Loosely Coupled Web Services in Remote Object Federations

Uwe Zdun

Department of Information Systems, Vienna University of Economics, Austria
zdun@acm.org

Abstract Loosely coupled services are gaining importance in many business domains. However, compared to OO-RPC middleware approaches, emerging technologies proposed to implement loosely coupled services, such as Web services or P2P frameworks, still have some practical problems. These arise in many typical business domains, for instance, because of missing central control, high network traffics, scalability problems, performance overheads, or security issues. We propose to use ideas from these emerging technologies in a controlled environment, called a federation. Each remote object (a peer) is controlled in one or more federations, but within this environment peers can collaborate in a simple-to-use, loosely coupled, and ad hoc style of communication. Our design and implementation relies on popular remoting patterns. We present a generic framework architecture based on these patterns together with a prototype implementation.

1 Introduction

Loosely coupled (business) services are nowadays propagated and/or enabled by many different technologies, including Web services, P2P systems, coordination and cooperation technologies, and spontaneous networking. Compared to OO-RPC middleware approaches, such as CORBA, RMI, or DCOM, these approaches promise loose coupling, a service-based architecture, ease of use, and ease of deployment. However, as we will point out in Section 2, today all these technologies have their limitations in the context of business-critical systems, for instance, regarding central control tasks, network traffics, scalability, performance, or security.

Typical applications of loosely coupled (Web) services in different business domains are workflows, groupware, legacy integration, or coordination of business components. When we offer loosely coupled (Web) services in these business domains, there are some specific, recurring requirements. For instance, if spontaneous connections are allowed, we require some level of control to ensure that a business service cannot be misused. Consider an e-commerce service that should be provided only to service users who have paid for the service. One business peer can play more than one role in different contexts. Consider a peer that represents the delivery service of a content provider: it also has to provide a contract engine and handle rights enforcement. To model such situations, we cannot use the "all peers are equals" model of current P2P environments in the whole system. However, it would be useful, if peers that are actually equals in a certain situation can be handled as such. As we could use a very simple remoting model in such cases, this would ease the development of distributed programs significantly.

Moreover, service-based architectures and ad hoc connectivity may ease deployment in a controlled environment, say, within a company.

We propose a federated model of remote objects as a solution. Within a federation, each peer offers Web services (and possibly other kinds of services) to its peers, can connect spontaneously to other peers (and to the federation), and is equal to its peers. Each remote object can potentially be part of more than one federation as a peer, and each peer decides which services it provides to which federation. Certain peers in a federation can be able to access extra services that are not offered to other peers in this federation via its other federations. A semantic lookup service allows for finding peers using metadata, exposed by the peers according to some ontology. Thus it enables loosely coupled services and simple self-adaptations for interface or version changes.

We present a framework for loosely coupled Web services built internally with well known (OO-)RPC remoting patterns (from [21,22,23]). We will discuss a reference implementation written in the object-oriented Tcl variant XOTcl [18] using SOAP-based communication. The pattern-based design has the aim that a similar framework can be implemented in any language with any Web service framework. The framework is designed to be extensible and implementation decisions, such as using a particular SOAP implementation as the communication protocol, can be exchanged, if required.

In this paper, we first discuss prior work in the areas of Web services, P2P systems, coordination technologies, and spontaneous networking. Then we discuss open issues in these approaches regarding loosely coupled services. Next, we discuss a generic framework design and a prototype implementation in XOTcl on top of SOAP, called Leela. Finally, we discuss in how far the open issues are resolved in our concepts and conclude.

2 Related Work

In this section, we discuss related work in the areas of Web services, P2P systems, coordination technologies, and spontaneous networking. We will see that all of these concepts implement some of the desired functionalities, but leave a few issues open.

2.1 Web Services

Web service architectures center around the service concept, meaning that a service is seen as a (set of) component(s) together with a providing organization. Thus Web services are a technology offering both, concepts for deployment and providing access to business functions over the Web. Technically, Web services build on different Web service stacks, such as IBM's WSCA [15] or Microsoft's .NET [17]. These have a few standard protocols in common, but the Web service stack architectures are currently still diverse. At least, HTTP [11] is usually supported for remote communication. Asynchronous messaging protocols are also supported. SOAP [4] is used as a message exchange protocol on top of the communication protocol. Remote services can be specified with the Web Service Description Language (WSDL) [7]. WSDL is an XML format for describing Web services as a set of endpoints. Operations and messages are described abstractly, and then bound to a concrete communication protocol and message format to define an endpoint. Naming and lookup is supported by UDDI [19].

