Abstract: 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. Usually, it takes several iterations to get a collaborative system right. Here we discuss the lessons learned from the evolution of our conferencing tool over four generations, with an emphasis on the selection and adoption of appropriate application frameworks.
Adopting Frameworks to Develop a Distributed Conferencing System: Lessons Learned

Fabio CALEFATO, Filippo LANUBILE, and Mario SCALAS

Dipartimento di Informatica
University of Bari
Bari, Italy
{calefato, lanubile, scalas}@di.uniba.it

Abstract—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. Usually, it takes several iterations to get a collaborative system right. Here we discuss the lessons learned from the evolution of our conferencing tool over four generations, with an emphasis on the selection and adoption of appropriate application frameworks.

Index Terms—Software evolution; software reuse; application framework; communication framework; collaborative system.

1. INTRODUCTION
In the last decade being a good programmer has more and more shifted from mastering programming languages to mastering application frameworks as well. Good programmers have to be 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 [19] and are defined as a set of cooperating classes that make up a reusable design for a specific software [21], [26]. Here, by using the expression application frameworks (or simply frameworks) we mean both closed-source COTS (Commercial Off-The-Shelf) 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 [41].

Typically, a number of iterations are necessary to develop a new system leveraging application frameworks, in order to fully realize their benefits. Indeed, although frameworks are a powerful means to reduce software development costs while improving quality, they are at the same time difficult to select and understand, as well as hard to learn, use, and integrate effectively and efficiently [25], [36].

Although there is no commonly accepted method for frameworks selection [32], [35], all 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 [40].

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, not for integration [30].

In this paper we describe the evolution of eConference [11], a distributed conferencing system that has been developed as 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 [8], [9].

When developing distributed conferencing systems, complexity and usability are major problems [33]. 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 complexity. However, the scope of a distributed conferencing system goes beyond supporting users' activities during the meeting itself, but also to facilitating the arrangement and set up of meetings.

Usually, it takes several iterations to get a collaborative system right [4]. Through the years, we first changed the underlying communication framework, from the JXTA P2P platform (ver. 1) to the 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 (ver. 3). Finally, in the latest version, eConference has reached communication protocol independency through the Eclipse Communication Framework (ver. 4).

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.

The remainder of this paper is structured as follows. In Section 2 we briefly describe the eConference tool. Then, from Section 3 to Section 6 we present each of the four versions of eConference, motivating the design choices and discussing the problems encountered during its development. In Section 7 and 8 we, respectively, discuss the open issues and lessons learned. Finally, in Section 9 conclusions are drawn.

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 control the discussion during distribute meetings. Indeed, eConference is structured to accommodate the needs of a meeting without becoming an unconstrained on-line 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 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 (“started”, “stopped”), 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 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 to the real-life social protocol, the hand raise feature of eConference also gives to the moderator the ability to preview queued questions, showing a tooltip when hovering the mouse pointer over them.

[Insert Figure 1 here]

Figure 1. A screenshot of eConference

The organization of a meeting in eConference follows a protocol inspired by the eWorkshop tool [2]. 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 moderator
and scribe. The *moderator* is supposed to facilitate the meeting and has control over 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 meeting minutes. The role of scribe is flexible in that the participant who is selected as 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*, in that they will attend the meeting, but they will not be able to actively contribute to the discussion.

### 3. P2PConference: Problems with JXTA

The first version of our tool, also known as P2PConference [6], was developed using the Java binding of JXTA [29]. Project JXTA is an open-source effort which provides a general-purpose, language-independent middleware for building P2P applications [5]. 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.

The choice of adopting a fully-decentralized, P2P approach stemmed from our intent of building a distributed meeting system easy to use and set up, with administration cost kept at minimum. JXTA seemed a promising technology because, by exploiting its virtual network, we aimed at using existing resources that live on the edge of the Internet infrastructure (e.g., bandwidth, storage space of the PC’s running eConference). No central server to maintain and no single point of failure is what the platform promised. JXTA did not deliver on all of its promises though.¹

¹ All the experiences reported and judgments expressed here refer to versions of the platform previous to JXTA 2.3.
**Low-level Instable API & End-User Complexity**

The development of P2PConference started in early 2002, using the Java binding of JXTA. The first useable version of P2PConference was released at the end of the same year. The project was active during the year 2003, when file sharing and co-browsing features were added, but it was completely discontinued in 2004.

