SOA Meets Robots - A Service-Based Software Infrastructure for Remote Laboratories

Peter Tröger[1], Andreas Rasche[2], Frank Feinbube[2] and Robert Wierschke[2]
[1] Blekinge Institute of Technology, Ronneby, Sweden
[2] Hasso Plattner Institute at University of Potsdam, Germany

***Abstract*—With the ongoing internationalization of virtual laboratories, the integration aspect becomes more important. The meanwhile commonly accepted 'glue' for such legacy systems are service oriented architectures, based on standardized and accepted Web service standards.**

We present our concept of the 'experiment as a service', where the idea of service-based architectures is applied to virtual remote laboratories. In our laboratory middleware, experiments are represented as stateful service implementations and jobs as logical service instances of these implementations. We discuss performance, reliability, security and monitoring issues in this approach, and show how the resulting infrastructure - the Distributed Control Lab - is applied in the European VetTrend project.

***Index Terms*—remote laboratory, service-oriented architecture, VetTrend, stateful service instance, monitoring data model**

I. INTRODUCTION

The Distributed Control Lab (DCL) is a virtual laboratory at the Hasso Plattner Institute in Potsdam, which enables the remote usage of experiments for teaching purposes. Authenticated users can submit control programs for an experiment by the help of different front-ends, such as the Web interface, a development environment plug-in or a command-line interface. Each control program by a particular user is called a *job*, which is executed by a matching *experiment controller* that steers the according physical experiment hardware.

The DCL infrastructure is responsible of distributing incoming jobs to available experiments. Multiple experiments of the same type, being able to handle the same kind of job, are called *experiment types*. The infrastructure supports both physical experiments and simulations for the same experiment type. Simulations are intended to help out in case of high load on experiments, e.g. before a student assignment deadline. Simulations can act as full replacement for the real experiment in most cases, since students submit most of their jobs for checking the correctness of their control application. This only demands mainly a compiler run for the control program in the particular runtime environment, but not a real execution of physical activities [4, 6].

Beside the support for teaching activities, research in the DCL project covers the question of protecting the infrastructure against malicious code, which can potentially harm experiment hardware or execution nodes. It utilizes techniques such as automated source code analysis, run-time monitoring and dynamic adaptation for protecting the experiment infrastructure [5].

Within the DCL, several real-time control experiments have already been integrated. Fig. 1 gives an overview of some experiments connected to the DCL. Foucault's Pendulum as an experiment imitates Leon Foucault's famous experiment for measuring the earth rotation. An iron ball is used for the pendulum that can be accelerated using an electro-magnet connection to a control-PC via USB. Two orthogonal laser-based light barriers provide position information about the swinging ball. In this experiment, students have to implement an algorithm which evaluates the light barriers and switches the magnet on and off to keep the pendulum swinging.

A second experiment is the Higher Striker, which works like a linear motor. Seven electro-magnets are placed around a tube of glass and can be used to accelerate an iron cylinder. Light barriers among the tube can be used to determine the position of the cylinder. The task of the experiment is to analyze the data stream sampled from

