RZ 3608 (# 99618) 04/28/05
Computer Science 8 pages

Research Report

Application Integration Using Instant Messaging Based Web Services

Carl Binding and Christian Hörtnagl

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{cbd,hoe}@zurich.ibm.com

Ted Bonkenburg and Benjamin Reed

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Application Integration Using Instant Messaging Based Web Services

Carl Binding
and Christian Hörtnagl
IBM Zurich Research Laboratory
Email: {cbd,hoe}@zurich.ibm.com

Ted Bonkenburg
and Benjamin Reed
IBM Almaden Research Laboratory
Email: {tedbo,breed}@almaden.ibm.com

Abstract

This paper explores the usage of Web services over an instant messaging protocol to integrate structured instant messaging into user-facing applications. The work is based on the premiss that conventional usage of instant messaging protocols is primarily based on exchanging hunan-readable textual messages, but that instant messaging offers persona related, real-time, addressing which can be explored to exchange structured data information between user-facing applications. We describe an architecture which allows interactive applications to use a deployed instant messaging environment for the exchange of user initiated, structured dialogue data through the use of Web services. One of our sample applications extends the Eclipse workbench and enhances joint development by enabling developers to monitor, in real-time, resources of their partner's environment.

I. Introduction

Instant messaging has become a widely used and vastly popular communication technology in addition to more traditional means of communication, such as letters, fax, e-mail, or short messaging services. End users register with the instant messaging environment and are enabled to instantaneously communicate with peers (sometimes coloquially referred to as "buddies") using dedicated windows for textual input and output.

A *presence service* is used as a central registration point at which subscribing end-users register their presence. Other, befriended end-users can be alerted to the presence of newly registered participants, using the *presences subscription* and *notification* services. Thus, end-users become aware of potential, on-line, communication partners and may engage in actual instant messaging or *chat* sessions.

Such systems have found wide-spread acceptance in the inter- and intranet community with commercial offerings from Lotus [6], America Online [20], and Microsoft [21], as well as research projects [13].

Instant messaging services exhibit the following characteristics:

- *Text based*: the various instant messaging services currently are text based. The entire dialogue consists in textual, in-lined information limited by the character set in use.
- *Instantaneous*: expressed in the technology's naming, communication is real-time and quasi simultaneous. Text input is collected until a line is completed and transmitted to one or multiple communication partners.
- *Multi-party*: chat sessions can include multiple communicating users. The input of one party is propagated to all other parties and becomes visible quasi simultaneously to all of them.
- *Persona centric*: chat partners are people, identified by some (nick) name which is independent of the physical machine used by the chatter. The central chat server's identity and the chat communities are configured into the chat applications by using the network name of the chat proxy's host and the (nick) name of the chat community respectively.
- *Presence*: the central chat proxy registers the presence of newly joining members and propagates their presence to other members of the chat community[1]. Thus, the indication of a member's presence implies his or her current reachability and mode of operation.
- *Reachability*: chat applications can cross networking firewalls between connected hosts, if the firewalls are configured appropriately. Transporting additional information to invoke Web services over such a channel guarantees instanteneous reachability, at the cost of potential security risks.

The premiss of this paper is that the use of instant messaging systems can be expanded by incorporating structured information items into the dialogue between end-users. Furthermore, we explore the use of standardized, open, middleware - namely Web services [31], [32] - to augment text-oriented instant-messaging applications. This differentiates our approach from others, ad-hoc, extensions of instant-messaging infrastructures with application specific functionality [5], [26].

Structured information exchanged over our messaging infrastructure originates from specific applications and is collected during an ongoing chat conversation. It is then sent to a chatter's partner via Web services over the instant messaging infrastructure and propagated into an application specific data handler which uses standardized Web services functions to

[1]Users can express the set of members for which they wish to receive presence notifi cations.

service an incoming request. As examples of this scenario, consider the following use cases.

A. Shared Editing

Modern programming environments such as Eclipse [8], [24] or Microsoft's Visual Studio [23] support the creation and management of *projects* of diverse natures such as various Web applications styles or more conventional application programming in languages such as Java, C#, C, or Visual Basic. Such projects map to directories containing their diverse *resources* i.e. configuration files, source code files, markup information, etc.