Table 1 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 [15]. 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.

<table>
<thead>
<tr>
<th>Table 1. Impact of JXTA platform changes [28]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Insert Table 1 here]</td>
</tr>
</tbody>
</table>

One of the main disadvantages of JXTA was its overly low-level instable API, which made 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 considerable amount of extra code and complexity.

Our initial feeling about the low level and complexity 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 for all the most commonly used JXTA primitives.

Until the release of the first useable version of P2PConference, there were two major-update released (ver. 1.0 build 65e and ver. 1.0 final), with 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 developers. The impact, as assessed in the release-announce emails sent on the JXTA mailing list, was occasionally optimistic.

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 tool. 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 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, i.e., we could not properly use fundamental services like routing or discovery, and run our system over the Internet, in a truly distributed mode, but only in our subnet, using IP multicast.

JXTA was not only complex for developers, but even for end users. The first time a JXTA peer was started and each time network configuration changed, the user had to manually set up the platform through the JXTA configurator.

The configurator was overwhelmingly complex because a plethora of settings were provided, 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 an edge, rendezvous or relay). Furthermore, it did not try to make any automatic setup (e.g., use of HTTP tunneling rather than TCP, behind a firewall/NAT). 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 template configuration files (e.g., HTTP
firewalled edge peer, TCP rendezvous peer) and it did not always work well without manual tuning.

Lack of a Reliable Group-Messaging Mechanism

The main issue that forced us to abandon the P2P platform was the inadequateness 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 2). Since 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. Propagate pipes, instead, 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 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 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, results were discouraging, with high message drop rate and low resource discovery recall.

Although we have not collected data from formal tests or benchmarks, other research studies have somewhat confirmed the problems of the JXTA messaging architecture in general. Benchmarking JXTA is a hard challenge and test results show a high variance, because of the several platform releases and the very many different network settings and peer configurations to take into account (e.g., using multicast or rendezvous discovery, relay peers or direct connection, TCP, UDP or HTTP). In their analysis of pipe services performance in versions 1.0 build 49b and 1.0 build 65e, Seigneur et al. found that unicast pipes behaved reliably only using TCP in local/LAN test scenario, whereas an extremely high message-drop rate was found when using HTTP [37]. Halepovic & Deters tested performances of core and non-core pipe services for three JXTA releases (1.0, 2.0, and 2.2) in both LAN and WAN settings [23], [24]. Results reported in these studies are positive in terms of scalability in both settings, also for propagate pipes, although the authors also say that they perform best on the LAN when UDP multicast is available. However, these tests were performed considering only one sender and an increasing number of receivers (1, 2, 4, and 8 peers). Hence, these tests on the scalability of propagate pipes
did not take into account the realistic case of multiple senders and receivers in a large peer group over the Internet, messaging through relays and performing discovery on rendezvous. Finally, Antoniou et al. concluded that throughput of JXTA socket is similar to plain socket throughput on LAN, whereas latency values are higher, due to the verbosity of XML messages [1].

4. Rebuilding over 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, not mature enough, probably just on the heels of the growing popularity and hype of P2P. The discontinuation of the process to standardize the JXTA protocol is a reflection of this immaturity. 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 [27]. In addition, JXTA messaging service proved to be inadequate for developing a fully decentralized meeting system. Considered the several issues we encountered during the development of P2PConference, we decided to port the tool onto a different communication platform. Our choice fell onto Jabber/XMPP.

The Jabber project 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 IETF as XMPP, eXtensible Messaging and Presence Protocol. XMPP was finally approved in early 2004 (RFC 3920–3923) [43] and now it is being used to build not only a large and open IM network, but also and mostly to develop a wide range of XML-based applications, from network-management systems to online gaming networks, content syndication, and remote instrument monitoring [39].

Compared to JXTA, XMPP offered us three clear advantages. First, XMPP provides by design
a robust, extensible, secure 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 keep 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 APIs, available for a number of programming languages. XMPP programmers do not even need to 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 to our previous P2P solution, choosing XMPP might look somewhat contradictory. However its architecture is not purely client/server, but a hybrid, very similar to email. XMPP entities are identified by a unique Jabber ID, which is all that is needed in order to exchange messages. The XMPP network is formed by hundreds public servers, which are all interconnected to form the XMPP federation. Although running an XMPP server which is not part of the federation is still possible for a corporate LAN, from our perspective, using the XMPP federation was preferable because it allowed us to develop a client/server meeting system, without abandoning the goal of keeping at minimum the infrastructure costs (i.e., again no central server to install and administer, and no infrastructure costs, as in the case of P2P).

