The Architecture of the Instant Messaging System I^{3}M

Hans Albrecht Schmid

University of Applied Sciences,
Brauneggerstr. 55
D - 78462 Konstanz
xx49-07531-206-631 or -500
schmidha@fh-konstanz.de

Abstract Web services are, due to the excellent tool support, simple to provide and use in trivial cases. But their use in non-trivial Web service-based systems like I3M poses new difficulties and problems. I3M is an instant messaging and chat system with distributed and local components collaborating via Web services. One difficulty is to make a series of related Web service invocations in a stateful session. A problem is the performance of collaborating collocated, service-oriented components of a system, due to the high Web service invocation overhead as is shown by measurements. Solutions to both the difficulty and the problem are proposed.

1. Introduction

Web services define an standardized interface for the invocation of services (compare [LC02]). Currently, the main interest concentrates on the RPC-like, stateless request-response service, using HTTP as transport protocol and SOAP for a definition of the XML format describing the service invocation, parameters and result.

Working with Web services is amazingly simple due to the excellent tool support. On the Java side, there are e.g. the tools Axis and Java WSDP from Sun; on the Windows-C# side, the tools are integrated with dotnet and the IIS server. You do not have to know neither HTTP nor SOAP to provide or invoke a Web service. You develop classes and let the tools generate helper classes, called proxies or stubs, and skeletons or ties, to transform method calls in Web services and inversely.

The server side (see the translation component Figure 1 right) is an HTTP-based application server. When you develop for it e.g. a C# class like Web service implementation that is derived from the class WebService, each method with a modifier ”[webmethod]” implements a single service. You may let a tool generate the WSDL (web service description language) description that describes the format of the services provided by that class.

When you put a WebService class into an ASP (advanced server pages) directory of the IIS server, a skeleton like the translation skeleton is automatically generated. It accepts at the WS (Web service) interface an HTTP request in SOAP format, transforms it and calls the matching method of the Web Service implementation

class. The result is transformed back by the skeleton to the SOAP format and sent as an HTTP response to the caller.

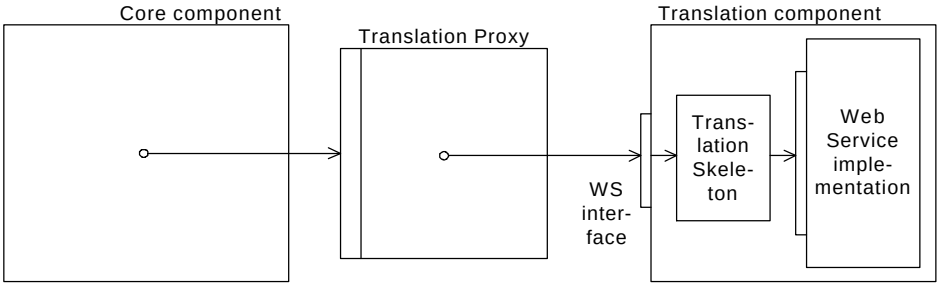


Fig. 1. Core component as client invoking a Web service via the translation proxy and translation skeleton from the Web service-implementation class

On the client side (see Figure 1 left), you need to obtain from the server the WSDL description of the Web service you want to invoke. You let a tool generate a proxy [GHJV95] like the translation proxy from the WSDL description. The proxy has a method for each service described in the WSDL. The proxy methods transform, inversely to the skeleton methods, methods calls, originating from the core component, in Web service invocations, and the result back in a method result.

Thus, it is very simple to provide a simple Web server and invoke its services from a client. But it gets more complicated if you build a system based on Web services (shortly: WS-based). A WS-based system is a distributed system that consists of a set of components collaborating by service-oriented interfaces that are formed by sets of related Web services. A client (which may be itself a system component) invokes typically not a single service, but a set of related services in a session. Distributed business applications and many other applications may be realized as WS-based systems.