Programming workbenches often also include facilities for shared program development. Project resources can be committed to a centralized, shared data repository from which other team members can update their environment with the latest development releases. One example of such a shared source code control system is the well known Code Versioning System (CVS) [14]. Whilst this sharing of resources leads to a well coordinated cooperative model, extreme programming situations [2] between members of a geographically dispersed team warrant tighter and more real-time co-operation. Systems like the Eclipse extension described here and others, such as the JAMM [3] and Jazz [5], intend to address these needs.

In our shared editing application scenario, we have integrated some of Eclipse's resources observation and differencing features into the chat environment: one Eclipse user can find out about the resources of other Eclipse users registered within the instant messaging environment. He can be notified of changes in resources or perform a quick-diff[2] against a resource owned by his chat partner but of common interest.

B. Shared Calendaring

Electronic calendars are an essential part of today's electronic office environments [7], [22]. Users can schedule meetings, set reminders, and may share their calendars with a pre-determined number of colleagues, friends, or relatives. This sharing is done using the native GUI features of the calendaring application.

Specifically, we can envision the following features for calendar sharing across instant messaging:

- Transmission of calendaring information over a chat session: if user *A* wants to display selected entries of his calendar to his chat partner *B*, he may not wish to force *B* to start up his calendar application and navigate to retrieve *A*'s entry of interest. Instead, *A* just transmits the calendar event over the active chat session and it is the chat application which renders the event.
- Setting watch-dogs on a calendar: when user *A* modifies specific calendar entries which are under observation by *B*, *B* is alerted over the running chat session.
- Scheduling a meeting: *A* may propose a meeting time to *B* who can accept or deny the suggested event and communicate in parallel with *B* over the chat session.

The above functions can partially be performed with today's electronic calendaring systems. However, the co-operation is not real-time but asynchronous in nature. User *A* may send calendaring information to user *B* via e-mail[3] but user *A* cannot infer when user *B* will receive and read the e-mail. The same argument on timeliness applies for alerts on shared events or scheduling a meeting. Integrating calendaring features into an instant messaging environment thus does improve on the timeliness of the application usage.

C. Help Desk Support

Another potential application area we have considered is sharing of computing and network management information in a help-desk situation. The help-desk assistant may engage in a chat session with the end-user and, in parallel and quasi-instantaneously, communicate with the end-user's system management application to retrieve, respectively to set, information and thus detect system problems and repair them. The key feature in this scenario is again the timeliness of the shared information, reuse of the existing instant messaging infrastructure, and the persona-centric addressing of the end-user's environment.

The remainder of this paper is organized as follows: section II details the system architecture and describes the programming APIs. Section III describes the realization of the Eclipse application and section IV accounts experiences with extending a calendar application. Some related work, mainly concerned with collaborative programming environments, is reviewed in section V. Finally, section VI comments on the lessons learned and potential impact of our approach.

II. Architecture

This section describes our approach to augment an instant messaging protocol with Web services capabilities. We give a brief review over Web services technology, instant messaging architecture, and the XMPP [10], [25] protocol as one particular example of an instant messaging protocol. We then proceed with the description of our architectue to use the instant messaging infrastructure for Web services enablement.

A. Web services

Web services are an XML [30] and – originally – an HTTP [15] based technology allowing to interchange messages between computing entities. The message format is defined in [31] and the Web Services Description Language (WSDL) [32] defines *data types* (using XML Schema [33]), *messages*, *service types*, *bindings* of service types to protocols and messaging formats, and service end-points, called *ports*.

The WSDL interface defines the external signature of a given service, its implementation can use diverse kinds of programming and execution environments and languages. The

[2]QuickDiff is an Eclipse feature which summarily differentiates a workbench editor's buffer against some reference of the same resource and highlights the differences in the editor window.

[3]By either including the event or a reference to the event in the e-mail message.

independence between a specific language and run-time and the externally visible service specification is key to the interoperability aspects of the technology: heterogeneous infrastructures are enabled to interact via Web services. In addition, running SOAP over HTTP enables inter-company reachability of Web services across enterprises' firewalls.

Fundamentally, the technology sends well-formed XML datagrams over a given protocol from one end-point to another. Thus, usage patterns such as remote protecure call (i.e. request – reply) and one-way messaging are made available for communicating entities.

Depending on the message, a receiving end-point triggers the execution of some computing function, consuming the message's payload data. The use of XML as a data modeling language enables the receiving entity to process the message data using widely available XML tools and thus conveniently supports the interchange of semi-structured data entities.

