Pattern-Based Design of a Service-Oriented Middleware for Remote Object Federations

UWE ZDUN
Vienna University of Technology

Service-oriented middleware architectures should enable the rapid realization of loosely coupled services. Unfortunately, existing technologies used for service-oriented middleware architectures, such as Web services, P2P systems, coordination and cooperation technologies, and spontaneous networking, do not fully support all requirements in the realm of loosely coupled business services yet. Typical problems that arise in many business domains are for instance missing central control, complex cooperation models, complex lookup models, or issues regarding dynamic deployment. We used a pattern-based approach to identify the well working solutions in the different technologies for loosely coupled services. Then we reused this design knowledge in our concept for a service-oriented middleware. This concept is centered around a controlled environment, called a federation. Each remote object (a peer service) 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. A semantic lookup service is used to let the peers publish rich metadata about themselves to their fellow peers.

Categories and Subject Descriptors: D.2.7 [**Software Architectures**]: Patterns; D.2.12 [**Interoperability**]: Distributed objects; C.2.4 [**Distributed Systems**]: Distributed applications

General Terms: Design

Additional Key Words and Phrases: Service-oriented Architecture, Middleware, Software Patterns

1. INTRODUCTION

Service-oriented architectures (SOA) aim to enable loosely coupled (business) services. At the middleware level, a number of technologies have been proposed to realize such loosely coupled services, including Web services [Kreger 2001; Apache Software Foundation 2004; Microsoft 2003], P2P systems [Chtcherbina and Voelter 2002; Morris et al. 2001; Rowstron and Druschel 2001], coordination and cooperation technologies and languages [Gelernter et al. 1985; Carriero et al. 1995; Ciancarini et al. 1998], and spontaneous networking [Arnold et al. 1999; HAVI 2001]. Consider the use of services in typical business applications, such as workflows, groupware, legacy integration, or coordination and cooperation of business components. In such applications, a SOA middleware should enable loose coupling, ease of use, and ease of deployment with regard to remote objects or services. In the

Author's address: Uwe Zdun, Information Systems Institute, Vienna University of Technology, Argentinierstrasse 8/184-1, A-1040 Wien, Austria

context of such business applications these goals lead to a set of recurring, technical requirements for the SOA middleware:

—*Dynamic invocations:* In a loosely coupled environment, we cannot always foresee all interfaces of remote objects or services before runtime. Instead new remote objects or services or new versions of them can get deployed at runtime. To cope with such situations, the middleware should support dynamic invocations.

—*Central control:* 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. The middleware technology should offer means to easily define and ensure a suitable control model.

—*Dynamic deployment:* New services or new versions of a service can get deployed and removed at any time. The middleware technology should support the easy deployment and announcement of services at runtime.

—*Simplified cooperation model:* Within a company, it should be very easy to cooperate with other loosely coupled services, for instance, to easily connect services to workflows or rapidly wrap legacy systems. Compared to OO-RPC middleware approaches, such as CORBA, RMI, or DCOM, a simplified cooperation model is needed, but it must not violate the central control requirements.

—*Lookup support:* If interfaces can change and be added at runtime, a flexible lookup service for *interface lookup* is needed, so that business applications can find out about changes in the service landscape. Because any interface detail might be unknown before runtime, a flexible *lookup based on properties* and/or flexible *lookup queries* are useful in more loosely coupled situations.

—*Service separation:* It might optionally be useful to support a clear model for service separation within the local scope of an application. This way the middleware can enforce that the service interaction model is not violated. This eases the implementation of higher-level service interaction models or security features.

—*Quality attributes:* A SOA middleware technology should not have – in comparison to related approaches – a highly negative influence on important quality attributes, such as security, performance, scalability, complexity, or performance.

Each of the existing middleware approaches has its particular strengths in some of these areas, but weaknesses in other ones. For instance, Web service approaches offer dynamic invocation models and central control at the server side. But today's Web services stack architectures are already relatively complex. They follow the classical client/server model, and hence do not offer a simplified cooperation model. The deployment model and the lookup approaches are rather static and inflexible. These problems are to a certain extent resolved by P2P systems, which provide a simpler cooperation model and allow for more spontaneous connections. Current P2P environments offer only an "all peers are equals" model, which makes central control hard to achieve. The same can be assessed for spontaneous networks. Coordination and cooperation technologies offer a shared tuplespace as a central, simplified cooperation space. However, they are not specifically designed for dynamic service deployment and standard lookup support is not provided.

We propose a novel middleware concept for SOAs and claim that this concept addresses the holistic set of requirements summarized above. Even though each of the individual requirements has been solved in various different ways before, this problem is challenging because up till now an integration concept for the various concepts and solutions is missing. In contrast to the earlier approaches, we use software patterns, which describe the proven practices from existing middleware implementations and concepts, as a conceptual foundation. Our claim regarding this design approach is that using software patterns as a foundation for a novel middleware design leads to a design that combines the benefits of those other middleware implementations and concepts – with regard to a specific set of requirements.

There are the following key contributions: We present a novel SOA middleware concept that is based on a federated model of remote objects. 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. We also present a novel semantic lookup service concept that is aligned with the SOA middleware concept: Each peer provides semantic metadata about itself to its federation's lookup service. Peers can perform lookups in all lookup services of their federations. The lookup service enables loosely coupled services and simple self-adaptations for interface or version changes. The developer can extend the lookup service both with domain-specific ontologies and queries.

Our approach has the limiting assumption that the requirements, listed above, are necessary for the targeted application. We found them to be rather typical in the field of business services, and our solution is optimized in this direction. Of course, there are other application areas, where other requirements are more relevant and hence another middleware approach should be preferred. Our general design approach, to use software patterns, is only applicable in fields which have been documented pretty completely in terms of patterns. This is the case for the middleware field, but in many other areas, the pattern literature is less comprehensive. Hence, to follow the general pattern-based design approach, the patterns in those other areas would have to be mined first, which is a significant effort.

In this article, we first give a motivating example in Section 2. Then in Section 3 we explain our approach to use patterns for the reuse of design knowledge from other technologies and our earlier works on SOA patterns as foundations of this article. Next, in Sections 4-6 we describe the pattern-based design of the main architectural components of our SOA middleware approach. In Section 7, we provide the implementation details of our prototype implementation and in Section 8 a larger example using the SOA middleware for the composition of components in business processes (resolving the motivating example from Section 2). Finally, we relate our approach to other work, evaluate and discuss our approach, and conclude.

2. MOTIVATING EXAMPLE: FLEXIBLE ASSEMBLY OF BUSINESS PROCESS COMPONENTS

Let us consider a non-trivial example of using a SOA middleware for a company that sells custom applications built upon workflow solutions and other business components. For this article, we want to discuss the example of business processes for marketing measures and campaigns, but the same business components should also be applied in other business scenarios. The company typically uses a Java-based workflow engine to realize business processes that invoke a number of other systems as activities (tasks) in the workflow. These other systems are written in a number of languages and/or running on different platforms.

We require adapters and interfaces for the workflow engine and for all other systems. In the concrete example, we need to connect to a document archive system, a form input system, a PDF generator, and a PDF printing system. A workflow engine must be able to send messages to connected systems (i.e. it must have service-based adapters to access these systems), but it also must be able to receive invocations because results from the connected systems must be received asynchronously (i.e. the workflow engine also offers a service-based interface). In addition, often workflows in different workflow engines need to be composed. This is for instance done to separate long-running business processes from short-running, technical processes (see [Zdun et al. 2006]).

A typical business process for a marketing measure or campaign is started by a trigger event, such as a manual start or a customer inquiry via a Web site. Next, one or more activities collect the input data for the business process. That is, either inputs are needed from human agents and/or inputs are received from a database. Using this data, PDF files are generated, printed, and sent out. All inquiries and documents are also archived in the document archive system.

Consider we want to use Web services technology to realize such a system. First, we need to write adapters and interfaces for all business components. We must also write adapters and interfaces for the workflow engines. Depending on the engine and the Web services framework this might be difficult because an integration of the engine with the application server is needed and the engine must be prepared for asynchronous communication – as a client and a server. In most cases, this should be possible, though.

The example includes long running business processes with human involvement, which cannot easily be stopped. Nevertheless, new components or new versions of components might get added at runtime. Web service frameworks offer suitable means for dynamically invoking new services, but as we cannot easily extend lookup queries with the concern of component versions, we must built this abstraction as a service from scratch. Also, it might get difficult to adapt interfaces centrally, because central control is only given if all services run on the same server. That is, if we scale the system to multiple servers, we also need to realize an interface adaptation system from scratch.

The most important problem is that there are many typical recurring abstractions which must be built from scratch on top of the Web services framework – which is costly and error-prone. For instance, we must develop a concept how to integrate processes and services. This is a non-trivial effort, which includes to find suitable abstractions for different types of service-based activities and business object integration. Additional business abstractions, such as business units, business unit boundaries, and roles in business units, also must be hand-built, and all domain-specific control requirements must be implemented for the Web services on the application server. Also, the interaction of services with humans and their organizational roles must be built as services. All these issues relate to the problem that the client/server model does not really reflect the cooperation model of the business problem. There are systems available which solve some of these issues on top of Web services, but developers must then understand a number of technologies and integrate them.

Similar problems occur for realizing additional technical requirements. Consider, for instance, an extension of the lookup service to support load balancing or security roles, is needed. This means that a server side Web service must be implemented that realizes this new functionality because the lookup queries and query models in typical lookup servers used for Web services (such as UDDI) are not extensible.

Finally, the integration with other service-based and non-service-based applications, like the IT systems of other departments or legacy systems is needed. Adapter Web services must be implemented, and many integration issues must be built from scratch in these services.

In all these examples, it is non-trivial to ensure that quality attributes, such as security, performance, scalability, complexity, or performance, are not influenced negatively during the implementation of all these additional Web services.

For all these issues, individual solutions exist, as well as proven practices. However, so far no SOA middleware addresses all of these issues with an easy to use concept. Instead it is necessary to implement many of the recurring requirements discussed in the previous section separately as services. On the other hand, it is impossible to realize a middleware that can address all possible business requirements that might arise in any business application. However, in this article, we argue that the requirements in this example application are rather typical and recurring for applications that include loosely coupled business services, and that they can be attributed to the generic requirements introduced in the previous section. Hence, it would be desirable that the middleware offers suitable generic abstractions for realizing these particular recurring requirements. This is the goal of the SOA middleware described in this article. To make sure that we really identify the recurring solutions to recurring problems, we have based our approach on software patterns (see next section) that describe the proven practices in the SOA field. To illustrate the suitability of our concepts, in Section 8 we will explain how to realize the example, introduced in this section, with our SOA middleware concepts.