Each Web service can be accessed ad hoc, and services are located and bound at runtime. Additional composition of services is supported by business process execution languages, such as BPEL4WS [1]. Such languages provide high-level standards for (hierarchical) flows of Web services.

Web services are providing a loosely coupled service architecture and a service deployment model. However, today's Web service stack architectures are already relatively complex and have a considerable overhead, especially for XML processing. Federated or grouped composition is not yet in focus, even though technically possible.

2.2 Peer-To-Peer Systems

Peer-to-Peer (P2P) computing refers to the concept of networks of equal peers collaborating for specific tasks. P2P environments allow for some kind of spontaneous or ad hoc networking abilities. Typical applications of P2P are file sharing, grid computing, and groupware. P2P computing is a special form of distributed computing that has gained much attention in recent times, especially P2P systems for personal use, like Gnutella, Napster, and others.

Technically there are still quite diverse views on P2P. For instance, P2P can be interpreted as a variant of the client/server paradigm in which clients are also servers; it can also be interpreted as a network without servers. Often P2P is referred to as a type of network in which each peer has equivalent capabilities and responsibilities. This differs from client/server architectures, in which some computers are dedicated to serving the others. Note that this is only a distinction at the application level. At the technology level both architectures can be implemented by the same means.

Basic functionalities shared by most current P2P systems are that they connect peers for a common purpose, permit users to lookup peer services, and provide a way to exchange data or invoke remote services. These basic properties are still quite vague – and do also apply for many client/server architectures. There are many optional properties, one can expect from a P2P system, but none is a single identifying property – that is, all can also be missing [8]:

- usually there is some kind of sharing of resources or services,
- there is an ease-of-use for users or developers,
- there is a direct exchange between peer systems,
- usually clients are also servers,
- load distribution may be supported in some way, and
- there is a notion of location unawareness regarding a used service – provided mainly by the lookup service used to locate a desired service.

Regarding remote business services, P2P offers a set of potential benefits: it can be used to provide a very simple remoting infrastructure and loose coupling is inherently modeled. However, missing central coordination may cause problems regarding security, performance, scalability, and network traffic.

2.3 Coordination Models

Coordination models are foundations for a coordination language and a coordination system, as an implementation of the model. A coordination model can be seen as a formal framework for expressing the interaction among components in a multi-component system [9]. As related work for our work, especially coordination of distributed and concurrent systems is of interest. The coordination language Linda [13] introduced the view that coordination of concurrent systems is orthogonal to the execution of operations (i.e. calculations) in this system. Linda can be used to model most prevalent coordination languages. It consists of a small number of coordination primitives and a shared dataspace containing tuples (the tuplespace).

The original Linda has no notions of multiple federations; a single tuplespace is used for all processes. However, this has its practical limitation regarding distributed systems, as the single tuplespace is a performance bottleneck. Moreover, there is no structuring of sub-spaces and scalability is limited. Bauhaus [6] introduces the idea of bags that nest processes in tuplespaces. A process can only be coordinated with other processes in the bags, or it has to move into a common bag to coordinate with other processes. PageSpace [10] structures Linda spaces by controlled access using different agents for user representation, interfaces to other agents, administrative functionality, and gateways to other PageSpaces.

2.4 Spontaneous Networking

Spontaneous networking refers to the automatic or self-adaptive integration of services and devices into distributed environments. New services and devices are made available without intervention by users. Services can be provided and located in the network. Providing means that they can be dynamically added to or removed from the network group without interfering with the global functionality. Failure of any attached service does not further affect the functionality of the network group. Failing services are automatically removed and the respective services are de-registered.

Jini [2] is a distributed computing model built for spontaneous networking. Service providers as well as clients firstly have to locate a lookup service. A reference to the lookup service can be received via a multicast. Service providers register with the lookup service by providing a proxy for their services as well as a set of service attributes. Each service receives a lease that has to be renewed from time to time. If a lease expires, the service is automatically removed from the network. Clients looking for a service with particular attributes send a request to a lookup service for such a service. In response the client receives all those proxies of services matching the requested service attributes.