B. *Instant Messaging*

The end-user level functionality and key features of instant messaging systems have already been given in section I. Such systems adhere to an architectural model for presence and instant messaging defined in [12]. So-called *presentities* register with a centralized *presence service* which in turn notifies a set of registered *subscribers*. The centralized service approach is also advocated for the *instant messaging service* in which sender and receivers of messages communicate over the centralized instant messaging proxy. Commonly, the functions of the presence service and instant messaging services are merged into one logical service, called the *service proxy*. Figure 1 illustrates this model.

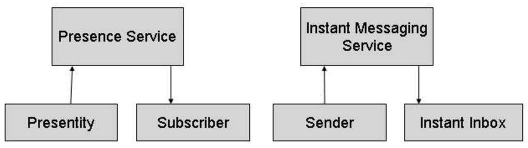


Fig. 1. Model for Presence and Instant Messaging

The Extensible Messaging and Presence Protcol (XMPP) [25] establishes an XML stream acting as envelope between a client and the proxy. Embedded in that stream, server and client exchange *XML stanzas* which are well-defined XML constructs to describe presence notifications, messages, and information queries. During establishment of the stream, authentication and authorization can be performed by either using Transport Layer Security (TLS) [9], [27] or the XMPP specific Simple Authentication and Security Layer (SASL) protocol.

To transport SOAP formatted XML content, we use the XMPP *information query* (`<iq>`) stanza as proposed in [17]. The `<iq>` stanza allows to specify sender and destination of the stanza and we use the *type* field to distinguish between the entity requesting a web service and the response to the request[4]. On the sending side, the sender embeds the SOAP-XML envelope, containing any headers and the SOAP body, into the `<iq>` stanza and on the receiving side this information is extracted, parsed, and propagated to the Web services handling code.

C. *Combining instant messaging and Web services*

Our architecture is shown in figure 2. We use the chat application running on the end-user's machine as a proxy for traffic between *chatlets*. Chatlets communicate with the local chat using local sockets. The local chat embeds the received SOAP message into an XMPP `<iq>` stanza and forwards it to the chat proxy which forwards it to the receiver's, remote, chat. The latter forwards the message to the target chatlet as appropriate, again using local sockets.

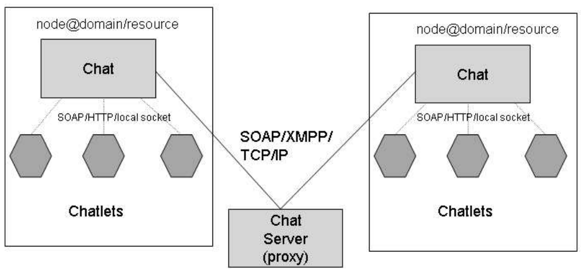


Fig. 2. System architecture

The routing and correlation of requests and replies relies on the use of WS-Addressing [1], which provides header elements to indicate the destination address (`To`), the origin address (`From`), message identity (`MessageId`), and a relates-to (`RelatesTo`) element to correlate a reply with an original request when an asynchronous transport protocol, such as XMPP, is used for Web services `in-out` message exchange patterns[5].

Chatlets register themselves with their (local) chat proxy, which propagates their presence to other, subscribing, chat proxies – and thus users belonging to a particular chat community – using the instant messaging infrastructure. The (remote) chat proxy then forwards the presence of remote chatlets to identically typed chatlets[6].

When a new user goes on-line, his local chat proxy emits a presence notification to other chat proxies. These reply with their presence and query the chatlets of the new, on-line, chat proxy.

The protocols for chatlet de-registration and chat proxy termination perform the de-registration of chat proxies and chatlets.

All of these protocol operations are implemented as Web services provided by the chat proxies. Presence notification

[4]Concretely, we use the *type* values `set` and `result`.

[5]Although this message exchange pattern does not strictly imply synchronicity, we primarily focus on synchronous replies to a given request.

[6]Currently, we simply use name equality to assert chatlet type equality; a functional signature based approach could be envisioned.

and detection are provided by the instant messaging infrastructure; these trigger execution of protocol specific operations.