Building applications as WS-based systems from components (compare [Sz97]) with Web service interfaces has considerable advantages:

1. A system may be heterogeneous, i.e. consist of different kinds of components, like JavaBeans and DCOM components written in C#, running on different platforms.
2. The flexibility of building and maintaining a system is highly increased. A component that provides a set of services may be exchanged easily against another one, be it local or remote.
3. The access to system components that were intended to be internal, may be granted to external users if required.
4. The components are location-independent: if the workload is getting to large, the components may be distributed over several computers without requiring modifications.
5. The interfaces of all system components are documented in a uniform way in a description in WSDL format.

A disadvantage is the high overhead of Web service invocation.

We present I3M, an instant messaging and chat system we developed, as an example for a WS-based system, and report on our interoperability experiences. I3M has distributed and local components running on two different platforms, which collaborate via Web service invocations.

Building our system as a WS-based system, we have identified a smaller difficulty and a problem, in addition to known problem areas like security and transaction support that are currently under work:

1. How can a set of related Web services be invoked in a session with a client-specific state, though HTTP is a stateless request-response protocol?
2. Can the performance degradation for system components collaborating via Web service invocations (which is confirmed by measurements) be neglected in the light of the advantages, in particular if the components are collocated, i.e. located on the same network node?

The difficulty and problem identified are not specific for our system, but due to the characteristics of WS-based systems, as we show. The smaller difficulty has been solved by using the heritage of application servers which provide session states with the stateless HTTP protocol already for Web applications, together with a new variant of the memento pattern, called session memento. We present a new solution for the second problem by introducing intelligent proxies that avoid the Web service invocation overhead for invocations of collocated components.

2. Requirements for Web WS-based System Architectures

A client of a WS-based system may often want to invoke a set of related services in a session. This characteristic is in contrast to the Web service characteristic that an invocation is stateless.

This statelessness is perfectly suitable for simple services. For example, if you want to know the current value of a stock, you may invoke a supposed *DAX Web service: getStockValue(SAP),* which returns the current stock value of SAP. The same holds if you want to know the current balance of your account. The *Web service: get balance* would have not only the account number as a parameter, but also the name, a password and perhaps a transaction-id for the identification of the caller.

But there is a problem if a user does not only want to know the balance, but wants to make afterwards different transfers from and to his account. It should not be required to re-send the name, password and a new transaction-id for the identification of the caller with each Web service like get balance, transfer money, etc., since the re-identification each time is costly, not user-friendly and consumes transaction-ids.

Rather, it should be possible to open a user session on a banking account with a Web service, like login, which performs the user identification and opens a session. Subsequent Web service invocations for the account would require no caller re-identification, if it is guaranteed that they can be made only by the user that opened the session. At the end, the user has to close the session, or a time-out will close it.

Requirement 1: It should be possible to organize a set of related Web services such that an invocation of a service, like login, establishes a session, and subsequent service invocations are bound to the session.

The invocation of Web services for the collaboration of system components creates a considerable overhead. Measurements we made show the cost for collocated invocations of three different Web services on a Pentium III 800 MHz with Windows 2000: service 1 calculates a square with an integer parameter and result, service 2 concatenates two strings with two string parameters and a string result, service 3 has two vector objects from a user-defined vector class as parameters and returns the sum as a vector result.

