, are fully extensible because they are a middleware that builds an overlay, logical network atop of the physical one, where the definition of application protocols is left to developers.

4.2. Static memory overhead

A known issue when building application on frameworks is the increased amount of additional disk space that an application needs when using a framework [6]. The static memory overhead is the result of additional framework code that is linked into an application, even though the application may not necessarily use it.

In general, Eclipse RCP is an excellent framework that, with a little more coding, offers to an application all the benefits seen in Eclipse (e.g. pure-plugin architecture, perspectives, update manager, help system). However, the final size of the product itself largely increased because of all the Eclipse RCP libraries to be included, even if not all of its services were utilized. The final

DOI: 10.1002/spe

size of eConference was raised from less than 1 MB in ver. 2 to about 9 MB in ver. 3, with the custom plugins developed, plus all the other third-party libraries we used (e.g. SMACK), accounting for only 980 kB. This limitation of Eclipse RCP is already acknowledged [43] and the Eclipse community has been working to reduce the minimal set of libraries needed.

With regard to dynamic memory overhead, no comparison can be made because ver. 2 used the pure-Java Swing toolkit for building the user interface, whereas ver. 3 used the SWT library, which uses OS native widgets (i.e. buttons, trees, menus) when possible, rather than rewriting all of them in Java, thus granting in general a smaller memory footprint.

The problem of excessive code size has long been acknowledged (e.g. see [42]), but, despite the large advances in the ease of use and features provided, new application frameworks still fall short of addressing this limitation.

5. LESSONS LEARNED

After four generations of eConference, we learned some lessons, which may be of help when making critical architectural decisions about frameworks usage.

5.1. Stability as a key aspect

According to Garlan *et al.* [42], when building products on application frameworks problems arise because of mismatched assumptions about used components, that is, the primary computational and storage entities of the system (e.g. databases, servers, etc.), and connectors that determine the interactions between components (e.g. client/server protocols, RPC links).

Our experience with JXTA was not positive. Although JXTA aimed at addressing a real problem (i.e. the fragmentation and redundancy of services offered by the plethora of existing P2P platforms), it failed to deliver a robust, general-purpose platform that can serve as the building blocks for P2P communication-intensive applications. Paradoxically, its messaging framework proved inappropriate for implementing group communication unless a client/server-like approach is used.

Developing a prototype for risk assessment and reduction would have probably shown that JXTA pipe services were not suitable for many-to-many communication in the pure P2P approach, and that the platform API was too low level and complex. However, a prototype could not spot the platform API instability issues, which were identified only after several releases.

Ideally, the interface to a framework component should never change. In practice, however, new versions of software components often change their interfaces, resulting in an increased effort of development and maintenance. Stability is a key aspect of any application framework to guarantee the promised independence between producers (i.e. software developers who write the framework implementation) and consumers (i.e. software developers who write code with method calls to the framework). Hence, changes in the framework API require changes in the consumers' code as well [44]. Determining the right set of narrow interfaces for developers to use has been acknowledged as a factor affecting the success of application frameworks [6]: If key interfaces are not stable, developers may have difficulty in understanding and applying the framework effectively and efficiently.

As framework consumers, we did not expect the JXTA API to change often and our mismatched assumption about the JXTA connector was that we did not imagine it to have backward compatibility issues. Although the importance of keeping an application framework stable, it seems a rather

DOI: 10.1002/spe

obvious lesson, our experience shows that it was not so obvious to the framework producers. This lesson suggests that, before selecting a framework, stability should be ensured by carefully observing and analyzing API changes over a proper timeframe, if source code is available.

5.2. Beware of reputation attribute in framework selection

Although there have been a number of formal models for the selection process of application frameworks [45], and more generally of COTS products [7], field studies [4] have found that developers seldom follow any formal selection process. Familiarity with frameworks and the reputation of frameworks and vendors are often the only attributes considered. For instance, we used the SMACK library for developing eConference ver. 2 because it was reputed to be the most complete Java API for XMPP. We reused the SMACK library also in eConference ver. 3 because we were already familiar with its internal event-dispatching model. Both attributes can lead to suboptimal choices, but while framework familiarity can flatten the learning curve, reputation can be totally misleading.