3. PATTERNS FOR DESIGN REUSE

Software patterns provide concrete guidelines for the design of software architectures. Software patterns capture reusable design knowledge and expertize that provides proven solutions to recurring software design problems that arise in particular contexts and domains [Schmidt and Buschmann 2003]. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [Alexander 1979]. A pattern language is a collection of patterns that solve the prevalent problems in a particular domain and context, and, as a language of patterns, it specifically focuses on the pattern relationships in this domain and context. As an element of language, a pattern is an instruction, which can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant [Alexander 1979]. Software patterns are a systematic reuse strategy [Schmidt and Buschmann 2003]: in a pattern successful software models, designs, and/or implementations that have already been developed and tested are documented so that they can be applied to similar software problems. Thus software patterns in the first place help to avoid the costly cycle of re-discovering, re-inventing, and re-validating common software artifacts.

As explained before, our goal is to combine successful solutions from different technologies, including middleware, Web services, P2P systems, coordination and cooperation technologies, spontaneous networking, and the Semantic Web. In different previous works we and others have mined the relevant patterns in these technologies. These are especially:

—In the POSA book [Buschmann et al. 1996] general architectural patterns, such as LAYERS[1] or BROKER are documented. The LAYERS pattern separates responsibilities by decomposing systems into groups of subtasks in which each group of subtasks operates at a particular level of abstraction. The BROKER pattern mediates object invocations among communication participants. Both patterns are used as an architectural foundation for our SOA middleware architecture.

—The Remoting pattern language [Voelter et al. 2004] extends the general architecture, described in the BROKER pattern, to address the full range of how to use, extend, integrate, and build distributed object middleware systems. Our SOA middleware follows the Remoting pattern language closely.

—There are a number of other general architectural patterns that are used in the SOA middleware architecture. In [Avgeriou and Zdun 2005] these patterns from different sources are explained as a coherent pattern language. The original pattern sources include [Buschmann et al. 1996; Gamma et al. 1994; Hohpe and Woolf 2003; Shaw and Clements 1997].

—The POSA 2 and POSA 3 books [Schmidt et al. 2000; Kircher and Jain 2004] describe many patterns used to implement distributed systems, especially at the communication protocol layer in distributed object middleware and for resource management. These are used at the lower-level layers of our SOA middleware.

—A SOA middleware usually requires the integration of languages and components because it should be independent of language and platform details and be able to glue other systems' components (e.g. legacy systems). Patterns for component and language integration [Zdun 2004b; 2006; Goedicke et al. 2000] are hence used at higher-level layers of our SOA middleware architecture.

—A SOA middleware also has high demands for adaptation to different components, platforms, configurations, etc. At higher-level layers of the SOA middleware architecture this is handled by patterns for aspect-oriented programming and software adaptation [Zdun 2004a; 2003].

These are the most important patterns and pattern languages that influenced the design of our concepts for a SOA middleware (explained below). Please note that the named pattern collections and languages contain a great number of patterns

[1]In the following text, we highlight patterns names in SMALLCAPS font and explain them each with 1-2 sentences in the text. Please refer to the original pattern sources for a full pattern description.

with many possible combinations. To aid the composition of these patterns and pattern languages to novel SOA middleware architectures, we have documented the proven practices how these patterns are used in existing SOA systems as a survey of patterns for SOAs (see [Zdun et al. 2006]). This survey of patterns for SOAs is the conceptual foundation for the most important design decisions explained below. We introduce the patterns and their dependencies in the SOA context implicitly in the text below.

To illustrate the pattern concept, let us provide one abbreviated example of a pattern description: the BROKER pattern [Buschmann et al. 1996]. The BROKER pattern [Buschmann et al. 1996] is, from the perspective taken in this article, a compound pattern that is typically implemented using a number of patterns from the Remoting Patterns language [Voelter et al. 2004].

Name: **Broker**
Context: A distributed object system should be designed. That is, objects should be offered by a server process to clients that access them across a network.
Problem: Distributed software system developers face many challenges that do not arise in single-process software. One main challenge is the unreliable communication across networks. Other challenges are the integration of heterogeneous components into coherent applications, as well as the efficient usage of networking resources. If developers of distributed systems must master all these challenges within their application code, they might loose their primary focus, to develop distributed applications that solve the application problems well.
Forces: The communication across the network is more complex than local invocations: network connections must be established, invocation parameters must be sent over the network, and network-specific errors like network failures must be handled.
It should be avoided that aspects of remote programming are scattered across the application code.
The network address and other parameters of the server should not be hard-coded in the client application. This is important to allow a distributed service to be realized by other servers, without having to change the clients.
Solution: Separate the communication functionality of a distributed system from its application functionality by isolating all communication related concerns in a BROKER. A BROKER hides and mediates all communication between the objects or components of a system. Local BROKERS on client side and server side enable the exchange of requests and responses between the client and the remote object. A BROKER consists of a client-side REQUESTOR to construct and forward invocations, as well as a server-side INVOKER that is responsible for invoking the operations of the target remote object. A MARSHALLER on each side of the communications path handles the transformation of requests and responses – from programming-language native data types into a byte array that can be sent across the network.
Consequences: A BROKER has the advantage that it simplifies distributed communication. The BROKER infrastructure can be reused by different distributed applications. Because the BROKER is responsible for locating the distributed object/server via a name or ID, the BROKER enables location transparency.
The BROKER architecture also incurs the liability that it offers a little less performance and has a higher resource consumption than a well-designed application in which static objects are directly bound to the network. A BROKER has a certain complexity that must be un-

derstood by developers. For very simple distributed applications, such as some embedded systems, more simple architecture might perform as well as the BROKER architecture, but are easier to understand and maintain. For most other distributed applications, such as enterprise systems, the use of a BROKER is advisable.

4. PATTERN-BASED DESIGN OF A FEDERATED PEER SERVICES ARCHITECTURE

In this section, we will step-by-step discuss our concepts for peer federations. We illustrate our concepts with examples from our prototype implementation Leela [Zdun 2005a]. Leela is implemented in XOTcl [Neumann and Zdun 2000], an object-oriented scripting language based on Tcl, and uses SOAP-based communication. The pattern-based design has the aim that a similar framework can be implemented in any language with any Web services framework. The framework is designed to be extensible and implementation decisions, such as using a particular SOAP implementation as the communication protocol, can be changed, if required.

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.

4.1 Basic communication framework

As its basic communication resource, each Leela application uses a class implementing both a CLIENT REQUEST HANDLER [Voelter et al. 2004] and a SERVER REQUEST HANDLER [Voelter et al. 2004]. These two patterns are responsible for the basic tasks of establishing connections and message passing between client and server. That is, the class `RequestHandler` is responsible for sending requests across the network and receiving responses (see Figure 1). At the same time it receives incoming requests, dispatches the requests into the server application, and sends 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 (a subclass of the request handler) can be accessed by each peer.

A SOA should be independent from the protocol used internally. Thus, at the communication layer, represented by the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER, we require a high flexibility regarding the protocols used. These communication protocols might require different styles of communication, such as synchronous RPC, asynchronous RPC, messaging, publish/subscribe, and others. Variation at the communication layer is usually handled via PROTOCOL PLUG-INS [Voelter et al. 2004]. PROTOCOL PLUG-INS extend the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER with support for multiple, exchangeable communication protocols to be configured from the higher layers. Leela's request handler contains PROTOCOL PLUG-INS [Voelter et al. 2004] for different protocols that actually transport the message across the network. For each protocol supported, the `RequestHandler` instantiates a pair of corresponding plug-ins, a client plug-in for sending requests and receiving responses, and a server plug-in for receiving incoming requests and sending responses.

Currently, Leela supports SOAP [Box et al. 2000] and socket PROTOCOL PLUG-INS. However, any other communication protocol can be used as well. As described below, Leela supports different invocation and activation styles (see Sections 4.2 and 5.1). Thus the specialties of most protocols supporting mainstream communication models, for instance following the REMOTE-PROCEDURE-CALL [Avgeriou and Zdun 2005; Shaw and Clements 1997] or MESSAGE QUEUING [Avgeriou and Zdun 2005; Hohpe and Woolf 2003] styles, 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 – with different trade-offs.

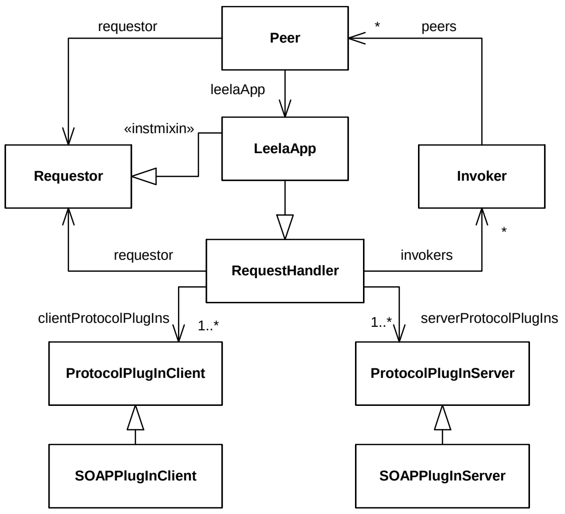

Fig. 1. Structure of the Leela communication framework

Remote invocations are abstracted by the patterns REQUESTOR [Voelter et al. 2004] and INVOKER [Voelter et al. 2004]. The client-side REQUESTOR constructs and forwards invocations, and the server-side INVOKER calls the target remote objects' operations. The Leela REQUESTOR is responsible for building up remote invocations at runtime and for handing the invocation over to the `RequestHandler`, which sends it across the network. The REQUESTOR offers a dynamic invocation interface, similar to those offered by OO-RPC middleware approaches such as CORBA or RMI.

A CLIENT PROXY [Voelter et al. 2004] is a placeholder for the remote object in the client process. By presenting clients an interface that is the same as the remote object's, the proxy lets the client interact with the remote object as if it were a local object. Internally, the client proxy transforms the invocations it receives into REQUESTOR invocations. In addition to the dynamic invocation interface, Leela also supports peer and federation proxies that can act as a CLIENT PROXY, offering the interfaces of a remote peer or federation.