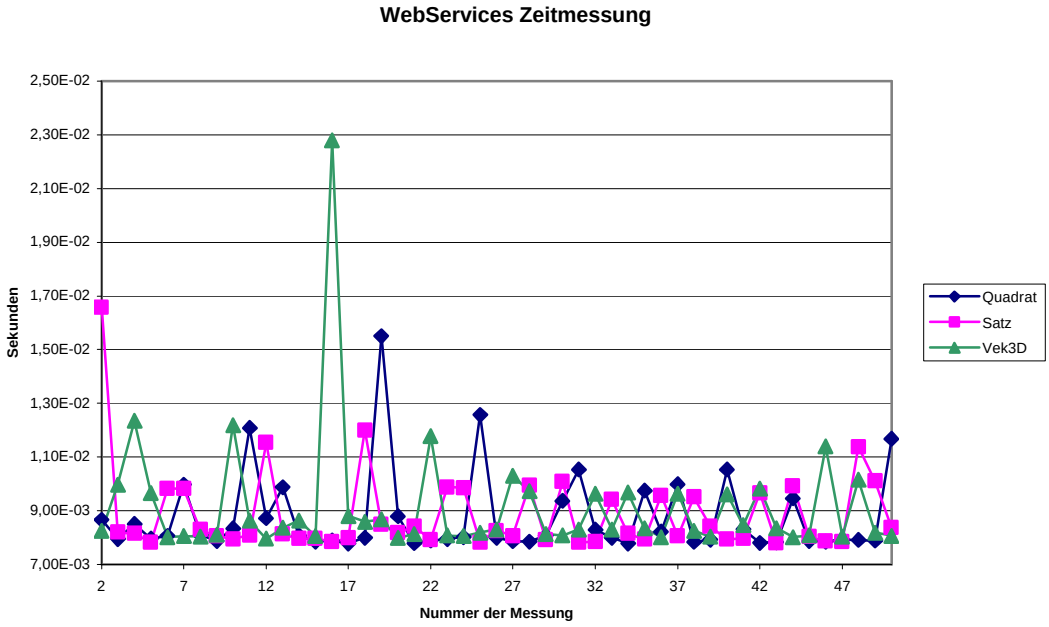


Fig. 2. Cost (in seconds) of local Web service invocation in Windows 2000 IIS C# environment

A collocated Web service invocation costs under Windows 2000 with IIS and C# (see Figure 2) between 8 and 13 milliseconds (disregarding a few exceptions), on an average for service 1, 2 and 3: 8.8, 8.9 and 9.2 milliseconds; under Java 2 JDK 1.3 with Tomcat/Axis (see Figure 3)), on an average for service 1, 2 and 3: 22, 19, and 22 milliseconds, and under Java with Sun WSDP on an average for service 1, 2 and 3: 13, 10, and 11 milliseconds. The numbers for remote invocation over a fast Intranet are roughly of the same magnitude.

A local call of these methods costs, however, between 1.5 and 3.5 microseconds, a CORBA [CACM98] [Se98] [Si98] remote invocation among collocated objects about

0.5 milliseconds and among remote objects about 0.9 milliseconds, as our measurements show (compare measurements of [OH99]).

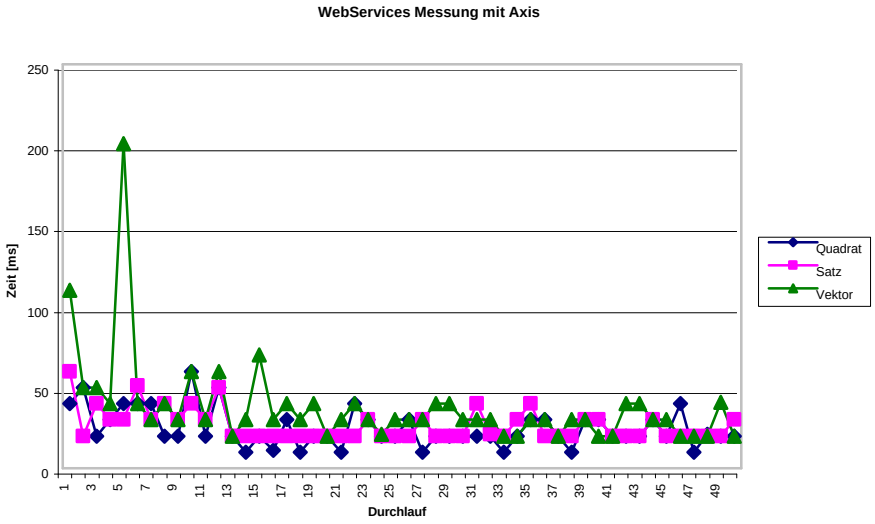


Fig. 3. Cost (in milliseconds) of local Web service invocation in Java Tomcat/Axis environment