The first lesson taught us that, although prototypes can be useful to spot the strengths and weaknesses of frameworks, they cannot capture instability problems, which can only be observed over longer time intervals. With communication frameworks, however, stability can be ensured by protocol standardization. In 2002, when we were starting the development of eConference ver. 1, JXTA and Jabber/XMPP were selected as candidates because they both matched the criteria of being platform-independent communication frameworks that could ensure the reduction of infrastructure costs. At that time, however, neither JXTA nor Jabber/XMPP were already a standard, as they had only been contributed to IETF as drafts. The main force that drove us to select JXTA instead of Jabber/XMPP was the fact that the former was an effort led by Sun Microsystems, which had generated a lot of hype, whereas the latter was still an effort of volunteers, which had not attracted the attention of large companies yet. Nevertheless, the standardization process of JXTA failed because IETF refused to create a working group for it, whereas Jabber/XMPP, after several revisions, was finally approved in early 2004.

Thus, the second lesson that we learned is that to select good communication frameworks from the candidates when they are not standards, the progression of frameworks' standardization process is a more reliable attribute than the reputation of the vendors or of the frameworks themselves.

5.3. Hand over maintenance of critical infrastructure to a large community

One of the benefits of building applications upon frameworks is the easier development of elaborated architectures and complex pieces of software. Nevertheless, leveraging frameworks do not relieve developers from the cost of maintaining the resulting artifact. In our case, although the use of Eclipse RCP eased the complexity of implementing an abstract network architecture, it did not relieve us from the cost of change. With the development of eConference ver. 3 we realized that a small research group could not sustain the cost of maintaining on its own such an abstract communication network infrastructure.

One benefit of rewriting eConference ver. 4 upon ECF was that by employing a standard network technology, which is maintained by a larger community, the costs of adapting to the changes in the network layer could be avoided altogether. When you employ a framework, you effectively outsource a portion of your software development and, consequently, developers will not be responsible for

DOI: 10.1002/spe

the maintenance except to the extent that they are developing extensions or plugin components for the framework [46]. While it is possible for few developers to support a project with a limited scope, having a large community around makes the project more sustainable.

6. CONCLUSION

To date, four generations of eConference have been developed. The experience gained in developing the first two generations of our prototype helped us in identifying the basic features to provide and the characteristics that a communication protocol should exhibit to work well with our tool. In the third generation, the choice of Eclipse RCP gave us a means to build a system with greater flexibility and maintainability, capable of coping with change. However, RCP provides no facilities that allow for change also at the level of the communication protocol to be employed. Hence, in the fourth release we used ECF, a networking framework that promises to provide any RCP-based tool with the ability to support several communication protocols simultaneously.

The key contribution of this paper is represented by the lessons learned from the evolution of the eConference tool, with an emphasis on the selection and usage of appropriate application frameworks to build a flexible distributed conferencing tool. Lessons learned about framework stability, reputation attribute, and continuous delegation to outer communities have the potential to affect the evolution process of similar communication-intensive applications.

ACKNOWLEDGEMENTS

This work has been partially supported by the 2006 Eclipse Innovation Award and the MiUR-Italy under grant 2006 'Metamorphos'. We wish to thank Mario Scalas and all the other students at University of Bari who contributed to the development and evolution of the eConference tool as a part of their final year theses.