The INVOKER gets the invocation from the `RequestHandler` and performs the invocation of the peer. In Leela, there are different INVOKERS for different activation strategies (see Section 5.1). The `RequestHandler` 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 [Voelter et al. 2004]. INVOCATION INTERCEPTORS are automatically triggered before and after request and response messages pass the REQUESTOR and INVOKER. The interceptor intercepts the message at these spots and can add services to the invocation. That is, the invocation on both sides can be transparently extended 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 then read and handle this information.

The REQUESTOR, INVOKER, and request handler handle synchronization issues on client and server side. The request handler handles the invocations according to the invocation and activation styles used. On server side, the `RequestHandler` receives network events asynchronously from a REACTOR [Schmidt et al. 2000]. A REACTOR observes network events and reacts on them using callbacks. 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. Depending of the activation strategy, the access of a particular peer can either be queued (synchronized) or handled by a multi-threaded pool of peers (following the POOLING pattern [Kircher and Jain 2004]). The results are queued again, and handed back to the receiving thread.

It is pretty easy to make an ordinary class a peer service in Leela. On server side a script containing the class implementation is handed to a peer, e.g. upon its creation. (In larger examples the class would rather be loaded on demand from a package.) For instance, the following code is a peer that uses a class that calculates the current date and time:

```
Peer dateService -script {
    Class dateService
    dateService instproc dateAndTime args {
        clock format [clock seconds] -format %c
    }
}
```

To make this peer an exported service, we simply have to export the methods of the service remotely:

```
dateService export {dateAndTime}
```

4.2 Invocation styles

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 handler uses an event loop to queue up incoming and outgoing requests in a message queue. Client-side invocations run in a separate thread.

The result arrives asynchronously on client side and has to be obtained from the receiving thread. This is done by raising an event in the `RequestHandler's` event loop. This event executes a callback specified during the invocation. An ASYNCHRONOUS COMPLETION TOKEN (ACT) [Schmidt et al. 2000] is used to map the result to its invocation. The ACT pattern is used to let clients identify different results of asynchronous invocations when the response has arrived. Using this callback model we can implement three different asynchronous invocation styles (following the patterns from [Voelter et al. 2004]):

—A FIRE AND FORGET invocation describes best effort delivery semantics for asynchronous operations but does not convey results or acknowledgments.

—A POLL OBJECT describes invocation semantics that allow clients to poll (query) for the results of asynchronous invocations, for instance, in certain intervals.

—A RESULT CALLBACK describes invocation semantics that actively notify the requesting client of asynchronously arriving results using a callback.

Both, POLL OBJECTS and RESULT CALLBACKS, are informed about events using the same `Callback` interface. For instance consider we want to invoke the date and time example peer (from the previous section) using a RESULT CALLBACK. First, we need to define the result callback behavior. We need only to implement a simple interface with two methods `inform` and `error`:

```
ResultCallback DateCB
DateCB instproc inform {act result} {
    puts "Date ($act) = $result"
}
DateCB instproc error {act m i} {
    puts "ERROR: $act $m $i"
}
DateCB rc
```

The class `DateCB` simply prints all results to the screen. We create a RESULT CALLBACK object `rc` from this class. Now we can construct a request invocation object using the REQUESTOR that is configured with this callback object, and then send the invocation to the server:

```
RequestInvocation ri -aor [httpAOR make localhost 8015 dateService] \
  -method dateAndTime -act $myACT -arguments {} -callback rc
leelaApp send ri
```

4.3 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 a unique OBJECT ID [Voelter et al. 2004] 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. Other protocols, however, require different location information. The OBJECT ID plus location information implement the pattern ABSOLUTE OBJECT REFERENCE (AOR) [Voelter et al. 2004] – a unique reference for the particular service in the network. In Leela AORs are abstracted using a class `AORHandler`. Subclasses of this class implement the protocol-specific AORs, e.g. the class `URL_AOR` implements URL-based AORs as used in the SOAP PROTOCOL PLUG-INS. All Leela classes at LAYERS above the PROTOCOL PLUG-INS, such as the `Invocation` classes, use only the generic `AORHandler` interface, so that they can be used with different protocols despite different kinds of AORs used in these protocols.

In addition to the invocation and location information, the invocation might contain INVOCATION CONTEXT [Voelter et al. 2004] data. The INVOCATION CONTEXT 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.

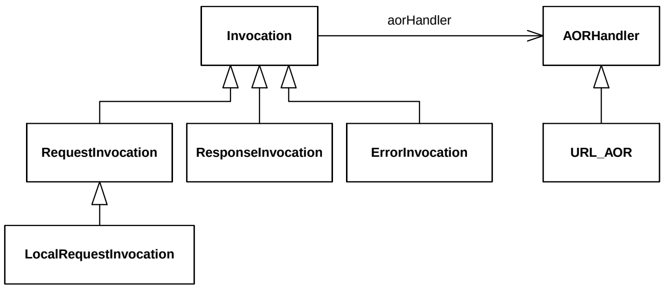

Fig. 2. Invocation types and Marshallers

Leela sends the message payload as a structured string (we use a format similar to Tcl lists). These strings are different for different invocation types. Currently, Leela supports request, response, and error invocation types (see Figure 2). The scheme is extensible with any other kind of invocation type. The error message type is used to implement the pattern REMOTING ERROR [Voelter et al. 2004] – we use it to signal remoting-specific error conditions in the Leela framework. Additionally, Leela supports local request invocations. The local request invocations are faster than ordinary request invocations because they do not have to go through the network stack, but only through the local message queue of the request handler.

A MARSHALLER [Voelter et al. 2004] on each side of the communications path handles the transformation of requests and responses from programming-language-native data types into byte arrays that can be sent over the wire. 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. The protocol-specific marshaling of AORs is handled by the respective subclasses of `AORHandler`. Marshaling the AOR is need, for instance, because the AOR of the invoking peer must be transported with the request so that the response can be sent to the correct peer.

5. FEDERATIONS AND PEERS

A federation is a concept to manage remote objects in a remote object group (the federation members are called peers). Each federation has one federation object that manages the federation data consistently. From the viewpoint of the peers, federations are, among other things, an implementation of the pattern CONFIGURATION GROUPS [Voelter et al. 2004] because the federation imposes configurations and INVOCATION INTERCEPTORS on its peers. 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 to and removed from a federation.

A federation can be accessed remotely by a federation proxy. This is a special CLIENT PROXY 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.

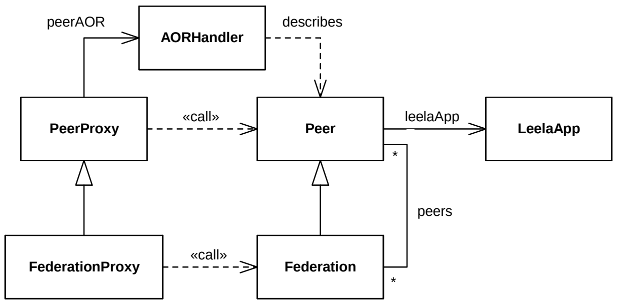

Fig. 3. Peer and federation structures

The peers can construct invocations using the REQUESTOR of the Leela application. 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, the peer proxy allows a peer to interact with a remote peer using the peer's interface instead of using the REQUESTOR for invocations. The federation and peer structures are shown in Figure 3.

Federations are just instantiated like all other peers. For instance, we can create a federation `dateFed` for the date service peer example from above and add the `dateService` peer to it:

```
Federation dateFed
dateFed addPeers {dateService}
```

From now on the federation controls access to the peer (using the federation interceptors explained below). A client peer that wants to invoke this peer must first join the federation. The client can try this by instantiating a federation proxy to the federation and by trying to add one of its peers to it:

```
FederationProxy fp -serviceName dateFed -aor $federationAOR \
  -callback rc -act $fedACT
fp addPeers {myDateClient}
```

If the federation allows the peer to join the federation, `myDateClient` can invoke methods exported by `dateService`. Otherwise access is denied.

5.1 Peer activation

Activation means creation and initialization of a remote object so that it can serve requests. In a loosely coupled remoting environment, activation of remote objects is a critical issue because server resources (especially memory) should not get wasted, and the peers should handle requests efficiently. Leela supports the following activation STRATEGIES [Gamma et al. 1994] (see Figure 4) which are triggered and managed by the LIFECYCLE MANAGER [Voelter et al. 2004]:

—STATIC INSTANCE: The peer is already activated before it is exported and survives until it is explicitly destroyed or the Leela application stops.

—PER-REQUEST INSTANCE: The class of the peer is exported, and the peer is activated when the request arrives. Then this peer handles the request and is de-activated again. PER-REQUEST INSTANCES use the pattern POOLING: they are pre-initialized in a pool to reduce the activation overhead for instantiation.

—CLIENT-DEPENDENT INSTANCE: A FACTORY METHOD [Gamma et al. 1994] 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 INSTANCE is not needed anymore, except the client explicitly destroys it. If a given object is not accessed for a while this 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 LEASING pattern [Voelter et al. 2004; Kircher and Jain 2004] helps the federation to decide whether a certain client-dependent peer is still required. For each client-dependent peer a lease is started when the peer 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 lifecycle operation `renewLease`. When the lease expires and is not renewed, the client-dependent peer gets removed from the federation.

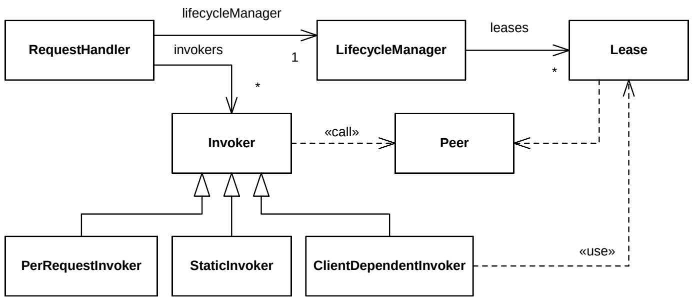

Fig. 4. Activation strategies implemented on Invokers

In Leela the LIFECYCLE MANAGER pattern – implementing the management of the activation STRATEGIES – is triggered by the `RequestHandler` when an invocation arrives. Peers are registered with an activation strategy. There are different INVOKERS for the different activation STRATEGIES. The appropriate INVOKER is also chosen by the `RequestHandler`.