Thus, the Web service invocation has an overhead of roughly 9 milliseconds in comparison to local invocations and of 8 milliseconds in comparison to CORBA invocations on our Windows 2000 IIS platform. This overhead is the cost to be paid for the platform independence and simplicity of remote invocations with Web services. When a system is constructed so that the number of remote invocations is kept small [BR 00], the high overhead can be tolerated.

But for collocated invocations among system components, the overhead does not seem to be acceptable in many cases. Performance problems have been experienced even with the smaller CORBA overhead for components like Enterprise JavaBeans [G01]. Suppose that a system invocation from an external user results in 49 internal component invocations like those of Web services 1-3. If the system provides a CORBA interface and uses internally local component interfaces, an external invocation costs roughly 1 millisecond. But if the system provides external and internal Web service boundaries, an external invocation costs 450 milliseconds. Though these assumptions are not necessarily representative (usually, the processing time in a component will take longer), the resulting numbers show clearly that the Web service invocation overhead cannot be neglected and often not tolerated.

Requirement 2: It should be possible to avoid the Web service invocation overhead for calls among local system components without program changes.

3. I3M - an Instant Messaging Based on Web Services

This section presents a WS-based system architecture based on Web services. I3M (International Internet Instant Messaging) provides in chat-rooms instant messaging services, and in conferences instant or delayed messaging services. It is similar to the well-known ICQ, but enhanced in different areas like e.g. automatic message translation. Since it is based on Web services that use HTTP as a transport protocol, the difficulties with firewalls to be encountered with ICQ are avoided. I3M has been designed and built by Eugen Eissler, Andreas Müller, Dirk Plate, Henning Schäfer and Gunther Würz at the Fachbereich Informatik of the Fachhochschule Konstanz starting in 2002.

Though I3M is not really a complex system, we have experienced the problems described in section 2. Let us give a short overview over the main functionality of I3M and the problems encountered.

3.1 Interoperability

I3M consists of a server and of clients. An I3M server installed on an HTTP-application server stores centrally all user data and provides the I3M services in the form of Web services for the clients. The current server version written in C# runs on Windows 2000 with dotnet and IIS. The server provides Web services that are invoked by I3M clients. The current client version written in Java with SWT from IBM as a GUI may be run on any Java 2 JDK 1.4.1 platform.

Though two moderately complex Web service interfaces connect a client and the server, we experienced no difficulties in mixing C# and Java in a system. Without Web services, the connection would have to be made either very low level with TCP sockets, or from RMI/CORBA over IIOP to DCOM using adapters. Both solutions would have been much more difficult.

3.2 Client Sessions

A user must first register itself from a client with an I3M server, supplying the user name, password, language and similar information.

After registration, a client user may login with the I3M server and establish a session. A buddy list is presented showing which buddies are online or offline. A user may open a conference with selected buddies, also if some of them are offline, and exchange messages. The messages are presented instantly to the online buddies; an offline buddy receives the conference messages with delay when he logs on. Alternatively, he may receive them by fax or be email according to his choice. A user may also join a chat-room. This is similar to a conference except for all users being online. Figure 4 gives an overview about an I3M server (bottom) and its clients (top), which have, after starting a session (see tunnel from server to client), opened

conferences and joined chat-rooms. Conferencing and chat-room services are provided as core services by the server core component.