The protocol can be summarized as follows:

- Chat to Chat:
  - Query (remote) registered chatlets: when notified on the presence of a joining chat proxy, the notified chat queries the notifying chat about its set of registered chatlets.
  - Advertise/withdraw local chatlet: when a new chatlet registers, respectively unregisters, with its local chat proxy, the latter propagates this information to remote chat proxies.
- Chatlet to Chat:
  - Register/unregister local chatlet: when a newly instantiated chatlet registers with its local chat proxy, it also obtains the instant messaging identity of the chat proxy to complete its own address within the chatlet address space. Registration triggers the chat to chat advertisement operation.
- Chat to Chatlet:
  - Advertise/withdraw remote chatlet: to forward registration and de-registration events to registered local chatlets.
- Chatlet to Chatlet:
  - Application level services: these are arbitrary SOAP messages addressed from one chatlet to another chatlet. Chatlets are identified via URIs [16], [25] which include the identity of the chat proxy and the identity of the chatlet. (We use the proposed Jabber URI format `jabber://node@domain/resource` to identify the chat proxy which we augement with a URI query parameter `chatlet`.) Once a SOAP message is received by a chatlet, it can further dispatch or otherwise handle the message before sending the reply.

With regards to the application layer interface, chatlets may use operations which provide for the initialization and proper termination of a chatlet's life cycle. The application may also implement a call-back interface used to dispatch received application level SOAP messages to the application layer and provding following methods:

- `handles`: this method is called by the message dispatcher with a message's WS-Addressing `To` element value and the callee indicates whether it wants to accept messages addressed to the given target.
- `getWSAActions`: if the client handles a specific destination address, it can further indicate which actions it is prepared to handle. A wildcard operator indicates that the client handles all operations addressed to it.
- `received`: when the dispatcher has determined that a particular client handles an incoming message, it uses the `received` callback, passing a SOAP message structure as argument and expecting a SOAP reply message in return[7].

The astute reader may have observed that our protocol principally serves one purpose: to exchange communication end-points between communicating entities across the instant messaging infrastructure. Beyond the propagation of end-point presence we also exchange application level Web services traffic across the instant messaging infrastructure. However, a direct, TCP communication channel - bypassing the instant messaging proxies and server - could be envisioned. We rejected this approach as it would violate our persona-centric addressing paradigm, raises issues with firewalling, and complicates the appropriate handling of chat parties leaving the instant messaging community.

III. Extreme programming support for the Eclipse workbench

The Eclipse workbench [24] provides a framework to build development environments and relies on the use of so-called *plug-ins* to integrate new functionality into the workbench. The environment defines a set of *extension points* which are pre-defined by the workbench[8] and which are associated with corresponding plug-ins. When execution reaches an extension point, the code bound to related plug-ins is executed. The call interface is defined by individual extension-point types and the plug-in code must implement these.

We have used Eclipse's extension mechanism to associate a workbench instance with a user's chat environment and to allow the end-user to interact with remote Eclipse workbenches across the instant messaging environment. Concretely, following features are supported:

1) *Select a partner workbench*: The Eclipse workbench behaves as a chatlet of our architecture in section II. When starting an instance of the workbench, it registers with the local chat proxy. The local chat proxy propagates this information to remote chat proxies and queries these about instances of (remote) Eclipse chatlets. These are transmitted to the new Eclipse workbench chatlet, identifying potential interaction partners. The end-user then selects one of the remote Eclipse chatlets for further operations (see also figure 4).
2) *Subscribe to change events of selected resources*: Having selected one specific, remote, Eclipse chatlet, the user can query its set of resources. Resources are Eclipse abstractions to model projects, their contained folders, and their file resources. Eclipse provides an internal listener mechanism to detect changes in resources; for instance saving a particular file resource is mapped onto a change event of the file resource, its container resource, and the containing project resource.

   When the end-user selects a partner's workbench resource for change observation, he is notified whenever the remote resource is saved. Our plug-in code detects the Eclipse internal change event and propagates it to

[7]The data structures used are modelled after [29].

[8]However, extensions can in turn define further extension-points.

remote Eclipse chatlets. On the receiving side, we create a pop-up to inform the user about the change of the observed resource.

3) *Show observed resources*: The workbench keeps track which remote resources are observed and thereby allows the end-user to cancel an observation.
4) *Quick-diff against a remote resource*: The Eclipse workbench provides a quick-diff feature which summarily compares the content of an editor's buffer against a reference instance of that resource, in general the disk-saved file corresponding to the resource. We have augmented Eclipse by extending the quick-diff mechanism to use resources of the selected partner workbench. That is, if user A has selected user B's workbench, A can enable a quick-diff of his currently active editor buffer against B's corresponding resource[9].