The Home Audio Video interoperability (HAVi) standard [14] is designed for networking consumer electronics (CEs). Especially, self-management and plug & play functionalities are provided for spontaneous networking. Remote services are registered using a unique Software Element Identifier (SEID) with a system-wide registry for service lookup. HAVi specifies the communication protocols and access methods for software elements in a platform-independent way.

2.5 Open Issues for Loosely Coupled Service Architectures

The concepts, described in the previous sections, can be used to implement loosely coupled service architectures. However, all have different benefits and liabilities in the context of business systems, such as information systems within an organization (for workflows, groupware, etc.) or information systems offering services to the outside (such as e-commerce environments). In this paper, we propose to combine some ideas of these approaches to resolve the following open issues:

- *Control of Peers and Access to the Network:* If a peer offers a vital service that should not be visible to everyone (for instance, only to those who have paid), we have to control access in business environments. The idea is to combine the grouping concepts of coordination models, such as tuplespaces, with basic networking properties of the P2P model.
- *Dynamic Invocation:* If static interface descriptions are mandatory for remote invocations, ad hoc connectivity is hard to model. In the context of loosely coupled Web services we propose the use of dynamic invocation mechanisms (on top of SOAP).
- *Simplicity:* For the application developer, remoting technologies should be in first place simple. In a coordinated group, where we can be sure that access can be granted, developers of remote objects accessing the service should be able to use a very simple remoting model with direct interactions.
- *Security:* Access to coordinated groups and the permissions what a peer can do within a group have to be secured.
- *Performance and Scalability:* The internal protocols used should be exchangeable to deal with performance and scalability issues. It should be possible to replace performance-intensive (or memory-intensive) framework parts transparently and provide means for QoS control.
- *Deployment:* Each accessible remote object should provide services that expose the ease of deployment and access known from Web services.

3 Peer Federations

In this section, we will step-by-step discuss our concepts for peer federations. These are combining concepts from the different approaches, discussed in the previous section, to resolve (some of) the named open issues. We illustrate our concepts with examples from our prototype implementation Leela. Leela is implemented in XOTcl [18], an object-oriented scripting language based on Tcl. Our framework is designed with the remoting patterns[1] from [21,22,23]. We illustrate our designs with UML diagrams.

Before we describe the peer and federation concepts, we describe the basic concepts of the communication framework of Leela. The communication framework's model is tightly integrated with the higher-level peer and federation concepts. Therefore, it is important to understand its design before we go into details of the peer and federation concepts.

3.1 Basic Communication Framework

As its basic communication resource, each Leela application uses two classes implementing a CLIENT REQUEST HANDLER [23] and a SERVER REQUEST HANDLER [23] (see Figure

[1] We highlight pattern names in SMALLCAPS font.

1). The CLIENT REQUEST HANDLER pattern describes how to send requests across the network and receive responses in an efficient way on client side. On the server side, the requests are received by a SERVER REQUEST HANDLER. This pattern describes how to efficiently receive request from the network, dispatch the requests into the server application, and send the response back to the client side. Each Leela application instance acts as a client and server at the same time. The Leela application instance and its request handlers can be accessed by each peer.

The request handlers contain PROTOCOL PLUG-INS for different protocols that actually transport the message across the network. Currently, we support a SOAP [4] protocol plug-in. However, any other communication protocol can be used as well. As described below, Leela supports different invocation and activation styles (see Sections 3.4). Thus the specialties of most protocols supporting mainstream communication models, such as remote procedure calls (RPC) or messaging, can be supported. It is expected from the protocol that it can – at least – transport any kind of strings as message payload, and that one of the invocation and activation styles, supported by Leela peers, can be mapped to the protocol. For most protocols, it should be possible to map all invocation and activation styles of Leela to the protocol – of course, with different trade-offs.