REFERENCES

1. Fayad ME, Schmidt DC. Object-oriented application frameworks. *Communications of the ACM* 1997; **40**(10):32–38.
2. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Addison-Wesley, Longman Publishing Co., Inc.: Boston, MA, 1995.
3. Johnson RE. Frameworks = (components + patterns). *Communications of the ACM* 1997; **40**(10):39–42.
4. Torchiano M, Morisio M. Overlooked aspects of COTS-based development. *IEEE Software* 2004; **21**(2):88–93.
5. Hou D, Hoover HJ. What can programmer questions tell us about frameworks? *Proceedings of the 13th International Workshop on Program Comprehension* (*IWPC 2005*), St. Louis, Missouri, U.S.A., 15–16 May 2005; 87–96. DOI: 10.1007/11575801_11.
6. Schmidt DC, Gokhale A, Natarajan B. Leveraging application frameworks. *ACM*, *Queue*, 2004; 66–75. DOI: 10.1145/1016998.1017005.
7. Mohamed A, Ruhe G, Eberlein A. COTS selection: past, present, and future. *Proceedings of the 14th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (*ECBS'07*), Tucson, AZ, U.S.A., 26–29 March 2007; 103–114. DOI: 10.1109/ECBS.2007.28.
8. Ruhe G. *Intelligent Support for Selection of COTS Products* (*Lecture Notes in Computer Science*, vol. 2593). Springer: Berlin, Heidelberg, 2003; 34–45. DOI: 10.1007/3-540-36560-5.
9. Shull F, Lanubile F, Basili V. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering* 2000; **26**(11):1101–1118.
10. Mattsson M, Bosch J, Fayad ME. Framework integration problems, causes, solutions. *Communications of the ACM* 1999; **42**(10):80–87. DOI: 10.1145/317665.317679.