The GUIs of the instant messaging extensions for the Eclipse workbench are shown in figures 3 and 4. The first figure shows the Eclipse window's menu extension for the Jabber Plugin Extension. The first panel in figure 4 allows the user to select one particular chat partner also running the Eclipse chat extension. The middle panel displays the resources of the selected partner and the bottom panel lists the set of remote, observed resources.

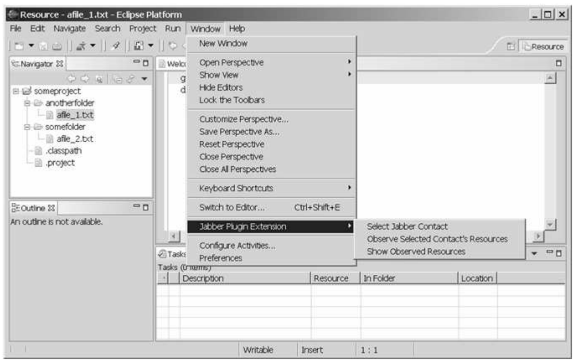
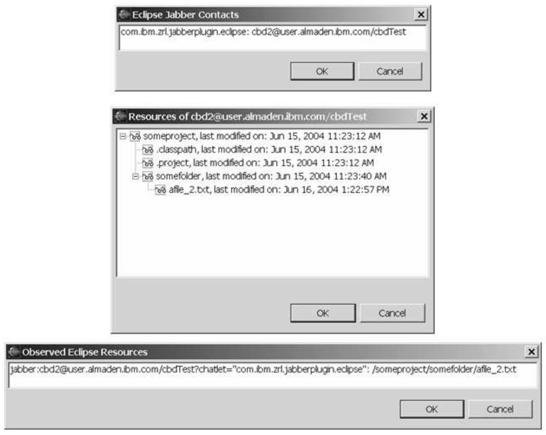


Fig. 3. Eclipse Application User Interface

Implementation of these features relies on Eclipse's internal architecture for resource representation, event listening, and extensibility mechanism described above. The plug-in in turn uses the chatlet abstraction of our architectural model in section II to register with the instant messaging environment and to communicate with other registered Eclipse workbench chatlets. Application level functions are invoked using Web services across the instant messaging channel and propagated to the Eclipse plug-in via the call-back interface described in section II. Figure 5 illustrates the interplay between Eclipse's internal data structures and our chatlet based architecture. Our implementation is based on JabberBeans, a Java package handling the Jabber/XMPP protocol [34]. For the chatlet application, we provide a Java class which handles the protocol with

[9]If B does not maintain a similarly named resource, no differencing operation is performed.


Eclipse Jabber Contacts
OK
Cancel
Resources of cbd2@user.almaden.ibm.com/cbdTest
OK
Cancel
Observed Eclipse Resources
OK
Cancel

com.ibm.zrl.jabberplugin.eclipse: cbd2@user.almaden.ibm.com/cbdTest

someproject, last modified on: Jun 15, 2004 11:23:12 AM
.classpath, last modified on: Jun 15, 2004 11:23:12 AM
.project, last modified on: Jun 15, 2004 11:23:12 AM
somefolder, last modified on: Jun 15, 2004 11:23:40 AM
afile_2.txt, last modified on: Jun 16, 2004 1:22:57 PM

jabber:cbd2@user.almaden.ibm.com/cbdTest?chatlet="com.ibm.zrl.jabberplugin.eclipse": /someproject/somefolder/afile_2.txt

Fig. 4. Eclipse Application User Functions

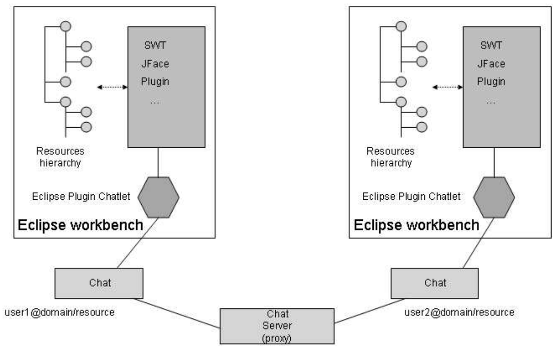


Fig. 5. Eclipse Application

the local chat proxy which in-turn interacts with the Jabber environment. The local chat proxy code has been integrated into [13].