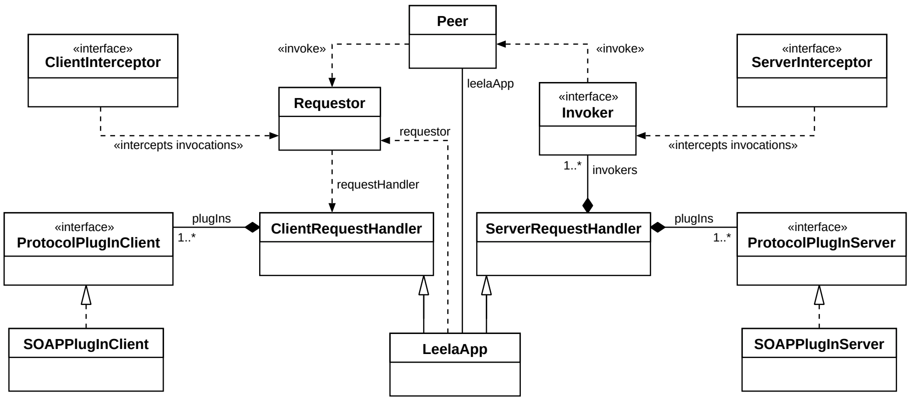

Figure 1. Structure of the Leela Communication Framework

Remote invocations are abstracted by the patterns REQUESTOR [23] and INVOKER [23]. A REQUESTOR is responsible for building up remote invocations at runtime and for handing the invocation over to the CLIENT REQUEST HANDLER, which sends it across the network. The REQUESTOR offers a dynamic invocation interface, similar to those offered by OO-RPC middleware such as CORBA or RMI. Leela also supports peer and federation proxies that can act like a CLIENT PROXY, offering the interfaces of a remote peer or federation.

The INVOKER gets the invocation from the SERVER REQUEST HANDLER and performs the invocation of the peer. In Leela, there are different INVOKERS for different activation strategies (see Section 3.4). The SERVER REQUEST HANDLER is responsible for selecting the correct INVOKER. The INVOKER checks whether it is possible to dispatch the invocation; in

Leela only exported objects and methods can be dispatched. This way, developers can ensure that no malicious invocations can be invoked remotely.

The Leela invocation chain on client side and server side is based on INVOCATION INTERCEPTORS [23]. That is, the invocation on both sides can be extended transparently with new behavior. Interceptors are used in Leela to add information about the Leela federation to the invocation (see below). Also, a client-side INVOCATION INTERCEPTOR can add security attributes and similar information to the invocation. A server-side interceptor can read and handle the information provided by the client.

The REQUESTOR, INVOKER, and request handlers handle synchronization issues on client and server side. The request handlers handle the invocations according to the invocation and activation styles used. On server side, the SERVER REQUEST HANDLER receives network events asynchronously from a REACTOR [20]. The SERVER REQUEST HANDLER can have multiple different PROTOCOL PLUG-INS at the same time. That is, network events can come in from different channels concurrently. The SERVER REQUEST HANDLER queues the network events in an event loop.

The actual invocations of peers are executed in a separate thread of control. The access of a particular peer can either be queued (synchronized) or handled by a multi-threaded OBJECT POOL [23]. The results are queued again, and handed back to the receiving thread.

On client side, different styles of asynchronous invocation and result handling are supported. Because in Leela each client is also a server, synchronous invocations – that let the client process block for the result – are not an option: if the Leela application blocks, it cannot service incoming requests anymore. Instead, Leela implements a variety of asynchronous invocation styles with a common callback model. The request handlers work using an event loop that queues up incoming and outgoing requests in a MESSAGE QUEUE. Client-side invocations run in a separate thread of control.

The result arrives asynchronously and has to be obtained from the receiving thread. This is done by raising an event in the CLIENT REQUEST HANDLER's event loop. This event executes a callback specified during the invocation. An ASYNCHRONOUS COMPLETION TOKEN (ACT) [20] is used to map the result to its invocation. Using this callback model we can implement different asynchronous invocation styles, described in [22,23]. We can send the invocation and forget about the result as described by the pattern FIRE AND FORGET. SYNC WITH SERVER is used, when a result is not needed, but we want an acknowledgment from the server. Finally, the patterns POLL OBJECT and RESULT CALLBACK allow us to receive the result asynchronously. POLL OBJECT lets the callback write the result to an object that is subsequently polled by the client for the result. RESULT CALLBACK propagates the callback to the client – that is, it informs the client actively of the result.

3.2 Invocation Types

A remote invocation consists of a number of elements. Firstly, the actual invocation data consists of method name and parameters. Secondly, a service name is required – it is an unique OBJECT ID that enables the INVOKER to select the peer object. Thirdly, protocol-specific location information is required – in the case of SOAP over HTTP this is the host and the port of the SERVER REQUEST HANDLER. The OBJECT ID plus location information implement the pattern ABSOLUTE OBJECT REFERENCE – an unique reference for the particular service in the network. Finally, the invocation might contain INVOCATION CONTEXT data. The INVOCATION CONTEXT [23] contains additional parameters of the invocation, such as information about the federation or security attributes. In Leela, the INVOCATION CONTEXT is extensible by peers and INVOCATION INTERCEPTORS.