Programmatically, it is pretty simple to change the lifecycle strategy of a peer. The strategy is provided using the method `lifecycleStrategy`, and then the `RequestHandler` and the LIFECYCLE MANAGER select the correct INVOKER. For instance, the following code instantiates two peers with different lifecycle strategies:

```
Peer A -script $initScriptA -lifecycleStrategy PerRequest
Peer x -script $initScriptX -lifecycleStrategy Static
```

5.2 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, and each of these threads has its own INTERPRETER [Avgeriou and Zdun 2005; Gamma et al. 1994] as THREAD-SPECIFIC STORAGE [Schmidt et al. 2000]. Thus peers have no direct access to the main INTERPRETER or to other local peers. The peer INTERPRETERS are synchronized by message queues (following the MESSAGE QUEUING style [Avgeriou and Zdun 2005; Hohpe and Woolf 2003]) implemented by event loops of the INTERPRETERS. I.e., the peer threads can only post "send" and "result" events into the main INTERPRETER, and the request handler decides how to handle these events.

A 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 per default 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. On client side, an INVOCATION INTERCEPTOR intercepts the construction of the remote invocation and adds all federation information for a peer into the INVOCATION 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 any 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 of peers. E.g., we can introduce an interceptor that only grants access if some security credentials, such as user name and password, are sent with the invocation. Figure 5 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 instance pool or dealing with the lease.

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

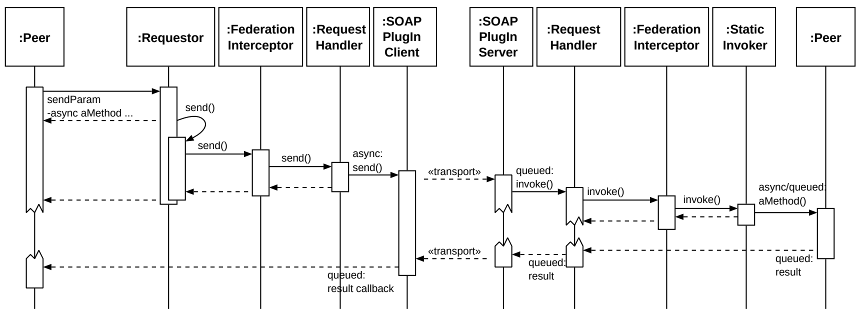

Fig. 5. Sequence diagram for a simple invocation with a static Invoker

cannot do. Optionally, peers can act as a "bridge" to another federation – offering some of that federation's services in the context of its own federation.

5.3 Peer component integration using a Command Language architecture

As mentioned above, each peer runs in its own INTERPRETER bound to its thread of control. This way we can ensure that peers cannot perform actions malicious to the main INTERPRETER or other peers. The peers, however, need access to some objects from the main INTERPRETER, namely the Leela application and the REQUESTOR. This problem is solved using the patterns COMMAND, COMMAND LANGUAGE, and OBJECT SYSTEM LAYER:

—A COMMAND [Gamma et al. 1994] encapsulates an invocation to an object and provides a generic invocation interface. This interface contains an operation to execute the encapsulated invocation (e.g. called `execute`, `invoke`, or `run`).

—A COMMAND LANGUAGE [Zdun 2004b; 2006] is a language composed of COMMANDS. These are not invoked directly using an API, but via the COMMAND LANGUAGE'S language elements. Each COMMAND is accessed with a unique command name.

—An OBJECT SYSTEM LAYER [Goedicke et al. 2000] is an object system that is provided as a language extension in the host language. It provides a well-defined interface to components that are non-object-oriented or implemented in other object systems. These components are accessible through the OBJECT SYSTEM LAYER, and the components can be treated as black-boxes.

Leela is implemented in XOTcl, a COMMAND LANGUAGE which realizes an OBJECT SYSTEM LAYER. That is, each object in XOTcl is realized itself as a COMMAND of the COMMAND LANGUAGE. For the COMMANDS it is possible to declare an alias that forwards invocations from one INTERPRETER context to another. In each peer INTERPRETER, we provide such an alias COMMAND, which is a PROXY for the Leela application object. This `LeelaAppProxy` object can be used by the peer objects to raise asynchronous events in the main INTERPRETER. That is, peers can send `Invocations` using the REQUESTOR. There is no other way for the peers to access either the main INTERPRETER or other (local or remote) peers.

We have mentioned above that the Leela INVOCATION INTERCEPTORS need to be quite flexible regarding the objects they might extend. We potentially need extensions of peers, peer types, INVOKERS, REQUESTORS, PROTOCOL PLUG-INS, or even whole Leela applications. Note that these objects might be implemented in different INTERPRETERS. Many frameworks use a chain of INTERCEPTOR objects to realize an interceptor architecture (examples are: .NET [Microsoft 2003], CORBA [Object Management Group 2004], or Apache Axis [Apache Software Foundation 2004]). In order to use an interceptor chain, the objects which are extended by the interceptors must be specifically prepared. This, however, is rather inflexible and requires us to foresee any kinds of objects that might require interception.

We used the patterns for aspect-oriented programming and software adaptation from [Zdun 2004a; 2003] to remedy this problem. In particular, in our Leela prototype, the pattern MESSAGE INTERCEPTOR realizes the INVOCATION INTERCEPTORS in the local INTERPRETER context. All Leela INTERPRETERS are realized using the XOTcl OBJECT SYSTEM LAYER, and each OBJECT SYSTEM LAYER contains a MESSAGE REDIRECTOR that dispatches all invocations to the objects in the OBJECT SYSTEM LAYER. Instead of hooking a chain of INTERCEPTORS into the invocation path at the REQUESTOR and INVOKER (as done in .NET, CORBA, or Apache Axis), we realize the MESSAGE INTERCEPTORS in the XOTcl MESSAGE REDIRECTOR. We can thus register MESSAGE INTERCEPTORS for each object under the control of a MESSAGE REDIRECTOR. INVOCATION INTERCEPTORS can be distinguished from other MESSAGE INTERCEPTORS because they are derived from the classes `InterceptorClient` or `InterceptorServer`, which provide the interfaces to intercept invocations in the client or the server request flow respectively.

This short discussion should have illustrated that our Leela prototype has heavily benefited from using internally an architecture that combines an OBJECT SYSTEM LAYER and a COMMAND LANGUAGE. This architecture can be used for many other useful tasks that are needed throughout the framework. For instance, we provide an AUTOMATIC TYPE CONVERTER [Zdun 2004b; 2006] for converting to/from strings and lists (used in the SOAP PROTOCOL PLUG-INS). INTROSPECTION OPTIONS and an accessible CALLSTACK are provided to automatically obtain and maintain context information. In our prototype, for most of these tasks we reuse the facilities of the scripting languages XOTcl and Tcl.

This architecture is also useful for component integration: a COMPONENT WRAPPER [Zdun 2004b; 2006] to a component written in another language (such as C, C++, and Java) can easily be built: we realize the COMPONENT WRAPPER as an object of the OBJECT SYSTEM LAYER, which is also a COMMAND. The COMMAND abstraction is used to forward operations from the COMPONENT WRAPPER object (i.e. the Leela peer) to the component implementation in the foreign language. This way, Leela peers can also be used to offer services realized by external components (e.g. from legacy systems).

6. LOOKUP IN LEELA REMOTE OBJECT FEDERATIONS

Time and again, services in distributed systems get added or removed. Clients need to know which services are currently offered by which server application, running at which location. To allow for efficient, inexpensive, and timely publication of services, the LOOKUP pattern is used as a service by most middleware frameworks. Typical implementations of lookup services, such as naming or trading services, can be queried using a name or a set of properties. Then, the service searches in a table (or some other data structure) for matching elements. As a result of the lookup, the ABSOLUTE OBJECT REFERENCE of a remote object is returned, or other location information such as an INTERFACE DESCRIPTION like a WSDL file.

Today, we see many new requirements for lookup services arising that are due to new kinds of applications for these services and new lookup requirements – especially in SOAs. For instance, in Leela we require a highly dynamic lookup repositories with services that can connect, disconnect, move, and change their properties dynamically. In many existing lookup implementations, however, the properties and relationships of elements in the lookup service are too simple or hard to extend. Sometimes lookup queries need to be constructed or modified at runtime as well – for instance, to construct complex lookup queries based on previous lookup results. We also want to allow for more advanced lookup functions – examples are ontology-based queries, behavioral extension, query extension, load-balancing, and fault tolerance. This requires a high flexibility regarding inputs, processing, and outputs of the lookup service.

This section presents a semantic lookup service which we have developed to cope with these challenges (see [Zdun 2005b] for more details). There are two simple alternative solutions for a semantic lookup service design. First, we can use a semantics-based metadata services, such as Triple [Sintek and Decker 2002], Ontobroker [Decker et al. 1999], or Fact [Horrocks 2001], that provides more powerful metadata definitions and queries. However, this approach is not well suited for all application areas, because the solutions might get hard to understand and/or hard to debug. In addition, the semantics-based metadata services do not (yet) offer a concept to define and redefine at runtime domain-specific ontologies and query languages.

A second alternative is to use an ordinary naming or trading service, and perform the additional computations on client side. The benefit of this solution is that clients can formulate customized queries in an imperative language. This is easy to use for developers used to such languages, but as a liability for complex queries a lot of data needs to sent across the network and/or a lot of queries might have to be performed. For example, the complex query "get all resources that have the property X and are connected using the property 'sub-class-of' transitively" would yield multiple atomic queries across the network. It is hard to extend the lookup with domain-specific abstractions or queries because before a newly developed lookup abstraction or query function can be used, it must be deployed to all clients and servers. In many middleware platforms this is not possible without stopping the running applications.

As a solution to these problems, we propose the lookup service architecture illustrated in Figure 6. It consists of the following three main components:

—*RDF store:* All metadata about the services is stored in an RDF store. RDF [W3C 2004] supports semantic metadata about Web resources described in some ontology or schema. For instance RDF Schema [Brickley and Guha 2004] and OWL [McGuinness and Harmelen 2004] support general relationships about resources, like "subclass of". Developers can also use RDF ontologies from other domains; for instance, in an e-learning system an ontology for learning materials can be used, such as LOM [IEEE WG 12 2004]. We use the Redland RDF store

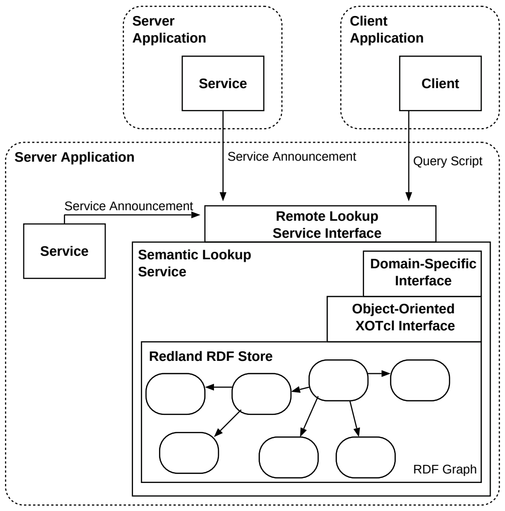

Fig. 6. Architecture overview of the semantic lookup service

[Beckett 2004]. The RDF store component allows developers to create and use domain-specific ontologies that can be deployed at runtime.