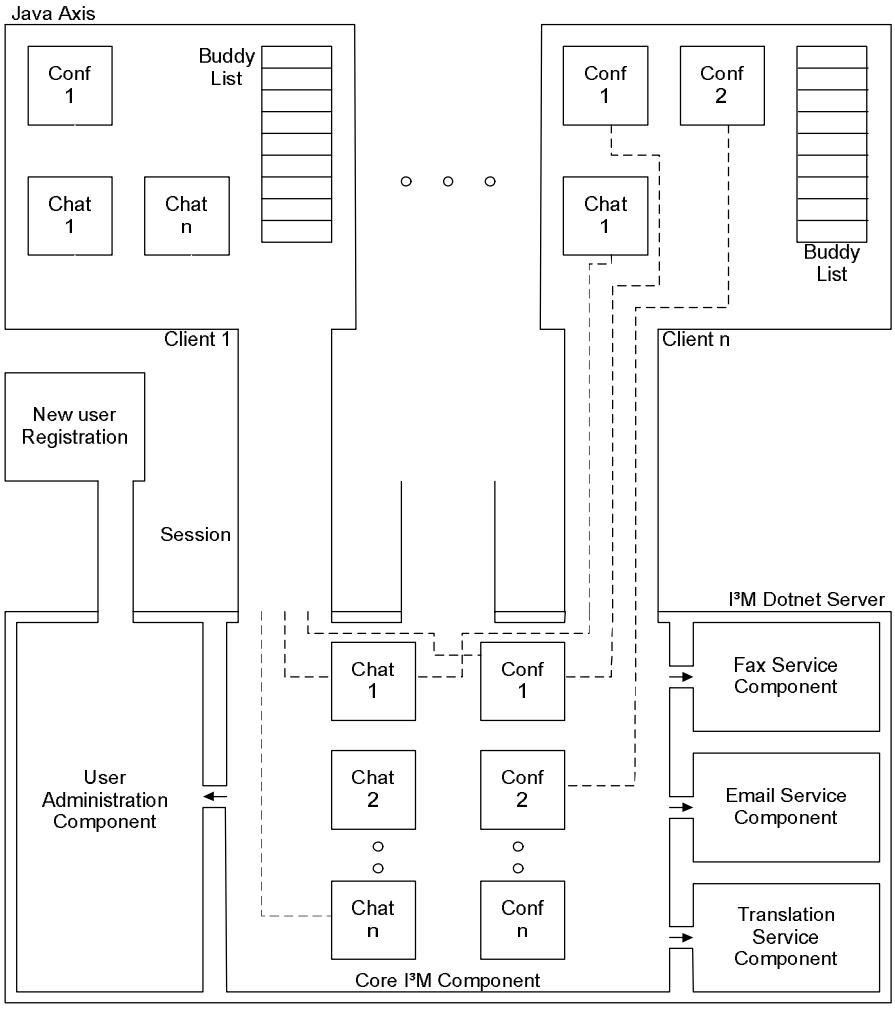


Fig. 4. I3M server (bottom) and clients (top) that have started a session (tunnel from server to client) and opened within it several conferences and chat-rooms

When a client user enters a message, the clients invokes a Web service to pass it to the server that keeps all state information centralized. Consequently, a client should change its state, i.e. the messages displayed, when it or other users participating in a chat or conference have entered a message. The standard solution [GHJV95] would be to make a client an observer of the server: upon a state change, the server would notify all concerned clients via invoking an ”update”-Web service

of the clients. We did not follow that approach since each client would have to embody and install an application server. This may cause, in addition to logistic and organizational problems, firewall problems for all clients working behind a firewall. Therefore, the selected approach is that a client polls the server for state changes in regular intervals.

The I3M system must make sure that only a user client who has gone through user identification obtains the state information about the chats and conferences which he has joined. It must be excluded that other Web users not registered and not successfully logged-in with a server, may invoke the Web services of the I3M server to get information about the users and the messages exchanged in chats and conferences, or that a logged-in user obtains information about chats and conferences that he has not joined.

3.3 I3M Components for Other Services

Other components of I3M provide their services to the core component via a Web service interface. There are different reasons why the collaboration is not done via local interfaces.