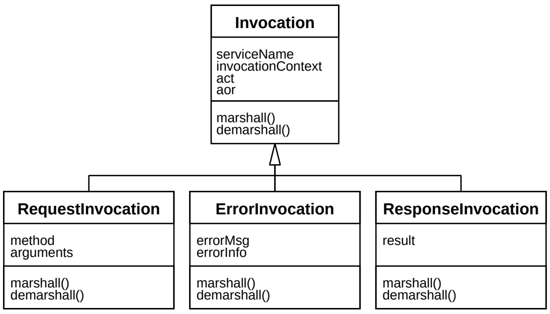

Figure 2. Invocation Types and Marshallers

Leela sends the message payload as a structured string (we use Tcl lists). These strings are different for different invocation types. Currently, we support request, response, and error invocation types. The scheme is extensible with any other invocation type. The error message type is used to implement the pattern REMOTING ERROR [23] – we use it to signal remoting-specific error conditions in the Leela framework.

As shown in Figure 2 the different invocation types contain different information. Converting invocations to and from byte streams that can be transported across the network is the task of the pattern MARSHALLER [23]. The invocation classes shown in Figure 2 are able to marshal and demarshal the information stored in them; thus they implement the main part of the MARSHALLER pattern for the Leela framework.

3.3 Federations and Peers

A federation is a concept to manage remote objects in a remote object group (here federation members are called peers). Each federation has one central federation object that manages the federation data consistently. To allow peers to connect to a federation, the federation itself must be accessible remotely. Thus the federation itself is a special peer. Peers can be added and removed to a federation.

A federation can be accessed remotely by a federation proxy. This is a special CLIENT PROXY [23] that enables peers to access their federation, if it is not located on the same machine. The federation proxy is a local object that implements the federation interface. In principle, it sends each invocation across the network to the connected federation.

Similar to federations, there is also a CLIENT PROXY for peers, the peer proxy. The peer proxy basically implements the peer interface and sends all invocations to the connected peer of which it holds the ABSOLUTE OBJECT REFERENCE. Thus, using the peer proxy, a local peer can interact with any remote peer that is part of its federation as if it is another local

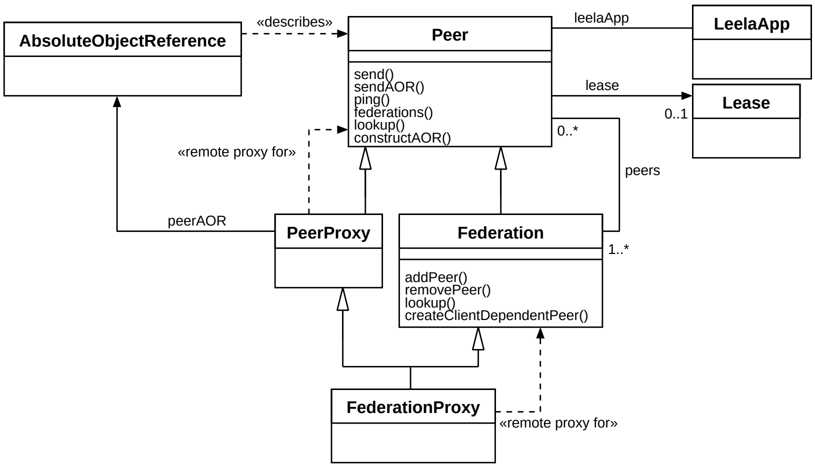

Figure 3. Peer Federations Structure

peer. The peer can invoke other local and remote peers using location information (with the method `send`) or using an absolute object reference (with the method `sendAOR`). The federation and peer structures are shown in Figure 3.

Federations can be introspected for their peers and properties using a semantic lookup service (see Section 3.6). Peers can also perform a lookup: here all federations of the peer are queried.