The implementation that used XMPP as network backend was called eConference (ver. 2.0) [16]. To develop eConference we chose SMACK [38], the most used Java-based XMPP library, because of its elegant event-based programming interface and its full compliancy with the protocol specs. Unfortunately, co-browsing and file sharing features could not be easily migrated to work with XMPP, as they needed to be rewritten from scratch.
5. **eConference as a Collaborative Platform**

When developing our prototype, we initially focused on basic features for supporting smooth discussion and facilitating meeting creation and execution, so as to maximize the tool effectiveness while minimizing complexity. The first time we used eConference was to organize and run sixteen distributed requirements workshops, with the main intent of testing the tool itself [7].

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 mainly feature suggestions and enhancements.

When we developed eConference ver. 2, porting our tool from JXTA to XMPP, we had lost some features (namely file- and browser-sharing), because they could not be easily adapted, but had to be rewritten from scratch. From this drawback we realized that we wished to avoid all the effort spent in adapting the tool to support another communication platform. 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 evolving eConference from a simple collaborative application to a full-fledged collaborative platform. Our intention was to have a platform that offered as core functionality a reliable, extensible, and scalable messaging framework, on the top of 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 decided to build another prototype exploiting the Eclipse Rich Client Platform (RCP) [31]. Since the release of version 3.0, Eclipse has evolved to become an open and fully extensible framework for developing rich client applications. Although mostly known as a powerful Java IDE, now Eclipse is actually a universal plug-in platform for creating other platforms. Eclipse RCP is a pure-plugin system and, hence, fully extensible by architectural design. This new modular architecture looked very attractive to us because it promised to help us in developing with a focus on modular functionality and writing new plug-ins for missing functions. In traditional plugin architectures plugins are mere add-ons that extend the functionality of a host application, i.e., binary 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 functionality. Indeed, every application behavior is provided by a federation of plugins orchestrated by the engine [3].

The ver. 3 of eConference is a rich client application, built upon Eclipse RCP. Besides all the benefits that come from using native widgets, our tool inherited all the capabilities of the RCP, in terms of extensibility and classical concepts from the Eclipse world, like views (i.e., UI widgets) and perspectives (i.e., the particular arrangements of views in the application windows). 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 have developed an abstract network layer that exposed the core communication features, which have to be mapped onto concrete network backends. If the mapping cannot be completed for a given protocol, it means that the protocol does not guarantee the minimal requirements needed. We used again the SMACK library because we were already familiar with its event-based model, which easily fit into our architecture.

The eConference ver. 3 has been developed incrementally, using a story-driven agile process [12]. We started building a feature (i.e., a collection of plugins in Eclipse terminology) to provide instant messaging and presence awareness capabilities, which are both at the core of XMPP and, thus, the mapping was almost effortless.

Then, we extended the existing feature to implement multi-user chat for reliable group communication. Unlike presence and instant messaging, multi-user chat is not a core functionality of XMPP. Instead, it is available as a XMPP Extension Proposal (XEP). The Jabber Software Foundation develops extensions to XMPP through a standards process centered on XEPs.

The Multi-User Chat XEP [42] 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 public servers belonging to the XMPP federation. One limit 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.

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, when we developed new features for web-browser and presentation sharing, we built onto the existing features and plugins, and created new perspectives to optimize the arrangements of the UI views.

To implement the eConference feature, we made the agenda editable by the moderator, when the meeting is already started, and added support for one-to-one private messaging. Finally, we also implemented the item-based threads in the message board, so that all the utterances related to an item were grouped together.

6. eCONFERENCE OVER ECF

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; 2) a burdensome publish/subscribe subsystem, in which every bundle implemented the Observer pattern without taking advantage of the Event Admin Service, i.e., the Eclipse/OSGi internal mechanism for appropriately handling events dispatching in a dynamic pure-plugin environment [22], [34].