—*Script interpreter:* The RDF store is accessed using an XOTcl INTERPRETER. The INTERPRETER can evaluate imperative scripts on server side. Two interfaces to the RDF store are offered: one can change the data in the store, the other one is used for querying only. The Redland RDF store is implemented as a set of C libraries. XOTcl is used as an OBJECT SYSTEM LAYER, which provides an object-oriented interface to the Redland store. I.e., the lookup service architecture is based on the same design as the foundation of the Leela architecture (see Section 5.3). The script interpreter component allows developers to define domain-specific query languages for the domain-specific ontologies. It also enables changes (i.e. redeployment) to the query language definitions at runtime.

—*Remote interface:* A remote query interface is offered so that remote clients can send scripts to the lookup service. The lookup service evaluates the scripts using its embedded script INTERPRETER. This component allows developers to query the data defined in the (domain-specific) ontologies at runtime using the (domain-specific) query languages offered by the lookup service.

The class diagram in Figure 7 gives an overview of the object-oriented interface to the Redland RDF store library. Each class depicted in this figure is a COMPONENT WRAPPER that wraps one of the Redland C libraries. In particular, all elements in one RDF store are managed by a "world" object. The world can either be filled by hand (programmatically) or by an XML parser. The RDF metadata is stored in an RDF model. The model consists of RDF statements (triples of RDF nodes for subject, predicate, and object). Some of these nodes are URIs, describing Web resources. Different storages exist in which the metadata can be stored, including the memory, files, and relational databases. Using the Model class, statements can be searched for some criteria. Such statements can be traversed using the Stream class. Individual lists of nodes can be traversed using the Iterator class.

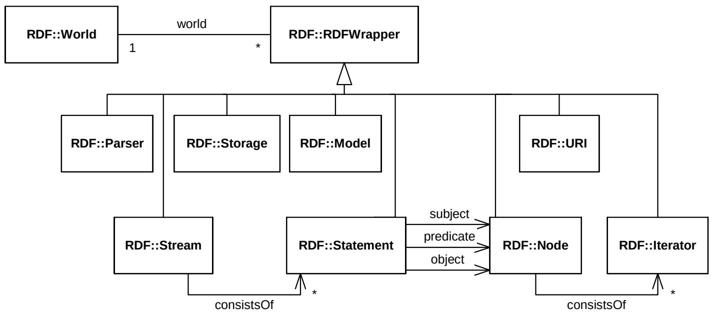

Fig. 7. Object-oriented RDF store interface

Based on this infrastructure we can build the lookup and query models. In fact, the basic lookup model is just the object-oriented COMPONENT WRAPPER interface. The basic model can be used for simple RDF graphs with ABSOLUTE OBJECT REFERENCES, properties, and relationships. In the case of ordinary Web services, we simply announce the services with a service name, their WSDL INTERFACE DESCRIPTION, and other information like the service's URL. More sophisticated descriptions might contain properties provided as domain-specific ontologies. For instance, information in the WSDL file (like operation names, supported protocols, etc.) can be reflected in the RDF graph as well.

Consider a simple example: we want to describe the "creators" of some peers as a simple literal property. The following script defines the creators using a respective predicate from the Dublin Core ontology [Dublin Core Metadata Initiative 2004]:

```
Statement s1 "http://www.exampleservice.org/aService" \
    "http://purl.org/dc/elements/1.1/creator" "Dave"
Statement s2 "http://www.exampleservice.org/anotherService" \
    "http://purl.org/dc/elements/1.1/creator" "Jim"
Statement s3 "http://www.exampleservice.org/yetAnotherService" \
    "http://purl.org/dc/elements/1.1/creator" "Jim"
rdfModel add {s1 s2 s3}
```

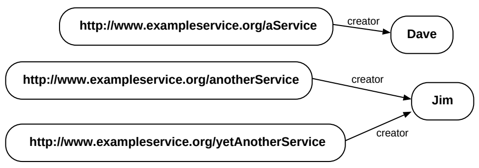
http://www.exampleservice.org/aService
creator
Dave
http://www.exampleservice.org/anotherService
creator
Jim
http://www.exampleservice.org/yetAnotherService
creator

Fig. 8. Example RDF graph

A peer can send this script to the script INTERPRETER of the lookup service. Here, three statements, each containing two URIs and a literal node, are created and added to the default RDF model. The simple example model is shown in Figure 8. We can use this RDF model for queries later on. E.g., the following simple query produces a list of all the contents of all statements that have a creator (according to the Dublin Core ontology) and the value of the creator is “Jim”:

```
set results ""
Statement queryStatement -predicate "http://purl.org/dc/elements/1.1/creator" \
  -object "Jim"
set stream [m findStatements queryStatement]
while {![$stream end]} {
    lappend results [[$stream current] toString]
    $stream proceed
}
set results
```

If a client peer sends this query to the lookup service, the following result is returned:

```
{{[http://www.exampleservice.org/anotherService],
  [http://purl.org/dc/elements/1.1/creator], [Jim]}}
{{[http://www.exampleservice.org/yetAnotherService],
  [http://purl.org/dc/elements/1.1/creator], [Jim]}}
```

The lookup and query model can be extended using domain-specific ontologies. Extending the store with ontology definitions is pretty easy: simply the standard XML-based RDF definitions of the new ontology need to be provided (This is supported by Redland’s parser, similar to most other RDF stores). The more important step for defining ontologies, however, is to define the semantics of the ontology elements – that is, how they are interpreted.

There are two ways to define the interpretation of ontology elements in Leela. The simple solution is that the client provides the interpretation in the query script. Alternatively, if such queries are recurring, we can provide the implementation of query semantics on server side so that the client does not have to provide the query logic with each request, but can reuse the pre-defined functions instead. Such a domain-specific query interface must then be loaded into the server-side INTERPRETER before the query happens. Deployment of such changes are supported at runtime. Of course, this kind of server side extension is only needed for convenience. Any kind of query can also be defined at client side and be sent with each query – thus the query model is fully extensible without the need for deploying changes to the server. For instance, we could define a convenience method like the following to abstract the query in the example above:

```
Class DublinCoreQuery
DublinCoreQuery instproc findAllServicesbyCreator {creator} {
  set results ""
  Statement queryStatement -predicate "http://purl.org/dc/elements/1.1/creator" \
    -object $creator
  set stream [m findStatements queryStatement]
  while {![$stream end]} {
      lappend results [[$stream current] toString]
      $stream proceed
  }
  return $results
}
```

Now the client can use more simple queries by using this server-side, domain-specific query interface. E.g., the query for services created by “Jim” now looks as follows:

```
DublinCoreQuery query1 -findAllServicesbyCreator Jim
```

7. IMPLEMENTATION DETAILS OF THE PROTOTYPE IMPLEMENTATION

Even though we have based our design and the discussion above on patterns, instead of focusing on implementation-specific details, for the practical implementation of the concepts, it might be useful to explain some details of our prototype implementation. We will summarize notable details of our prototype implementation in this section. As we have already given examples of the use of peers and federations, we will specifically concentrate on details of the internal dispatch process for requests. Our prototype defines the central `RequestHandler` class as follows:

```
Class RequestHandler -parameter {
    {protocolPlugIn TclSOAP}
    {invokers {PerRequest Static ClientDependent}}
}
```

The request handler is pre-configured with parameters for its PROTOCOL PLUG-IN and the INVOKERS it can use. These parameters can be changed by the user, when instantiating a request handler. E.g., instead of using the TclSOAP plug-in, the C-based GSOAP plug-in or a pure Sockets plug-in can be selected. The `invokers` parameter allows users to extend the Leela framework with new INVOKER types.

The request handler contains methods for sending requests (client part) and handling incoming requests (server part). The client and server parts of the request handler are initialized when a request handler object is instantiated. For both parts a thread pool is created. Each thread in the thread pool contains a thread-specific interpreter, which only offers the relevant objects and classes for its tasks. In the case of the client threads, these are the client PROTOCOL PLUG-INS, which can be used by Leela peers to send messages. The server thread pool is instantiated with the server PROTOCOL PLUG-INS plus all the INVOKERS configured in the `invokers` parameter. Because of the thread-specific interpreters, a peer implementation containing errors or that is malicious can only affect its own thread-specific interpreter, but not other peers or main components of Leela.

As a public interface the request handler class contains the following methods:

```
RequestHandler instproc dispatch {serviceName type args} {...}
RequestHandler instproc addPeers {peerList} {...}
RequestHandler instproc export {sn {methodList "*"}} {...}
RequestHandler instproc info {option args} {...}
RequestHandler instproc sendResponse {marshalled aor} {...}
RequestHandler instproc send {invocation} {...}
```

The `dispatch` method implements the central dispatcher of Leela. For the `Request` or `LocalRequest` invocation types, first it is checked that the desired service is existing, then the LIFECYCLE MANAGER is invoked to activate the object and check whether a lifecycle operation is triggered. Finally, the invocation to the `Invoker` class is constructed and handed to the thread pool, which selects the first idle thread interpreter to handle the request. `Response` and `Error` invocations are treated similarly, but they invoke the `inform` or `error` callback methods of the callback object of the peer that has sent the request.

The method `addPeers` is used by federations to announce service names of peers at the request handler. `export` is used to announce those methods that can be accessed remotely. Only those service name/method combinations which are previously announced will be accepted by the `Invoker` class (see below). `info` is used to introspect the exported service names and methods, and other internal details of the request handler. Finally, `sendResponse` and `send` provide an protocol-independent interface for sending requests and responses. These methods are not used directly by peers, but via an event-based invocation model (see below).