3.4 Peer Activation

In a loosely coupled remoting environment, activation of remote objects is a critical issue. Activation means creation and initialization of a remote object so that it can serve requests. Some peers are long living and/or persistent entities. Others are perhaps client-dependent such as peers that represent some session data. A client-dependent peer should be removed from the federation, at least, when the last peer that uses it, is destroyed or leaves the federation. In such cases, a lifecycle manager [23] has to ensure that these peers are removed from the federation, if they are not required anymore. We support the following activation Strategies [12] (see Figure 4):

- static instance: The peer is already activated before it is exported and survives until it is explicitly destroyed or the Leela process stops.
- per-call instance: The class of the peer is exported and the peer is activated when the request arrives. Then this peer serves the request and is de-activated again. Per-call activated peers apply the pattern object pooling [23] – that is, they are pre-initialized in a pool to reduce the activation overhead for instantiation.
- client-dependent instance: A factory operation is provided by the federation to create client-dependent peers, e.g. to store session data. In a remote environment, however, it is unclear, when a client-dependent object is not needed anymore, except the client explicitly destroys it. If a given object is not accessed for a while this

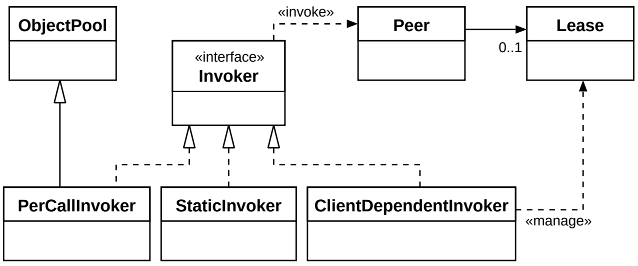

Figure 4. Activation Strategies Implemented on Invokers

can mean that the client has forgotten to clean up the instance, that it requires a longer computation time till the next access, that network latency causes the delay, or that the client application has crashed. The pattern LEASES [23] helps the federation to decide whether a certain client-dependent peer is still required. For each client-dependent peer a lease is started when it is created. The activating peer has to renew the lease from time to time. The lease is automatically renewed, when the peer is accessed. The client can also renew the lease explicitly using the operation `ping` of the peer. When the lease expires and it is not renewed, the client-dependent peer is removed from the federation.

In Leela the LIFECYCLE MANAGER pattern – implementing the management of the activation STRATEGIES – is implemented by different elements of the framework. Peers are registered with an activation strategy. There are different INVOKERS for the different activation STRATEGIES. The appropriate INVOKER is chosen by the SERVER REQUEST HANDLER.

3.5 Peer Invocation and Federation Control

A peer may be invoked only by the federation, or by a peer in one of its federations, or by a local object in its own scope (e.g. a helper object the peer has created itself). Peers are executed in their own thread of control, and each of these threads has its own Tcl interpreter as THREAD-SPECIFIC STORAGE [20]. Thus peers have no direct access to the main interpreter. The threaded peer interpreters are synchronized by MESSAGE QUEUES [23] implemented by event loops of the interpreters. That is, the peer threads can only post “send” and “result” events into the main interpreter – and the request handlers decide how to handle these events.

In other words, each federation controls its peers. These cannot be accessed from outside of the federation without a permission of the federation. Of course, some peers in a federation need to be declared to be publicly accessible. For instance, the federation peer is accessible from the outside – otherwise remote peers would not be able to join the federation.

Control of remote federation access is done by INVOCATION INTERCEPTORS [23] (see Figure 5). On client side, an INVOCATION INTERCEPTOR intercepts the construction of the remote invocation and adds all federation information for a peer into the INVOCATION

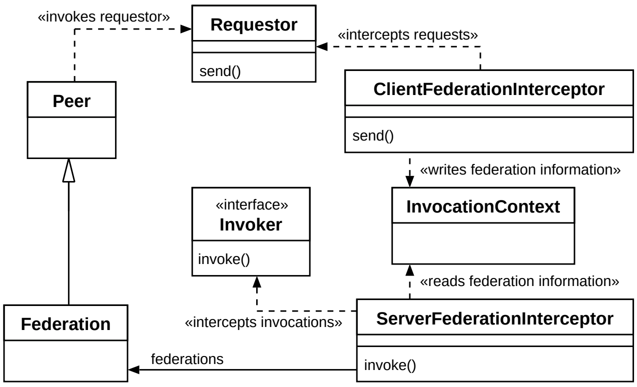

Figure 5. Peer Federation Interceptor Structure

CONTEXT. On server side this information is read out again by another INVOCATION INTERCEPTOR. If the remote peer is not allowed to access the invoked peer, the INVOCATION INTERCEPTOR stops the invocation and sends a REMOTING ERROR to the client. Otherwise access is granted.