We have currently not implemented any explicit security features into the system. Following considerations, however, can be made related to security and privacy concerns:

- Data exchange can be made secure using WS-Security [19] to secure the traffic between involved chatlets. Hence, no additional support beyond WS-Security needs to be considered.
- Participating in a chat session and enabling presence detection by others implicitly reveals some information about a chat participant. However, it is unclear to which extent a chatter's resources are made available to others. An access control list [28] based authorization scheme would be the appropriate and straightforward solution to manage finer grain access authorization and enable the owner of Eclipse resources to grant access to other chat participants.

Additional end-user functionality for cooperative programming can be envisioned. Expanding the current quick-diff feature into a richer viewing and merging capability which allows for deeper sharing of resources of common interest amongst members of a programming team would be an evident extension of the current prototype.

IV. Integration with a calendar environment

To implement the functionality suggested in I-B, we have chosen the IBM Lotus Notes calendar environment [7] and implemenetd the basic architecture of section II. However, in comparison with the Eclipse application, the Lotus Notes environment does not provide a similarly flexible and powerful extension mechanism. The calendaring chatlet is thus a free-standing Windows application implemented in Java which uses the Lotus Notes APIs [18] to access the calendar's database, as illustrated in figure 6. Data sharing between the chatlet and the application is done through the Lotus Notes calendar database and not via in-(virtual)memory shared data. The Lotus Notes application launches the calendaring chatlet as an independent operating system application using Notes' `Excecute` command. We currently only have implemented sharing of

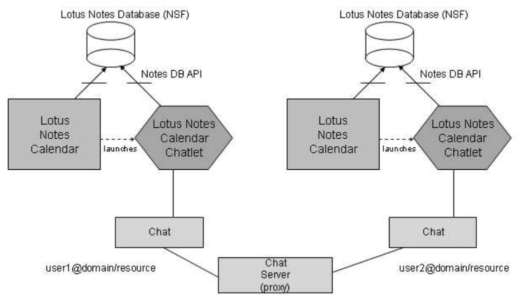


Fig. 6. Calendaring Application

calendaring information, where user *A* can send information from his calendar to a chat partner *B*. The triggering of this exchange is done from the Notes Calendar GUI which launches the Windows application in the background. This application, implementing the *chatlet* functions, accesses the Notes calendar database, extracts the relevant information, and forwards it to its pendant chatlet on user *B*'s chat environment where it is displayed.

The (structured) calendaring data exchange is based on the Web services over instant messaging protocol described in section II above. Message dispatching, handling, and replying is identical to the Eclipse application. The main differences between the two applications lies in the difference of integration and extensibility of the core application environment. Whereas in the case of the Eclipse workbench we could use its extension points mechanism, we could not as conveniently integrate the chatlet functionality into the Lotus Notes calendar environment. The provided scripting mechanisms for Lotus Notes indeed do not provide the powerful and rich environment of a full fledged Java environment as is the case for Eclipse, leading to the application architecture shown in figure 6.

Whilst the differences of the integration depth is not surprising given the two environments, it raises the more fundamental question of application extensibility. This issue also comes to play for the extensibility of the chat application itself when dealing with the issue of integrating the various chatlet GUIs into an already existing GUI. The choice there is threefold:

1) Integration with the core application: This is the approach we have chosen for the Eclipse application. It appears to be the most natural approach since first the end-user functionality of the chatlet is closely linked with the core application and secondly since Eclipse provides for convenient extension mechanisms. However, it depends on the openess of the application for extensions.
2) Integration with the chat application: We have rejected this approach based on the previous argument. First, the chatlet end-user functionality is more application specific than chat generic. Second, the chat application we used did not provide extensions mechanisms which strongly suggested using it for GUI extensions.
3) Free-standing application: In this case, the chatlet runs as a free standing GUI enabled operating system level application and represents the solution when the application does not provide convenient extensions mechanisms as has been the case for our calendaring scenario. Whilst this approach leads to workable results, it forces re-implementation of many function points which already are implemented in the application.

It may be worthwhile to further explore the extension mechanisms as examplified by Eclipse and their general applicability to other end-user applications such as word-processing, spread-sheets, mail programs, or calendaring.

V. Related Work

An up-to-date survey on collaborative development environments (CDE) is given by Booch and Brown [4]. The current state of the art is surveyed and the salient requirements for valuable CDEs are outlined. Our system only offers a rudimentary set of these CDE features, however our aim was not to implement a full-fledged CDE but to explore and demonstrate the use of an instant messaging system to carry Web services traffic; a technology which can then be used to augment existing IDEs into CDEs.