The translation service (see Figure 4 bottom right) is invoked by the core component in real-time if the users participating in a chat-room or conference speak different languages. The translation service may either be integrated in an I3M server or it may provided by an independent service provider. As of today, there are different Web sites on the Internet offering translation services, like Babelfish (on: www.xmethods.net/sd/2001/BabelFishService.wsdl). The core service component invokes the translation service via a Web service in order to be independent from the actual service provider and its location. The invocation of Babelfish with a few words to be translated costs about 0.5 seconds response time from Konstanz. Ideally, each I3M server installation should be able to make its own choice which translation service it uses, based on price and translation quality criteria.

Other services like the fax services or email services are also provided by components with a Web service interface. Today, there exist a few fax servers with Web service interface on the Internet so that I3M invokes the services of one of them. But any I3M server installation should be able to provide its own local fax server component without requiring I3M code changes, if the price or quality of the Internet fax service providers would not be appropriate.

The user administration is also provided as a separate component with a Web service interface (see Figure 4 bottom left), though it is an I3M component that plays a central role together with the core component. The core component will invoke the user administration for the identification of a user logging in, and also when a user does searches among the other users. One reason for the separation is that the core component accepts only login requests of registered users, so that the user administration is invoked directly for user registration. Another reason is the enhanced isolation of the sensible user administration.

A consequence of the separation is that it must be guaranteed no other program is calling the user administration component pretending to be the I3M core services. To this purpose, the core component uses a password or private key for identification when calling the administration component.

This short overview about the I3M components with a Web service interface has shown that some components play a central role, though they are not invoked very frequently, like the user administration. Other components give rather an add-on value, like translation or the fax services, but may be invoked very frequently. If the participants in a chat or conference speak different languages, the translation service may have to be invoked several times for each incoming message.

3.4 Flexibility versus Efficiency of Component Invocation

The flexibility gained is the primary reason for using components with a Web service interface to provide the added-value services. However, the flexibility required for I3M and business applications is not the kind of flexibility often envisaged from Web service proponents, namely a completely dynamical binding: an application invokes first a broker with UDDI as a directory, to search for a provider of the required service, and then the found service. I3M does not require dynamical binding of the service provider during program execution. Rather, an I3M server location should have, from time to time, the possibility to rebind the service provider without program changes or recompilation. That means a server should be able to re-select an add-on service provider, or even to integrate a component providing the service in its own installation. But then, the binding is fixed.

With the current tools available both on the Java Axis and Windows 2000 dotnet platform, a rebinding is not possible without recompilation (to our knowledge) since the Web service location is fixedly encoded into the proxy when generating it, and cannot be parameterized. Thus, the proxies have to be regenerated with the tool and recompiled!

The drawback of the flexibility gained is the efficiency lost. External invocations of a Web service will increase the system load by about the half of the local invocation cost, and the response time over the Internet is in the order of magnitude of 100 milliseconds or seconds. However, when an external provider is used, an external service invocation cannot be avoided.

But the situation is different if e.g. an internal translation service component is invoked via a Web service interface.

The invocation of an I3M Web service like addMessageToChat causes a system load of 4.5 milliseconds on the server (roughly the half of a local service invocation). This is practically only the cost of the Web service invocation; the productive work, adding the message to the chat object, costs a few microseconds so that it can be neglected.

If the message is to be translated for a multilingual chat or conference, each translation costs, on our platform, about 1 millisecond. The total server cost with locally invoked translation are 4.5+1 = 5.5 milliseconds. As a result, under the

assumption that the new message is translated one time, an I3M server processes maximally roughly 1000/5.5 = 181 ”addMessageToChat”-Web services per second, if the translation is invoked locally.

If the local translation component is invoked via a Web service, the server cost per invocation are 4.5 +1+8.8 = 14.3 milliseconds. The I3M server may process maximally about 1000/14.3 = 69 ”addMessageToChat”-Web services per second. Load measurements confirm these numbers.