Another central method is the method `invoke` of the `Invoker` class. It is responsible for checking if the invocation is valid, and then to select the appropriate INVOKER. Hence, the method first de-marshals the request invocation. Next, it checks whether the service exists. Then it checks whether the selected method has been exported for that service. If this all is the case, the service's INVOKER type is selected, and the request is handed to an `Invoker` object of that type. This is done by posting an event into the server thread pool. If the invocation causes no runtime error, a `ResponseInvocation` is constructed and sent to the client peer. In all other cases, `ErrorInvocations` are sent back.

```
Class Invoker
Invoker proc invoke {serviceExists serviceName serviceMethodList
  serviceObject serviceInvoker arguments} {
  ...
}
Class PerRequestInvoker -superclass Invoker
PerRequestInvoker instproc invoke {invocation} {...}
Class ClientDependentInvoker -superclass Invoker
ClientDependentInvoker instproc invoke {invocation} {...}
Class StaticInvoker -superclass Invoker
StaticInvoker instproc invoke {invocation} {...}
```

The `invoke` methods of the subclasses of `Invoker` are responsible for handling the invocations according to the invocation strategies. The `Invoker` class' method `invoke` selects the correct `Invoker` (using polymorphism).

The class `Requestor` supports different ways of sending a request. Basically, the `Requestor` methods construct a request invocation from their parameters or use a given `invocation` object, and send the invocation using the `RequestHandler`.

```
Class Requestor
Requestor instproc sendParam {aor source_aor callback
  act method arguments} {...}
Requestor instproc sendLocalParam {aor source_aor callback act
  method arguments} {...}
Requestor instproc send {invocation} {...}
Requestor instproc sendLocal {invocation} {...}
```

The Leela application class brings together the client and server views of a peer. It is a special request handler, which also implements the `LeelaAppInterface`. This interface basically specifies that the two abstract methods `send` and `sendLocal` must be implemented on any Leela application object. Typically, those are implemented by the `Requestor` class as a mixin. Additionally, the `FederationInterceptor` is configured as a mixin to enable application-wide federation support. The federation control can be turned off by de-registering the mixin.

```
Class LeelaApp -superclass {RequestHandler LeelaAppInterface}
LeelaApp instmixin {Requestor FederationInterceptor}
LeelaApp leelaApp
```

The peer class mainly offers a high-level interface for configuring a peer, such as the lifecycle strategy, the service name and object, the service method list, etc. A federation additionally provides ways to add peers to the federation.

It is important to note that peers, who reside in their thread-specific interpreters are not able to directly access the classes discussed so far. They get invoked by their INVOKER, and can only communicate with the main thread of the Leela application via events posted into the main event loop. The main event loop is thus also the primary device to synchronize the different threads. When a peer – in its thread-specific interpreter – accesses the object `leelaApp`, it does not access the `LeelaApp leelaApp` instance instantiated for the main thread in the program fragment above, but a `LeelaAppProxy`. The `LeelaAppProxy` can post specific events into the main event loop and implements the two methods `send` and `sendLocal` of the `LeelaAppInterface`. `send` and `sendLocal` of `LeelaAppProxy` are also implemented via event posting. That is, the only ways for a peer to communicate with the outside is to send events to the main thread or send invocations to other peers. In both cases the main dispatcher can decide how to handle these events.

```
Class LeelaAppProxy -superclass LeelaAppInterface
LeelaAppProxy instproc raiseEvent {args} {...}
LeelaAppProxy instproc send {invocation} {...}
LeelaAppProxy instproc sendLocal {invocation} {...}
```

All interceptors that should be used in the Leela invocation chain must implement the following interceptor interfaces:

```
Class InterceptorClient
InterceptorClient abstract instproc send {invocation}
Class InterceptorServer
InterceptorServer abstract instproc dispatch {serviceName type args}
```

This way interceptors can be applied as mixins in the dispatch process, like the `FederationInterceptor` on `LeelaApp` above. This interceptor implements both interfaces, because it adds federation information to the INVOCATION CONTEXT on client side and evaluates the federation information on server side:

```
Class FederationInterceptor -superclass {InterceptorClient InterceptorServer}
```

The semantic lookup service implementation provides the wrappers for the Redland library as detailed before. For the use in Leela, we have created a peer that binds the Redland-based lookup service to the Leela federation. This peer initially loads the Redland libraries and the wrappers into its thread-specific interpreter. This interpreter can only be accessed via the exported peer methods, and in the interpreter only the methods of the semantic lookup service can get executed. Like all peers, this peer can only communicate with the outside by the event-based methods offered by the `LeelaAppProxy` class. Hence, malicious query code can only affect its own sandbox. For our prototype, we have only dealt with vulnerabilities concerning the protection of the other Leela peers and the Leela application, i.e. vulnerabilities related to which commands can potentially get executed by query code. Further measures that could be taken are, for instance, to discontinue servicing clients that have caused multiple runtime errors in the lookup service or accepting only a maximum number of requests from a specific client at a time.

8. EXAMPLE RESOLVED: FLEXIBLE ASSEMBLY OF BUSINESS PROCESS COMPONENTS

Let us again consider the example from Section 2. We want to use Leela to interconnect the workflow engine running the business processes and the other systems.

To do so, we need to write Leela peers that work as adapters and interfaces for all components that participate in the workflow. First, we need an adapter and interface for the workflow engine itself to connect it to a Leela network of peers as a peer. Also the connected systems are each modeled as a peer: a document archive system, a form input system, a PDF generator, and a PDF printing system.

To leverage the flexible assembly of business processes, we instantiate one federation per business unit or organization. The peers of all business components connect to the marketing department's federation. The peers register themselves with the federation's lookup service and store metadata about themselves. We need two extensions to the ontology interpretation of the lookup service. First, the peers for the PDF generator and PDF printer services may be instantiated more than once in the system because for larger marketing campaigns multiple generators and printers should work in parallel. The lookup service should then assign the registered PDF generators and printers using a round-robin strategy to achieve a simple load balancing. Secondly, the form-based input requires an assignment to human agents. Thus, the form-based input server keeps track of the clients that are running on desktops and stores the human agents and their organizational role in the lookup service when the human agent logs in. We add two operations to the lookup protocol, allowing clients to select the form-based input peer on basis of the role of the human agent (e.g. to assign a form to any member of the marketing support staff) or to a specific person (e.g. to direct an answer of a customer to the human agent working on the customer's data). We also add a status flag to the form-based input peer, indicating whether the human agent is idle, performing an input, or away from work. The operations select the idle peers first.

Peers representing machine actions like document archiving are typically modeled as PER-REQUEST INSTANCES. In these cases, a RESULT CALLBACK waits for the result of the action because before the document is not archived, we must not proceed with the workflow, and if there are problems, compensation actions must be triggered. The workflow engine peer is modeled as a STATIC INSTANCE because it should survive as long as the engine runs. The peers representing human actions like form inputs are modeled as CLIENT-DEPENDENT INSTANCES because within one workflow the same human agent might be needed for a number of inputs (thus a session abstraction is needed) and, if the human agent does not complete a task in a business process, LEASING is needed, so that the workflow peer can get aware of this problem, and can take some compensation actions.

Figure 9 shows an examples of a possible component configuration for a marketing campaign workflow[2]. The connectors between peers and the federation are dynamically established during registering with the federation, and the connectors between the peers in the federation are dynamically established using a lookup in the federation's lookup service.

Other business units or organizations are modeled by other federations. If a business process needs to interact with another business unit or organization, we need a bridge across federation boundaries. Consider orders can be placed by customers within a marketing workflow, but are handled in a separate order department. To

[2]To model the federation component configuration in UML2 we use the "grouping" architectural primitive, introduced in [Zdun and Avgeriou 2005].

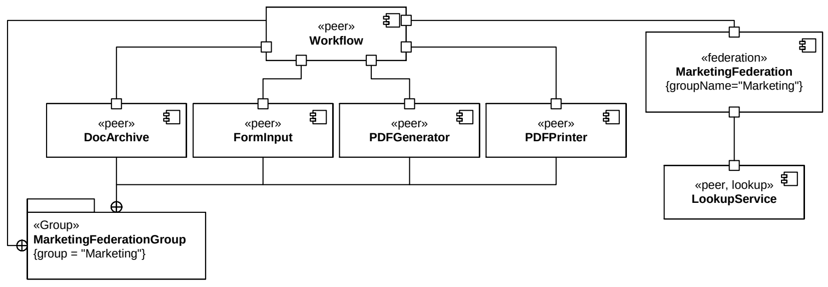

Fig. 9. Example: Component configuration for a marketing campaign workflow

handle this scenario, we instantiate a peer that is member of two federations: the marketing and the order federation. This peer only supports one operation for taking orders, which triggers the order business process. Thus this is the only operation that peers from the marketing federation can invoke in the order federation. The same operation is also exported to a peer representing the Web representation of the company. Thus the order implementation can be reused, but only this specific service of the order federation is exported to other federations. Internally, the order process works similar as the marketing business processes: it looks up the relevant peers in the lookup service and handles orders using the peer services. We model this scenario in UML2 using a peer that is member of the two groups, as shown in Figure 10.

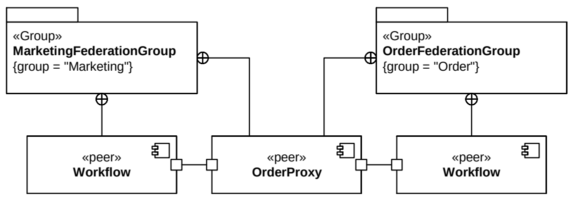

Fig. 10. Example: Component configuration with two federations and a bridging peer

To conclude, in this example, introducing Leela into the various business components required a similar effort as would be needed for other SOA realizations such as Web services frameworks: interfaces and adapters for all business components needed to be written. Only for the workflow engine peer the Leela model is a little simpler because Leela inherently supports peers that are both interface and adapter (server and client) at the same time. When using a Web service framework the integration to the application server would require additional development efforts. The benefit of the Leela architecture is that it inherently supports the federation abstraction, which resembles the organization into business processes. Thus the simple roles in the business units are quickly modeled and the components within one business unit can interoperate freely. Connecting to other business units is simply handled by proxy peers which are member of multiple federations.

The semantic lookup service supports dynamic composition of workflow component configurations, simple load balancing, and roles in business units. Both the composition features and the lookup service can be extended very flexibly, so its easy to cope with unexpected requirements or component configurations. The federation abstraction can also be used to handle security boundaries and concerns. In a Web service framework, in contrast, all these issues would need to be hand-crafted for each individual service.