Shared programming specifically within the Eclipse environment has been explored by other researchers. Cheng et al. [5] describe an extensive feature set which allows to manage and enable cooperation within small scale programming groups. Their system supports the initiation of structured chat sessions from within the Eclipse workbench, allows to share the Eclipse screen [11], to take part in a shared white-board service, and - similar to our prototype - to indicate editing activities on some partner's Eclipse resources.

The Jazz work concentrates primarily on providing and exploring new collaboration paradigms and how to apply these to joint programming; whereas our work is more concerned with the re-use of existing infrastructures to implement novel user features. We have not extended the instant messaging capabilities to support shared programming. Instead we expand the integrated development environment's features using the unmodified instant messaging environment to locate communication partners and provide communication channels.

The JAMM project [3] also addresses the issue of collaboration by instrumenting the Java widget library to provide a more flexible interaction model for collaboration. By reimplementing key widget classes, JAMM enables collaboration even in applications that were not written with collaboration support, much like the screen sharing scheme used by Jazz. Because the sharing is implemented at the widget layer, JAMM has the advantage of not confining collaborators to the same view of the application. For example, one collaborator can be working on some methods of a code module while his JAMM partner may be working on an unrelated feature of the same module. Visual clues are used to indicate what each collaborator is doing.

Our approach taken together with Jazz and JAMM illustrates a continuum of collaboration models. Jazz offers very tight collaboration where the application itself is not collaboration aware and the collaborators work in a follow-the-leader fashion using a collaboration specific middleware. In the middle of the continuum we have JAMM which transparently modifies the application by replacing key widget classes and allows the collaborators to work in a more independent fashion. Our work is on the far end of the continuum where we had to modify the application code but not to adapt the collaboration middleware and we allow collaboration even if the collaborators work on different document resources. For example, in Jazz two programmers can tightly work on a function in a source module together. In JAMM the same programmers can work on different functions in the module. Our project aims to let the programmers work on different versions of the same source module and share different parts of the module. It may be that the programmers are prototyping different ways of addressing a problem but want to share parts of each others changes.

Another piece of related work is described in [26]. Kaegi's work expands the Eclipse workbench very similarly to our approach: one user obtains information on another user's Eclipse resources and visualizes these. Unlike [5], there are no enhanced chat features, nor a shared white-board, or screen sharing. Kaegi uses a proprietary XML based protocol and, by not relying on an instant messaging infrastructure's presence capabilities, the system must provide its own partner location and detection mechanisms.

VI. Conclusion

We have shown the feasibility of carrying Web services traffic over an instant messaging infrastructure. Thereby we have enabled novel application functionality in which we open end-user application's data spaces to chat-like interactions. This allows to support quasi instantaneous information exchange of complexly structured application data - represented in XML. The use of an instant messaging infrastructure provides us with a firewall crossing communication channel to exchange Web services traffic, a persona centric addressing scheme, and the means to detect communication partners within the chat community.

The Web services paradigm allows to exchange application level, structured data and provides quasi-instantenous data exchange in a flexible, yet standardized way. Thus well-known procedures of Web services usage and data processing can be re-used.

Our work has focussed on providing the underlying architecture, its protocols, and to explore application paradigms. We have not specfically addressed performance issues. However, we have observed that repetitive message handling within different entities may cause performance issues, due in particular to the XML parsing needed to analyze XMPP stanzas and their payload. Care must be taken to avoid parsing the full message body when only header elements are needed for proper routing of the message.

An important differentiator in integrating instant messaging into user-facing applications has been the extensibility of the application itself. As argued above, a powerful and flexible extension mechanism supports reuse of application internal data and GUI features. The investigation of generic application extensions mechanisms and their capabilities has however not been a focus of this work.

A further facet of investigation can be integration of end-user applications which are more pervasive in their nature, running on PDAs or smart-phones. The user-centric and device independent addressability provided through the instant messaging environment, combined with the invocation semantics of Web services may very well ease and broaden inter-application integration on such platforms.

References

[1] BEA Systems Inc., International Business Machines Corporation, and Microsoft Corporation, Inc. *Web Services Addressing (WSAddressing)*, March 2004. ftp://www6.software.ibm.com/software/developer/library/ws-add200403.pdf.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[3] James Begole, Mary Beth Rosson, and Clifford A. Shaffer. Supporting worker independence in collaboration transparency. In *Proceedings of the 1997 Symposium on User Interface Software and Technology*, pages 55–64. ACM, 1997.