11. Calefato F, Lanubile F, Scalas M. Evolving a text-based conferencing system: An experience report. *Proceedings of the Third International Conference on Collaborative Computing*: *Networking*, *Applications and Worksharing* (*CollaborateCom 2007*), New York, U.S.A., 12–15 November 2007; 427–431. DOI: 10.1109/COLCOM.2007.4553869.
12. Calefato F. Supporting synchronous communication in distributed software teams. *PhD Thesis*, University of Bari, 2007; 18–24. Available at: http://www.sigsoft.org/phdDissertations/ [23 April 2009].
13. Calefato F, Damian D, Lanubile F. An empirical investigation on text-based communication in distributed requirements engineering. *Proceedings of the 2nd International Conference on Global Software Engineering* (*ICGSE '07*), Munich, Germany, 27–30 August 2007; 3–11. DOI: 10.1109/ICGSE.2007.9.
14. Nunamaker JF, Dennis AR, Valacich JS, Vogel D, George JF. Electronic meeting systems. *Communications of the ACM* 1991; **34**(7):40–61.
15. Basili V, Tesoriero R, Costa P, Lindvall M, Rus I, Shull F, Zelkowitz M. Building an experience base for software engineering: A report on the first CeBASE eWorkshop. *Product Focused Software Process Improvement*, *PROFES 2001* (*Lecture Notes in Computer Science*, vol. 2188). Springer: Berlin, Heidelberg, 2001; 110–125. DOI: 10.1007/3-540-44813-6_13.
16. Boehm B, Grunbacher P, Briggs RO. Developing groupware for requirements negotiations: Lessons learned. *IEEE Software* 2001; **18**(3):46–55. DOI: 10.1109/52.922725.
17. Calefato F, Lanubile F, Mallardo T. Peer-to-peer remote conferencing. *Third International Workshop on Global Software Development* (*GSD '04*), Edinburgh, Scotland, U.K., 2004. IEE Publishing: London, 2004; 34–38. DOI: 10.1109/ic:20040310.
18. JXTA community projects. Available at: https://jxta.dev.java.net/ [23 April 2009].
19. Rajlich V, Silva JH. Evolution and reuse of orthogonal architecture. *IEEE Transactions on Software Engineering* 1996; **22**(2):153–157. DOI: 10.1109/32.485224.
20. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. *Pattern-Oriented Software Architecture Volume 1*: *A System of Patterns*. Wiley: New York, 1996.
21. Brookshier D, Krishnan N, Govoni D, Soto JC. *JXTA*: *Java P2P Programming*. Sams Publishing: Indianapolis, IN, U.S.A., 2002; 39–85.
22. Calefato F, Lanubile F. Using the Econference tool for synchronous distributed requirements workshops. *Proceedings of the 1st International Workshop on Distributed Software Development* (*DiSD 2005*). Austrian Computer Society: Paris, France, 29 August 2005; 97–107.
23. eConference Project Wiki. Available at: http://cdg.di.uniba.it/index.php?n=Research.Econference [23 April 2009].
24. XMPP protocol specifications. Available at: http://xmpp.org/rfcs/ [23 April 2009].
25. St. Andre P. Streaming XML with Jabber/XMPP. *Internet Computing*, *IEEE* 2005; **9**(5):82–89.
26. SMACK library. Available at: http://www.igniterealtime.org/projects/smack/index.jsp [23 April 2009].
27. McAffer J, Lemieux J-M. *Eclipse Rich Client Platform*: *Designing*, *Coding*, *and Packaging Java*^{TM} *Applications*. Addison-Wesley Professional: Reading, MA, 2005.
28. Birsan D. On plug-ins and extensible architectures. *Queue*, *ACM* 2005; **3**(2):40–46.
29. Eclipse Communication Framework (ECF). Avilable at: http://www.eclipse.org/ecf [23 April 2009].
30. JXTA platform change history. Available at: https://jxta-jxse.dev.java.net/currentwork.html#history [23 April 2009].
31. Dig D, Johnson R. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution* 2006; **18**(2):83–107. DOI: 10.1002/smr.v18:2.
32. Antoniu G, Hatcher P, Jan M, Noblet DA. Performance evaluation of JXTA communication layers. *IEEE International Symposium on Cluster Computing and the Grid* (*CCGrid '05*), Cardiff, U.K., vol. 1, 9–12 May 2005; 251–258. DOI: 10.1109/CCGRID.2005.1558562.
33. Halepovic E, Deters R. The cost of using JXTA. *Third International Conference on Peer-to-Peer Computing* (*P2P '03*), Linköping, Sweden, 2003. IEEE Computer Society: Sliver Spring, MD, 2003; 160–167.
34. Halepovic E, Deters R, Traversat B. *JXTA Messaging*: *Analysis of Feature-Performance Tradeoffs and Implications for System Design* (*Lecture Notes in Computer Science*, vol. 3761). Springer: Berlin, Heidelberg, 2005; 1097–1114. DOI: 10.1007/11575801_11.
35. Seigneur J-M. Jxta Pipes Performance, 2002. Available at: http://cui.unige.ch/∼seigneur/ [23 April 2009].
36. JXTA IETF standardization. Available at: https://datatracker.ietf.org/drafts/draft-duigou-jxta-protocols/ [23 April 2009].
37. Cohn M. *User Stories Applied*: *For Agile Software Development*. Addison-Wesley: Reading, MA, 2004.
38. XMPP Multi-User Chat (MUC) XEP. Available at: http://xmpp.org/extensions/xep-0045.html [23 April 2009].
39. Gruber O, Hargrave BJ, Mcaffer J, Rapicault P, Watson T. The eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal* 2005; **44**(2):289–299.
40. Calefato F, Lanubile F, Scalas M. Porting a distributed meeting system to the eclipse communication framework. *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange* (*ETX 2007*), Montréal, Canada, 21–22 October 2007; 46–49. DOI: 10.1145/1328279.1328289.
41. Fowler M. Inversion of control containers and the dependency injection pattern. Available at: http://martinfowler.com/articles/injection.html [10 October 2008].

42. Garlan D, Allen R, Ockerbloom J. Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the 17th International Conference on Software Engineering* (*ICSE '95*), Seattle, WA, U.S.A., 24–28 April 1995; 179–185. DOI: 10.1145/225014.225031.
43. Eclipse RCP static memory overhead. Available at: https://bugs.eclipse.org/bugs/show_bug.cgi?id=53338 [23 April 2009].
44. de Souza CRB, Redmiles D, Millen D, Patterson J. Sometimes you need to see through walls: A field study of application programming interfaces. *International Conference on Computer Supported Cooperative Work* (*CSCW '04*), Chicago, IL, U.S.A., 6–10 November 2004; 63–71.
45. Cortellessa V, Marinelli F, Potena P. An optimization framework for 'build-or-buy' decisions in software architecture. *Computers and Operations Research* 2008; **35**:3090–3106.
46. Fayad ME, Hamu DS. Enterprise frameworks: Guidelines for selection. *ACM Computing Surveys* 2000. Article No. 4. DOI: 10.1145/351936.351940.