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.
The Eclipse foundation hosts an internal project meant just to address this problem for any RCP-based application. In fact, the Eclipse Communication Framework (ECF) project [17] provides rich-client applications with an abstract communication layer that can replace the whole network infrastructure layer. The ambitious goal of ECF is to introduce within the Eclipse platform typical collaborative services and features (e.g., instant messaging, white-boarding), bundled as set of plugins that can be reused by any RCP-based applications. Such components include core API definitions, graphical user interface widgets, and interfaces for specific network protocols. The ECF core includes an extensible framework, the SharedObject API, which provides a way for sharing data at application-level, without having to bother with protocol-specific details. The other notable components, available in ECF, include the Presence API, which handles the presence events, the File Transfer API, for sharing content between remote users, and the Remote Services API, which provides a RPC-like mechanism for remote procedure calls. All these APIs provide a high-level abstraction layer that promise to enable ECF-based applications to support multiple protocols wholesale, ignoring any implementation detail, which is 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, and Skype), although, being already an IETF standard, XMPP is the most stable and advanced provider.

Hence, we decided to develop the fourth version of eConference using ECF to replace the communication layer and relieve us from the burden of maintaining an abstract network layer to cope with future evolutions.
6.1. Porting to ECF vs. developing from scratch

When we were starting the development of eConference ver. 4, we thought to have two alternative solutions, i.e., 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 not to be feasible for the proper adoption of ECF [10].

One of the aspects we overlooked when we decided to adopt 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 API’s, visual components, and model objects. The integration the frameworks’ specific representations of the real-world components is a common problem, already acknowledged by previous research and referred to as the “overlapping principle” [30]. This problem is generally tackled through multiple inheritance, which typically causes an increase of complexity, other than being not supported by Java.

Thus, due to the larger than expected impact of adopting ECF, the efforts of porting and redeveloping were almost equivalent, since only a limited portion of GUI could be retained. Hence, we decided to rewrite the application from scratch, building upon the ECF API and services, and reusing the existing GUI code where possible.

Table 3 shows the main features and the associated graphic components available in
eConference ver. 3, and whether they could be fully replaced by adopting ECF. The fourth version of eConference (see Figure 2) 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).

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

Figure 2. The architecture of eConference over ECF (ver. 4)

7. OPEN ISSUES

While evolving eConference we found some issues that have been already acknowledged by previous research, but are still open to date.

7.1. Complexity on server side Vs. 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 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 ver. 2 and 3 a problem arose when synchronizing clients in case of latecomers or unintentional disconnections. The multi-user chat extension ensures persistency, 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 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 these events to be saved in the history, thus limiting the size of history to be stored and propagated. However, this fix, although easy to implement, only alleviated the issue. Instead, according to the XMPP philosophy (i.e., to move the complexity away from the client), to completely overcome it, the synchronization problem 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 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 time required. To be accepted, any new extension proposal has to go through the XEP standards process, which involves 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.

7.2. 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 [36]. 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 size of eConference raised from less than 1 Megabyte in ver. 2 to about 9 Megabytes in ver. 3, where the custom plugins developed, plus all the other third-party libraries we used (e.g., SMACK), accounted for only 980 Kilobytes. This limitation of Eclipse RCP is already acknowledged [18] and the Eclipse community has been working to reduce the minimal set of libraries needed.

With respect to dynamic memory overhead, instead, no comparison can be made because ver. 2 used the pure-Java Swing toolkit for building the UI, whereas ver. 3 used the SWT library, which uses native widget when possible, thus granting in general a smaller memory footprint.

8. Lessons learned

After four generations of eConference, we learned a few lessons, which may be of help when making critical architectural decisions about frameworks adoption.
8.1. Stability as a key aspect

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 systems), it failed at delivering 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 without using a client/server-like approach.

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 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 [14]. Determining the right set of narrow interfaces for developers to use has been acknowledged as a factor affecting the success of application frameworks [36]: If key interfaces are not stable, developers may have difficulty understanding and applying the framework effectively and efficiently.

As framework consumers, we did not expect the JXTA API to change often and we assumed the platform not to have backward compatibility issues as well. Although the importance of
keeping stable an application framework seems a rather obvious lesson, our experience shows that it was not so obvious to the framework producers.

8.2. Beware of reputation attribute in framework selection

Although there have been a number of formal models for the selection process of application frameworks [13], and more generally of COTS products [32], on-field studies[41] have found that developers seldom follow any formal selection process. Familiarity with frameworks and the reputation of frameworks, as well as 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. However, both attributes can lead to suboptimal choices, but whilst framework familiarity can flatten the learning curve, reputation can instead be totally misleading.