[4] Grady Booch and Alan W. Brown. Collaborative development environments. *Advances In Computer Science*, 49, 2003.

[5] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazzing up Eclipse with Collaborative Tools. In *OOPSLA 2003 - 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 45–49, Anaheim, CA, October 2003. ACM SIGPLAN. Eclipse Technology Exchange Workshop.

[6] IBM Corporation. IBM Lotus Instant Messaging and Web Conferencing. http://www.lotus.com/products/product3.nsf/wdocs/homepage.

[7] IBM Corporation. IBM Lotus Notes. http://www.lotus.com/products/product4.nsf/wdocs/noteshomepage.

[8] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–381, 2004.

[9] T. Dierks and C. Allen. *Transport Layer Security (TLS)*. Internet Society, January 1999. http://www.ietf.org/rfc/rfc2246.txt.

[10] P. Saint-Andre (ed.). *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. Internet Society, December 2003. http://www.ietf.org/internet-drafts/draft-ietf-xmpp-im-20.html.

[11] Constantin Kaplinsky et al. TightVNC Software. http://www.tightvnc.com/.

[12] M. Day et al. *A Model for Presence and Instant Messaging*. Internet Society, February 2000. http://www.ietf.org/rfc/rfc2778.

[13] Marc Eisenstadt et al. BuddySpace Instant Messenger. http://kmi.open.ac.uk/projects/buddyspace/.

[14] Per Cederqvist et al. *Version Management with CVS*. Signum Support AB, cvs 1.11.2 edition, 1993. https://www.cvshome.org/.

[15] Robert Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. IETF Networking Group, January 1997. http://www.faqs.org/rfcs/rfc2068.html.

[16] T. Berners-Lee et al. *Uniform Resource Identifier (URI): Generic Syntax*. Internet Society, August 1998. http://www.ietf.org/rfc/rfc2396.

[17] Fabio Fornio. JEP-0072: SOAP Over Jabber. http://www.jabber.org/jeps/jep-0072.html.

[18] IBM. *Lotus Domino Toolkit for Java/Corba*, 2.1 edition, January 2001. http://www-130.ibm.com/developerworks/.

[19] IBM, Microsoft, VeriSign. *Web Services Security (WS-Security)*, April 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/.

[20] America On-Line Inc. AOL Instant Messenger. http://www.aim.com/.

[21] Microsoft Inc. Microsoft Live Communications Server 2003: Windows Messenger. http://www.microsoft.com/offi ce/livecomm/prodinfo/default.mspx.

[22] Microsoft Inc. Microsoft Offi ce Online. http://offi ce.microsoft.com/home/.

[23] Microsoft Inc. Microsoft Visual Studio. http://msdn.microsoft.com/vstudio/.

[24] Object Technology International Inc. *Eclipse Platform Technical Overview*, 2001. http://www.eclipse.org/articles/index.html.

[25] Internet Society. *Extensible Messaging and Presence Protocol (XMPP): Core*, June 2004. http://www.jabber.org/ietf/draft-ietf-xmpp-core-21.html.

[26] S. Kaegi. Lightweight code sharing in the eclipse environment. http://www.scs.carlton.ca/˜skaegi/cdt/index.html.

[27] R. Khare and S. Lawrence. *Upgrading to TLS Within HTTP/1.1*. Internet Society, May 2000. http://www.faqs.org/rfcs/rfc2817.html.

[28] Charles P. Pfleeger. *Security in Computing*. Prentice-Hall, Inc., 1996.

[29] Sun Microsystems. *Class SOAPMessage*. http://java.sun.com/j2ee/1.4/docs/api/javax/xml/soap/SOAPMessage.html.

[30] W3C. *Extensible Markup Language (XML)*, February 1998. Version 1.0, http://www.w3.org/TR/REC-xml.

[31] W3C. *Simple Object Access Protocol (SOAP) 1.1*, May 2000. Version 1.1, http://www.w3.org/2000/NOTE-SOAP-20000508/.

[32] W3C. *Web Services Description Language (WSDL) 1.1*, March 2001. Version 1.1, http://www.w3.org/TR/wsdl.

[33] W3C. *XML Schema Part 0: Primer*, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[34] David Waite. JabberBeans: a Java library for the Jabber Instant Messaging network. http://jabberbeans.jabberstudio.org/.