9. RELATED WORK

Many standard Web services architectures, such as IBM's WSCA [Kreger 2001], Apache Axis [Apache Software Foundation 2004], or Microsoft's .NET [Microsoft 2003], are alternatives for deploying and providing access to business functions over the Web. These have a few standard protocols in common, but the Web services stack architectures are currently still diverse. SOAP [Box et al. 2000] is used as a message exchange protocol, WSDL [Chinnici et al. 2004] is used for interface description, and naming and lookup is supported by UDDI [OASIS 2004]. We have deviated from this approach by using WSDL only as an export format for describing the deployed services. In our approach, the primary source for interface information about peers and their federation is the semantic lookup service. We also used this approach to better support the business application goals, motivated in Section 1, than a centralized UDDI service. To support these goals, we usually require a more simple and more controlled lookup service that can be tailored to the particular problem domain. This is directly supported by the decentralized semantic lookup service concept, proposed in this article.

In the Web service realm, additional composition of services is supported by business process execution languages, such as BPEL4WS [Andrews et al. 2003]. Such languages provide high-level standards for (hierarchical) flows of Web services. Another example is WS-CDL [W3C 2006], which defines interoperable, peer-to-peer collaborations between Web services. In our concept, such approaches can be applied on top of the SOA middleware layer.

Peer-to-peer (P2P) systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric [Rowstron and Druschel 2001]. P2P has gained much attention in recent years, especially P2P systems for personal use, like Gnutella, Napster, SETI@Home, and others. Technically there are still quite diverse views on P2P. For instance, P2P can be interpreted as a variant of the CLIENT-SERVER style [Shaw and Garlan 1996; Bass et al. 1998] 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. Basic functionalities shared by most current P2P systems are that they connect peers for a common purpose, offer a LOOKUP of peer services, and provide a way to exchange data or invoke remote services [Chtcherbina and Voelter 2002]. 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. In our approach we have followed the general P2P architecture to leverage its benefits, but added the federation abstraction to avoid the problems and allow for more central control.

P2P architectures usually support the decentralized lookup of peers via peers. There are a number of research projects extending this basic concept. For instance, Chord [Morris et al. 2001] is a lookup service that uses the P2P paradigm for implementing the lookup service itself. It aims at efficiently determining nodes that store data items in a distributed network. Pastry [Rowstron and Druschel 2001] provides the decentralized, scalable, and self-organizing lookup and application-level routing for P2P systems. Our work also provide decentralized lookup services in different federations, but does not use all peers in the network, but only one lookup service per federation. Using many federations, many decentralized lookup services can be connected in P2P-style, but our approach does not (yet) support standard protocols for exchange or forwarding of lookup service requests.

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. They can get 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 [Arnold et al. 1999] 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 LOOKUP by providing a CLIENT PROXY for their services as well as a set of service attributes. LEASING is used to remove unused services automatically from the network. Clients looking for a service with particular attributes send a request to a lookup service. In response the client receives all those proxies of services matching the requested service attributes. The Home Audio Video Interoperability (HAVI) standard [HAVI 2001] is designed for networking consumer electronics. Especially, self-management and plug & play functionalities are provided for spontaneous networking. Compared to our approach, spontaneous networking environments are similar because they also introduce a model similar to P2P models, but offer more central control. The control features are, however, more focused on self-management and self-adaptation of a network of peers than on composition of loosely-coupled business services, as in our model.

The architecture of our SOA middleware resembles that of a distributed coordination model implementation. 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 [Ciancarini 1996]. As related work for our work, especially coordination of distributed and concurrent systems is of interest. The coordination language Linda [Gelernter et al. 1985] 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 repository 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 [Carriero et al. 1995] 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 [Ciancarini et al. 1998] structures Linda spaces by controlled access using different agents for user representation, interfaces to other agents, administrative functionality, and gateways to other PageSpaces.

10. DISCUSSION AND EVALUATION

Our approach is influenced by concepts known from Web services, P2P systems, coordination technologies, and spontaneous networking, but can also resolve some apparent open issues of these approaches. Unfortunately, many of the goals that we have implicitly introduced in this article are hard to quantify in the context of complex, real middleware platforms. We thus use a qualitative frame of reference to compare the desired features of Leela with the other technologies.

In particular, in Table I we compare "typical" realizations of the following technologies to Leela[3]: Web Services frameworks like Apache Axis, P2P systems like Gnutella, Napster, or SETI@Home, coordination and cooperation technologies like Linda or Pagespace, and spontaneous networks like Jini. In addition, we also compare to the combination of Web services with semantics-based metadata services, such as Triple [Sintek and Decker 2002], Ontobroker [Decker et al. 1999], or Fact [Horrocks 2001], an architecture which is closer to Leela's lookup service design than for instance Web services and UDDI lookup.

Please note that such a comparison is always subjective to a certain extent, and in this comparison we concentrate on the "desired" properties of the Leela framework; i.e. the properties the Leela design has been optimized for. That is, the ratings in Table I should not be understood in the way that our approach is generally superior to the other approaches, but that its central strength lies in application scenarios, where this specific combination of properties is required. In application scenarios, where some of these properties are not needed or even unwanted, most likely other approaches are better suited. For instance, if a central control model is not needed at all, most likely P2P system are better suited because they avoid the overhead implied by the additional control model.

All six technologies provide good support for dynamically constructing invocations, one of the basic requirements for the Leela framework. Even though the realizations of dynamic invocations are slightly different we do not observe highly superior or inferior solutions. Many helpful features supporting dynamic invocations, like the lookup of interfaces and properties, differ more clearly, and are evaluated in the following paragraphs.

In many business scenarios a certain level of control is required. In the basic Web

[3]We use a simple scoring scheme in the figure: ++ very good support, + good support, o some support, - weak support, - - no support.

	Web Services	*P2P Systems*	*Coordination and Cooperation Technologies*	*Spontaneous Networks*	*Web Services + Semantic Web Reasoner*	*Leela*
Dynamic Invocations	++	++	++	++	++	++
Central Control	o/+ Server-based control in basic Web services stack/ More control by higher-level protocols	- -	+/++ Shared Tuplespace/ Pagespace	- -	o/+ Server-based control in basic Web services stack/ More control by higher-level protocols	++ Federation model
Dynamic Deployment	o Static WSDL	++ Spontaneous connections of peers + initial broadcasts or server-lists	o Central “well-known” cooperation space + coordination primitives	++ Spontaneous connections + initial broadcasts	o Static WSDL	++ Spontaneous connections of peers + one “well-known” federation
Simplified Cooperation Model	- - Client-Server	++ Peer-based model	++ Tuplespace-based model	++ Nodes in the spontaneous network	- - Client-Server	++ Centrally managed peer model
Interface Lookup	++	++	- No standard lookup	++	++	++
Lookup of Properties	o/+ UDDI complex/ Simple property lookup	+ Simple property lookup	- No standard lookup	+ Simple property lookup	++ Ontologies for properties	++ Ontologies for properties
Lookup Queries	- Static, client	- Static, client	- Static, client	- Static, client	+ Dynamic, rule-based	++ Dynamic, domain-specific extensions
Service Separation	- One scope/ Client-Server	- One scope/ different processes	-/- - One scope/ shared dataspace	- One scope/ different nodes	- One scope/ Client-Server	++ Thread-specific interpreter per peer

Table I. Qualitative comparison of Leela’s design goals with other technologies

services stack architectures, no specific support for central control is provided, but because Web services follow the CLIENT-SERVER style, servers can at least control their clients. Control of a (larger) set of clients and servers is only supported by additional, higher-level protocols such WS-CDL or BPEL4WS. P2P systems and spontaneous networks have no direct support for central control. Coordination and cooperation technologies support a shared tuplespace, which can be used to realize central control tasks. In the Pagespace variant, different agents can control access and provide a connection to other Pagespaces, like proxy peers in Leela. In Leela, we have provided a simple central 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 control and fine-tune the remote access.

Our deployment model is similarly simple as Web services 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 services frameworks – but they are different to those P2P environments or spontaneous networks that are exploiting broadcasts and similar means. Cooperation and coordination technologies can use the central "well known" tuplespace for deploying service information. The deployment is based on coordination primitives offered by the coordination language.

The most important design goals of the Leela framework was ease-of-use regarding the development, use, and deployment of remote services. An important aspect for comparing ease-of-use is the cooperation model of the various technologies. We can contrast the cooperation models to the model of typical OO-RPC middleware platforms, such as CORBA or DCOM. Web services do not offer a simplified cooperation model, but just the classical CLIENT-SERVER model. P2P systems allow for simpler forms of cooperation following the peer-based model, which is similar to the nodes of a spontaneous network. Tuplespaces also provide a simplified cooperation model, however, they are different as they rely on a shared space for cooperation, whereas the other two technologies rather rely on the spontaneous connection of peers. Leela combines the two styles because it provides a peer-based model that is centrally managed.

Regarding the lookup of services all technologies offer interface lookup services, except for coordination and cooperation technologies, where no standard lookup is provided. But it can easily be provided using the shared tuplespace. The same holds true for simple property lookup, which is also supported by many proprietary lookup services for Web services, P2P systems, and spontaneous networks. The UDDI lookup model is more ambitious in its vision, but it is also viewed as being complex and not exactly matching the requirements in many application scenarios. Both, Semantic Web reasoners and the Leela lookup service support (custom) ontologies for describing service properties. Regarding lookup queries, all technologies except for Leela and Semantic Web reasoners, support only static query models with queries constructed at client side. Semantic Web reasoners allow for the dynamic construction of rule-based queries, but the query model and the query language cannot be as easily extended as in the Leela model, which allows for the dynamic definition of domain-specific query languages.

A specialty of the Leela model is that it provides THREAD-SPECIFIC STORAGES for its peers, and hence supports the explicit notion of service separation and a model for asynchronous, event-based communication between local and remote peers and federations. This feature is not supported by any of the other technologies.

Many underlying parts of our framework can be exchanged for other (OO-)RPC middleware or implemented in other programming languages – a benefit of the pattern-based design. Thus we believe our results are generally applicable. We have verified this claim in a number of student projects. The students had the task to realize a simple middleware implementation following the pattern-based design. When we compare the (subjective) marks for code and design quality, and the completeness of the fulfillment of the tasks, all projects were rather successful: in average the results of the students were always clearly better than the middleware implementations of previous years that were realized without a pattern-based design as a guideline. These student projects give us the hope that the general claim that a pattern-based design can be used for other platforms and technologies holds. However, additional validation work in a more realistic setting than student projects would be needed to finally validate the claim. But this goes beyond the scope of this article.

As argued above, 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. 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. Other security issues are handled by the INTERPRETERS, INVOCATION INTERCEPTORS, and PROTOCOL PLUG-INS.