Thus, the price paid for the increased flexibility is very high, requiring more than doubled hardware resources. In many cases, this is not acceptable.

4. Sessions with Web Service-Based Systems

A great advantage of Web services is that they are based on HTTP as transport protocol, which is well-established and predominant for Internet applications. So Web services may profit from the solutions to problems detected in the context of HTTP-based Web applications, which have been established during the last years.

In a modern browser-driven Web application like e.g. a Web shop, a user opens a shopping session. The shopping session must not be accessed by other users, and the session state (which may contain e.g. a shopping cart) must be preserved. Very often, even the state of a business process that is executed by a modern Web application, must be preserved [SR02].

Application servers with servlets (in the Java world) have solved the problem how to provide a user session with the stateless HTTP request-response protocol. Though a servlet instance (running in parallel with other instances in different threads) is shared among many users (one after the other) sequentially over the time, like a transaction instance, the application server keeps transparently track of the user sessions and their state, by mean of the IP addresses of user nodes or of cookies placed there. It establishes a user session context when a new user requests an HTTP service, and it preserves the session context as long as the user is in a dialog with the server, until he logs off or a time-out causes the session to be terminated.

Technically, an application server provides, in addition to an application context and a request context, a session context in which a Java servlet, JSP (Java Server Page) or ASP (Advanced Server Page) may store and access session-specific information. The contexts contain named variables, which may refer to primitive data types, strings and objects. When a new user makes an HTTP request, a new session context is created automatically by the application server.

Web services generated e.g. with Axis and Sun Java WSDP are executed as servlets in the Java Tomcat environment, those generated with dotnet as ASPs in the Windows 2000 IIS environment. So Web services can use the session technology established for Web applications. Each class that provides methods implementing Web services is placed in a servlet or executed as a JSP or ASP. A Web service class has access to the session context and can store and access variables in it.

We have developed and use the session memento pattern, which is a variant of the memento pattern combined with the template method pattern [GHJV95], to

implement Web service classes. All state to be preserved between Web service invocations during a session is stored in a session memento class, to which the Web service class has a reference. The state required during a Web service invocation is stored as local state in the method implementing the service. The template method serves to invoke automatically, at the beginning of the execution of a Web service, a method that gets the reference of the session memento from the session context and stores it in the Web service class.

The I3M core component uses the session context to make sure that user clients have only after a successful log-in access to the chats, conferences etc. To this purpose, the core component keeps a session memento with a boolean member variable loggedIn, and with member variables referencing the chat and conference objects which a user has joined. The core component sets the loggedIn variable to true after a successful log-in, and checks with every subsequent Web service request in the session memento if the client is logged-in, and if it has access to the respective chats and conferences.

In general, it is a space-time performance optimization question in which form the session state is stored. Consider e.g. the account session example from section 2. The account of the user has to be preserved during the session. This may be done either by storing the account number, or a reference to the account object. In the first case, the account object has to be recreated from the database with each Web service invocation, which costs time, but the memory space required is small. In the second case, there is no time overhead but a larger memory space is required.

5. Collaboration of Components with Web Service Interfaces

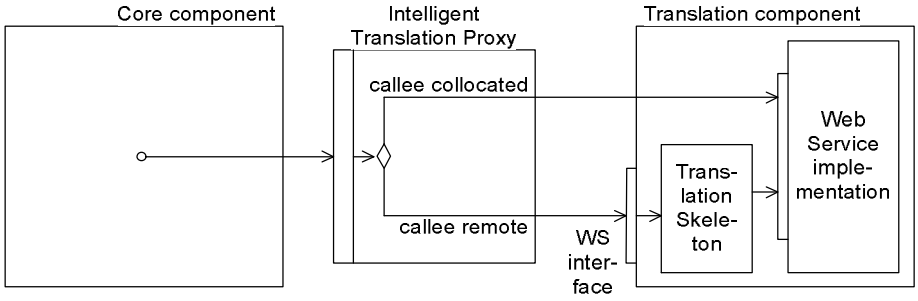
When the components of a system collaborate via Web service interfaces, there are considerable advantages, as discussed in section 1. On the other hand, as the example of the I3M server shows, the overhead for the invocation of collocated system components may be prohibitively great: it makes no sense to have to buy about the double hardware power in order to facilitate an easy exchange of components and to get a uniform description of their interfaces.