Peers within a federation can access their services with equal rights. Per default each peer is allowed to freely send invocations to each other peer in its federation and access exported services. Each service offered in a federation must be explicitly exported by a peer. Only exported services can be accessed by other peers. By introducing INVOCATION INTERCEPTORS for particular peers, peer types, INVOKERS, or REQUESTORS we can fine-tune the control how these elements can be accessed. For instance, we can introduce an interceptor that only grants access if some security credentials, such as user name and password, are sent with the invocation.

Some peers are members of multiple federations. Thus they are able to access services of peers in other federations, something the other peers in the federation cannot do. Optionally, peers can act as a “bridge” to another federation – offering some of that federation’s services in the context of another federation.

Figure 6 shows an example sequence diagram of an invocation sequence with a static INVOKER. Details, such as marshaling and demarshaling, are not shown here. The two other activation strategies just require some additional steps for interacting with the object pool or dealing with the lease.

3.6 Semantic Lookup Service

The idea to provide loosely coupled business services is often not easy to implement because dynamic invocation of these services requires us to know at least the object ID, operation name, and operation signature. To enable ad hoc connectivity this information can potential be unknown until runtime. Therefore, compile time INTERFACE DESCRIPTION [23] approaches (like interface description languages) are not enough. Instead a dynamic INTERFACE DESCRIPTION is required as well.

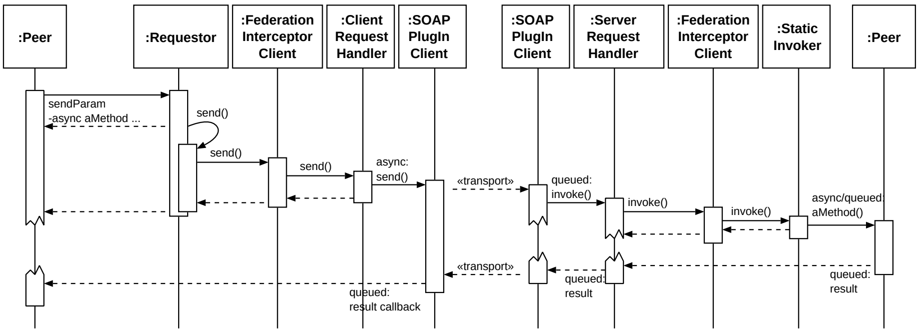

Figure 6. Sequence Diagram for a Simple Invocation with a Static Invoker

The pattern LOOKUP [23] is implemented by many discovery or naming services. These provide the necessary details of any remote object that is matching a query. Here, often another problem is that the designers of the lookup services cannot know in advance how the query might look like and which strategies should be applied to retrieve the results. For instance, always searching for all matching peers can be problematic regarding performance; always (deterministically) returning the first matching peer can cause problems regarding load balancing. Thus we propose a lookup service that is extensible regarding the provided information and the possible queries.

The basic concept of the Leela lookup service is that each peer provides semantic metadata about itself to its federation's lookup service. Peers can perform lookups in all lookup services of their federations. We use RDF [24] to describe the peers. RDF supports semantic metadata about Web resources described in some ontology or schema. For instance RDF-Schema [5] and OWL [16] support general relationships about resources, like “subclass of”. Developers can also use RDF ontologies from other domains; for instance, in an e-learning system probably an ontology for learning materials will be used.

The federation provides metadata about all its peers, such as a list of ABSOLUTE OBJECT REFERENCES and OBJECT IDS (the service names). Each peer adds information for its exported methods, their interfaces, and their activation strategy. This information can be introspected by clients.

Leela currently implements a distributed interface for the Redland RDF library [3] and its interface abstractions. Peers of a federation can read from and write to this metadata repository. As query abstractions, Redland supports the lookup of specific resources and sets of resources, the generation of streams, and iterators. The actual query is thus constructed on client side. In the future we plan to support a more powerful query engine on top of Redland.

4 Discussion and Conclusion

We have presented an approach for service-based remote programming based on remote object groups, called federations. The approach has similarities to Web services, P2P systems, coordination technologies, and spontaneous networking, but can also resolve some apparent open issues of these approaches. The most important design goal is ease-of-use regarding the development, use, and deployment of remote services. In many business scenarios often a certain level of control is required. For this goal, we have provided a simple control model introduced by the concept of peers that can join multiple federations. Only those services that should be accessed by a remote peer are exported. Interceptors can be used to fine-tune the remote access. Services can be introspected for metadata using a semantic lookup service. Thus we can deal with unexpected lookup information and query types; services just have to expose additional metadata and the appropriate queries can be constructed on client side.