Another important side aspect is the performance impact of the Leela framework. In contrast to many other properties evaluated above, this aspect can be measured. We compared invocations in different invocation styles in Leela (Version 0.2) to synchronous SOAP invocations using the same SOAP framework (TclSOAP 1.6.7) and Web server (TclHTTP 3.4). In particular, we tested a per-request method invocation, a static method invocation, and a client-dependent method invocation. For all three invocation styles we also tested one error scenario: a per-request invocation with a "cannot dispatch method" error, a static invocation with a "not exported" error, and a client-dependent invocation with a "lease expired" error. Finally, we also compared the average time in an invocation series of a local invocation, a remote invocation, and another remote invocation going backwards.

For all invocations we used the same service, which contains a computation running for 300 milliseconds (which is a typical scenario in our case studies)[4]. For each variant we have tested 1, 3, 10, and 20 invocation in a row. All thread pools were pre-configured with 4 workers. All results are measured in milliseconds. All measurements were performed on Intel P4, 2.53 GHz, 1 GB RAM machines running Fedora Core 4 Linux, and in a 100 Mbit network without further traffic. We have measured all performance tests 10 times and used the best results (the average results were quite close to the best results and therefore we omit them here).

We can see in Figure 11 that there is a slight overhead for the Leela framework, varying with the different invocation variants and sometimes Leela even performs slightly better. This might be due to the use of pre-initialized thread pools in Leela. Even the greatest overhead of Leela is rather low compared to the overall overhead for SOAP and HTTP processing. We have confirmed this interpretation by running the same tests using the Socket PROTOCOL PLUG-INS, which shows more or less the same picture (and hence is omitted here), but with great performance improvements. Hence, we can conclude for most high-level business scenarios (where

[4]Please note that we have also run the tests with a service doing nothing. Because the picture is more or less the same, we do not show these figures here.

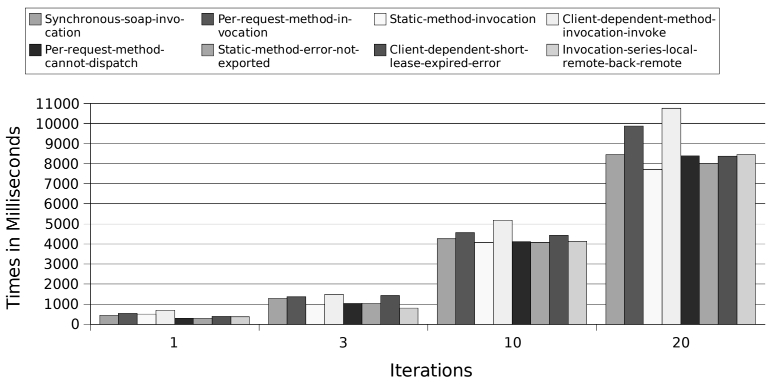

Fig. 11. Performance comparison

SOAP and HTTP communication is used), the overhead of Leela can be neglected.

There are also some liabilities of the current prototype implementation: it does not support structured federations (like hierarchical federations), provides only support for SOAP and Socket plug-ins, and offers no quality of service 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. We also plan to design more high-level services and protocols on top of Leela, for instance, dealing with more automatic and autonomic cooperation, configuration, and lookup of peers.

11. CONCLUSION

Our design and implementation are based on a number of pattern languages (see Section 3). We followed the pattern languages quite closely, but combined the patterns in innovative ways, following the pattern-based SOA reference architecture described in our earlier work [Zdun et al. 2006]. In particular, we have built a small, simple prototype on basis of the essential patterns of the pattern languages first, and then evolved the prototype incrementally. This pattern-based prototyping approach is very useful for researching on new architectures in domains that are already well covered with patterns – a situation that frequently occurs in research projects or when innovative products should be developed. New ideas can be tried out, without re-inventing the wheel for architectural aspects that are already well understood, because these well-understood aspects are described by the patterns. Another advantage of a pattern-based design is its generality. Due to the pattern-based design many framework parts are exchangeable, and at the same time, we can reuse large parts of the design for implementations in other environments.

The result of this research approach is a novel combination of concepts known to work in different technologies – Web services, P2P systems, spontaneous networking systems, coordination technologies, and Semantic Web systems – to a concept for a service-oriented middleware. The pattern-based approach helped us to preserve the benefits of these approaches and avoid some liabilities that we have identified in the area of loosely coupled business services.

REFERENCES

Alexander, C. 1979. *The Timeless Way of Building.* Oxford Univ. Press.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. 2003. Business process execution language for web services version 1.1. http://www-106.ibm.com/developerworks/library/ws-bpel/.

Apache Software Foundation. 2004. Apache Axis. http://ws.apache.org/axis/.

Arnold, K., Wollrath, A., O'Sullivan, B., Scheifler, R., and Wald, J. 1999. *The Jini Specification.* Addison-Wesley.

Avgeriou, P. and Zdun, U. 2005. Architectural patterns revisited – A pattern language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPloP 2005).* Irsee, Germany.

Bass, L., Clements, P., and Kazman, R. 1998. *Software Architecture in Practice.* Addison-Wesley Longman, Reading, MA, USA.

Beckett, D. 2004. Redland RDF application framework. http://www.redland.opensource.ac.uk/.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. 2000. Simple object access protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/.

Brickley, D. and Guha, R. 2004. RDF vocabulary description language 1.0: RDF schema. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-oriented Software Architecture - A System of Patterns.* J. Wiley and Sons Ltd.

Carriero, N., Gelernter, D., and Zuck, L. 1995. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems: Proc. of the ECOOP'94 Workshop on Modles and Languages for Coordination of Parallelism and Distribution*, P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds. Springer, Berlin, Heidelberg, 66–76.

Chinnici, R., Gudgin, M., Moreau, J., Schlimmer, J., and Weerawarana, S. 2004. Web Services Description Language (WSDL) Version 2.0 . http://www.w3.org/TR/wsdl20/.

Chtcherbina, E. and Voelter, M. 2002. P2P Patterns – Results from the EuroPLoP 2002 Focus Group. In *Proceedings of 7th European Conference on Pattern Languages of Programs (EuroPloP 2002).* Universitätsverlag Konstanz, Irsee, Germany, 195–232.

Ciancarini, P. 1996. Coordination models and languages as software integrators. *ACM Computing Surveys 28,* 2, 300–302.

Ciancarini, P., Tolksdorf, R., Vitali, F., Rossi, D., and Knoche, A. 1998. Coordinating Multiagent Applications on the WWW: A Reference Architecture. *IEEE Transactions on Software Engineering 24,* 5, 362–375.

Decker, S., Erdmann, M., Fensel, D., and Studer, R. 1999. Ontobroker: Ontology based access to distributed and semi-structured information. In *Semantic Issues in Multimedia Systems. Proceedings of DS-8*, R. Meersman, Ed. Kluwer Academic Publisher, 351–369.

Dublin Core Metadata Initiative. 2004. Dublin core. http://dublincore.org/.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gelernter, D., Carriero, N., Chandran, S., and Chang, S. 1985. Parallel programming in Linda. In *Proceedings of the 1985 International Conference on Parallel Processing.* 255–263.

Goedicke, M., Neumann, G., and Zdun, U. 2000. Object system layer. In *Proceedings of 5th European Conference on Pattern Languages of Programs (EuroPlop 2000).* Irsee, Germany.

HAVI. 2001. HAVI specification 1.1. http://www.havi.org.

Hohpe, G. and Woolf, B. 2003. *Enterprise Integration Patterns.* Addison-Wesley.

Horrocks, I. 2001. The FaCT System. http://www.cs.man.ac.uk/∼horrocks/FaCT/.

IEEE WG 12. 2004. Learning object metadata. http://ltsc.ieee.org/wg12/.

Kircher, M. and Jain, P. 2004. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management.* J. Wiley and Sons Ltd.

Kreger, H. 2001. Web service conceptual architecture. IBM Whitepaper.

McGuinness, D. and Harmelen, F. 2004. Web ontology language (OWL). http://www.w3.org/TR/2004/REC-owl-features-20040210/.

Microsoft. 2003. .NET framework. http://msdn.microsoft.com/netframework.

Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. 2001. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*. San Diego, CA.

Neumann, G. and Zdun, U. 2000. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*. Austin, Texas, USA, 163–174.

OASIS. 2004. UDDI. http://www.uddi.org/.

Object Management Group. 2004. Common Object Request Broker Architecture (CORBA/IIOP). http://www.omg.org/technology.

Rowstron, A. and Druschel, P. 2001. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. 329–350.

Schmidt, D. and Buschmann, F. 2003. Patterns, frameworks, and middleware: Their synergistic relationships. In *25th International Conference on Software Engineering*. 694–704.

Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. 2000. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd.

Shaw, M. and Clements, P. C. 1997. A field guide to Boxology: Preliminary classification of architectural styles for software systems. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*. IEEE Computer Society, 6–13.

Shaw, M. and Garlan, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley.

Sintek, M. and Decker, S. 2002. TRIPLE–A query, inference, and transformation language for the Semantic Web. In *Proceedings of the First International Semantic Web Conference (ISWC)*. Sardinia.

Voelter, M., Kircher, M., and Zdun, U. 2004. *Remoting Patterns*. Pattern Series. John Wiley and Sons.

W3C. 2004. Resource Description Framework (RDF). http://www.w3.org/RDF/.

W3C. 2006. Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/ws-cdl-10/.

Zdun, U. 2003. Patterns of tracing software structures and dependencies. In *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003)*. Irsee, Germany, 581–616.

Zdun, U. 2004a. Pattern language for the design of aspect languages and aspect composition frameworks. *IEE Proceedings Software 151,* 2 (April), 67–83.

Zdun, U. 2004b. Some patterns of component and language integration. In *Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPLoP 2004)*. Irsee, Germany.

Zdun, U. 2005a. Leela. http://leela.sourceforge.net/.

Zdun, U. 2005b. Semantic lookup in service-oriented architectures. In *Engineering Advanced Web Applications*, M. Matera and S. Comai, Eds. Rinton Press, Princeton, 124–135.

Zdun, U. 2006. Patterns of component and language integration. In *Pattern Languages of Program Design 5*, D. Manolescu, M. Voelter, and J. Noble, Eds. Addison-Wesley, Reading, MA, USA, 357–400.

Zdun, U. and Avgeriou, P. 2005. Modeling architectural patterns using architectural primitives. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2005)*. San Diego, CA, USA, 133–146.

Zdun, U., Hentrich, C., and van der Aalst, W. 2006. A survey of patterns for service-oriented architectures. *International Journal of Internet Protocol Technology 1,* 3, 132–143.