Our objective is to get the flexibility of component selection and allocation without having to pay the Web service invocation overhead for invocations of collocated components. More specifically: how can a calling component invoke a service via the local interface if the invoked component is collocated, and via the Web service interface if the invoked component is remote, without having to modify the code of the caller component and of the invoked component?

The solution is to provide intelligent invocation proxies. A non-intelligent invocation proxy as it is generated from the Web service tool-set on each platform, transforms always a method call into the invocation of the matching Web service. An intelligent invocation proxy checks first if the component to be invoked is collocated or remote. If it is local, it obtains the local component reference and forwards the call to the local interface of the called component. Only if the callee is remote, it transforms the method call into the invocation of the matching Web service. In both

cases, an intelligent invocation proxy waits for the result (with a synchronous invocation), and returns it directly or after a retransformation from the SOAP format.

Figure 5 gives an example from I3M, showing an invocation of the translation component by the core component. The core component invokes a Web service by calling a method of the intelligent translation proxy. This makes the decision if the callee is collocated or remote. In the first case, it forwards the method call to the callee's local interface; in the second case, it transforms the call in the invocation of a Web service which it sends off.



The difficulty, how to provide a user session with a stateless request-response protocol is solved by using the session context, as a heritage of Web application servers, together with the session-memento pattern that we propose.

The overhead of Web service invocations for collocated components is not negligible. For a solution of the performance problem, we propose the use of intelligent proxies. They allow to combine the flexibility of Web service invocation with the performance of ”conventional” invocations for WS-based system architectures. This allows to partition also systems, where the performance aspects play a important role, in collaborating components with Web service interfaces.

Note: Intelligent proxies are an intellectual property of the author and protected by patent.

Acknowledgements

My thanks go to Eugen Eissler, Andreas Müller, Dirk Plate, Henning Schäfer and Gunther Würz, who have done a great job in the detailed design, implementation and test of I^{3}M.

7. References

[ACM2001] D.Alur, J.Crupi, D.Malks: Core J2EE Patterns; Prentice Hall, Upper Saddle River, 2001

[BR00] K.Beschorner, W.Rosenstiel: Effiziente Datenuebertragung in EJB-Systemen (German); Proc. Net.ObjectDays2000, Net.ObjectDaysForum, Illmenau, 2000

[CACM98] Special section on CORBA; Communications of the ACM, Vol.41, No.10, October 1998

[G01] S.Grosse: EJB und Geschäftsanwendungen - Mythos und Realität; Java Spektrum, March 2001

[GHJV95] E.Gamma, R.Helm, R.Johnson, J.Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley, 1995

[LC02] J.Lu, Lihui Chen: An Architecture for Building User-Driven Web Tasks via Web Services. In E-Commerce and Web Technologies, Springer LNCS 2455, 2002

[OH99] R.Orfali, D.Harkey: Client/Server Programming with Java and CORBA; John Wiley & Sons, Chicester, UK, 1999

[Se98] K.Seetharaman: The CORBA Connection; in: [CACM98]

[Si98] J.Siegel: CORBA and the OMA in Enterprise Compurting; in: [CACM98]

[SR02] H.A. Schmid, G. Rossi: Designing Business Processes in E-Commerce Applications. In E-Commerce and Web Technologies, Springer LNCS 2455, 2002

[Sz97] C.Szyperski: Component Software, Beyond Object-Oriented Programming; Addison-Wesley, 1997