Our design and implementation are based on well-known remoting patterns (from [21,22,23]) and follow the pattern language quite closely. Therefore, many underlying parts of our framework can be exchanged with other (OO-)RPC middleware or be implemented in other programming languages – a benefit of the pattern-based design. We are currently working on a Java implementation using the Apache Axis Web service framework, and we are implementing more protocol plug-ins. Thus we believe our results as generally applicable. Moreover, we can potentially deal with scalability and performance problems, as the framework is designed in such a way that the internal protocols and technologies are exchangeable. Our deployment model is similarly simple as Web service and P2P models; however, we require to know the location of at least one "well-known" federation to connect to a business service environment. We consider this not as a drawback, but an incentive in many business scenarios. Note that this "well-known" federation might just provide a lookup service. Thus the activities how objects are initially located are quite similar to lookups in other middleware environments, such as CORBA or Web service frameworks – but they are different to those P2P environments that are exploiting broadcasts and similar means.

The security aspect is handled by controlling which objects can join a federation and that only exported methods can be invoked. Each peer executes in its own interpreter and thread of control – thus peers cannot interfere with each other. All other security issues can be handled by INVOCATION INTERCEPTORS and PROTOCOL PLUG-INS.

We believe our framework design to be usable in many (business) scenarios and plan to apply it for different applications as future work. Especially, we want to use the framework as a very simple remoting infrastructure. The federation model can be used for simple role modeling in a company; on top of such models, workflows and groupware applications can be implemented. As the scripting language XOTcl is primarily designed for component composition, we also want to use the Leela framework for distributed component gluing and coordination, especially in the context of legacy system integration.

Most of the ingredients of our approach are already known from other approaches, but we combine them in an easy-to-use remoting concept. The framework can be extended with add-on functionality. There are also some liabilities of the current prototype implementation: the current prototype does not allow for structured federations (like hierarchical federations), is implemented with SOAP only, and offers no QoS or failover control features (except those of the used Web server). We plan to deal with these issues in future releases of the Leela framework.

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. http://www-106.ibm.com/developerworks/library/ws-bpel/, 2003.
2. K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, , and J. Wald. *The Jini Specification*. Addison-Wesley, 1999.
3. D. Beckett. Redland RDF Application Framework. http://www.redland.opensource.ac.uk/, 2004.
4. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/, 2000.
5. D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/, 2004.
6. N. Carriero, D. Gelernter, and L. Zuck. Bauhaus linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems: Proc. of the ECOOP'94 Workshop on Modles and Languages for Coordination of Parallelism and Distribution*, pages 66–76. Springer, Berlin, Heidelberg, 1995.
7. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.
8. E. Chtcherbina and M. Voelter. P2P Patterns – Results from the EuroPLoP 2002 Focus Group. In *Proceedings of EuroPlop 2002*, Irsee, Germany, July 2002.
9. P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, 1996.
10. P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. Pagespace: an architecture to coordinate distributed applications on the web. *Computer Networks and ISDN Systems*, 28(7-11):941–952, 1996.
11. R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, 1999.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
13. D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
14. HAVI. HAVI specification 1.1. http://www.havi.org, May 2001.
15. H. Kreger. Web service conceptual architecture. IBM Whitepaper, 2001.
16. D. L. McGuinness and F. van Harmelen. Web Ontology Language (OWL). http://www.w3.org/TR/2004/REC-owl-features-20040210/, 2004.
17. Microsoft. .NET framework. http://msdn.microsoft.com//netframework, 2003.
18. G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.
19. OASIS. UDDI. http://www.uddi.org/, 2004.
20. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.
21. M. Voelter, M. Kircher, and U. Zdun. Object-oriented remoting: A pattern language. In *Proceedings of VikingPLoP 2002*, Denmark, Sep 2002.
22. M. Voelter, M. Kircher, and U. Zdun. Patterns for asynchronous invocations in distributed object frameworks. In *Proceedings of EuroPlop 2003*, Irsee, Germany, Jun 2003.
23. M. Voelter, M. Kircher, and U. Zdun. Remoting patterns. To be published by J. Wiley and Sons Ltd. in Wiley's pattern series in 2004, 2004.
24. W3C. Resource Description Framework (RDF). http://www.w3.org/RDF/, 2004.