The first lesson taught us that, although prototypes can be useful to spot 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 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 lead 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.

8.3. 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 relieved 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 changes in the network layer could be avoided altogether. When you adopt a framework, you effectively outsource a portion of your software development and, consequently, developers will not be responsible for maintenance except to the extent that they are developing extensions or plug-in components for the framework [20].

9. 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 provided no facilities that allow for change also at the level of the communication protocol to be employed. Hence, in the fourth release we adopted ECF, a networking framework which 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 adoption 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.

ACKNOWLEDGMENT

This work has been partially supported by the 2006 Eclipse Innovation Award and the MiUR-Italy under grant 2006 “Metamorphos”. We wish to thank all the 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

Figure 1

Collaborative workbench

**Agenda**

- Epic 1 - See buddies status and send IM
  - *A user can check the online status of contacts in the roster*
  - *A user can start an IM session with online contacts*

- Epic 2 - Create and join a chat room
- Epic 3 - Organize and join an eConfer

**Who’s on**

- Fabio
- Filippo
- Mario (SCRIBE)

**Message Board**

- Fabio > Epic 1 can be split into two stories of smaller size:
  - *A user can check the online status of contacts in the roster*
  - *A user can start an IM session with online contacts*
  - *A user we all agree on this*

- Filippo > Just a Note: An IM can be sent also to offline contacts, as XMPP server usually store messages sent to offline recipients

**Edit Panel**

- Epic 1 can be split into two stories of smaller size:
  - *A user can check the online status of contacts in the roster*
  - *A user can start an IM session with online contacts*

- Epic 2 can be split into 3 stories:
  - *A user can create a chat room*
  - *A user can join a chat room*
  - *A user can send invitations to people for i*

**Input Panel**

**Presence Panel**

**Hand raising**

<table>
<thead>
<tr>
<th>Id</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mario</td>
</tr>
</tbody>
</table>

- **Reject**
- **Approve**

**Hand Raise Panel**

Click here to download high resolution image
<table>
<thead>
<tr>
<th>Version</th>
<th>Release date</th>
<th>Estimated Change Impact (compared to previous release)</th>
<th>Actual Change Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 build 49b</td>
<td>2002/02/08</td>
<td>Low</td>
<td>Non-breaking</td>
</tr>
<tr>
<td>1.0 build 65e</td>
<td>2002/07/08</td>
<td>None</td>
<td>Non-breaking</td>
</tr>
<tr>
<td>1.0 final</td>
<td>2002/09/24</td>
<td>None</td>
<td>Non-breaking</td>
</tr>
<tr>
<td>2.0</td>
<td>2003/03/01</td>
<td>High</td>
<td>Breaking</td>
</tr>
<tr>
<td>2.1</td>
<td>2003/06/09</td>
<td>Low</td>
<td>Breaking</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2003/09/16</td>
<td>None</td>
<td>Non-breaking</td>
</tr>
<tr>
<td>2.2</td>
<td>2003/12/15</td>
<td>Medium</td>
<td>Breaking</td>
</tr>
<tr>
<td>2.2.1</td>
<td>2004/03/15</td>
<td>Medium</td>
<td>Breaking</td>
</tr>
</tbody>
</table>

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)
### Table 2. Alternative JXTA pipe services evaluated

<table>
<thead>
<tr>
<th>Pipe service</th>
<th>Since</th>
<th>Type</th>
<th>Needs a server for group communication</th>
<th>Reliability ensured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicast</td>
<td>v 1.0</td>
<td>1-to-1</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Secure</td>
<td>v 1.0</td>
<td>1-to-1</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Propagate</td>
<td>v 1.0</td>
<td>1-to-M</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>v 1.0</td>
<td>1-to-1</td>
<td>Yes</td>
<td>Yes (v 2.3+)</td>
</tr>
<tr>
<td>JXTA Socket</td>
<td>v 2.0</td>
<td>1-to-1</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Component</td>
<td>Available in eConference 3</td>
<td>Provided by ECF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>----------------------------</td>
<td>-----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message board*</td>
<td>Partially</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Roster View</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extension Points API</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hand Raising</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White board</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conferencing Events Manager (invitations, reminders, …)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Account creation / Login Manager</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Does not support multiple discussion threads