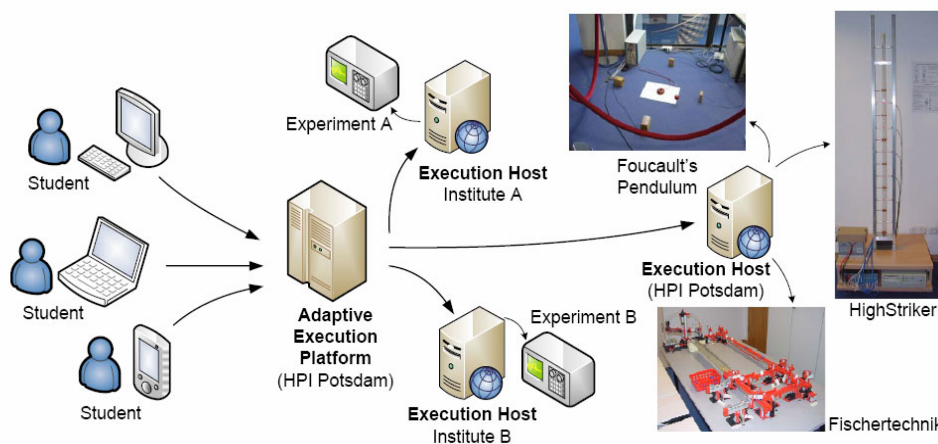

Figure 1: The Distributed Control Lab

the light-barriers and generate a control data-stream for the magnets to move the cylinder to the top of the tube. The electro-magnets and the light barriers are sampled by a control-PC with a frequency of 38,4 KHz. We use this experiment to teach the programming of embedded real-time control applications and students compare different real-time operating systems on the control-PC.

Another experiment shown in Fig. 1 is a model of an assembly line controlled by a programmable logic controller (PLC). In addition to the control program, which has to be implemented in a PLC-language (IEC 61131), monitoring and HMI components are implemented as Java and .NET programs, which be uploaded via our laboratory infrastructure. Students can use this experiment to experience heterogeneous embedded control systems. Further experiments also include the programming of Lego NXT robots and various simulators for our physical experiment installations.

A. *Motivation*

For several years, the DCL installation at HPI was based on a proprietary distributed .NET application. Both the experiment controllers and the job scheduler where realized with the .NET 1.1 framework and its proprietary remoting technology. Meanwhile, new experiment types require the experiment controllers to be implemented in Java and other languages that are either not or badly supported by the .NET remoting environment. This requirement motivated the switch to a service-oriented middleware, in order to integrate different execution platforms for the experiment controllers.

During the utilization of the DCL infrastructure for the European Vet-Trend project, we also faced the new problem of integrating experiment installations from different organizations. Since heterogeneous technologies and execution platforms are in use in the field, a way needed to be found to integrate these systems. This requirement also motivated the usage of service architecture to couple the experiment installations.

Another motivation to improve the existing DCL was the new separation of compile and execution step for single experiment runs. In the old architecture both steps were coupled, causing many users to wait for a compilation, while the physical experiment was in use by a long running job. We wanted to avoid this possible scalability bottleneck by using additional execution resources for compile services in a dynamic fashion.

II. STATEFUL SERVICE CONCEPT

In order to realize the DCL as service-based distributed environment, we applied results from our earlier *Service Infrastructure* research [8] to the domain of remote/virtual laboratories. Our stateful service approach extends the idea of stateless Web services with the explicit notion of *service instances*. Client applications (such as workflow engines in typical SOA environments) are enforced to perform an explicit service instantiation through a factory operation. The factory returns a reference to a *logical service instance*, which is described as *WS-Addressing*-compliant XML document [2]. This document is then used by the client for subsequent service invocations, which are all automatically related to the initially created session between client and server.

A logical service instance represents a stateful entity to the client, but does not necessarily need to be realized by only one *physical service instance* on a particular server. This slightly extends the idea of standard Web service frameworks, where services are referenced by an endpoint URI for a particular service instance on a particular machine. Instead, all clients communicate with a coordination layer that routes SOAP requests (specifically the SOAP body) to a matching execution host. All logical and physical instances relate to their according service implementation, which is realized as binary Web service component, such as a Java Servlet or a .NET Assembly. The kind of implementation for the particular service is transparent to the client in this case, and depends only on the available kind of execution hosts.

In our stateful service concept, a logical service instance has query-able state and monitoring information, expressed by uniformly accessible attributes. Our implementation uses the specifications for the *Web Services Resource Framework (WSRF)* to allow interaction with stateful SOAP implementations. WSRF combines the *WS-ResourceProperties (WS-RP)* specification, which defines read, write and list operations for Web service attributes [1], and the *WS-ResourceLifetime (WS-RL)* specification, which defines operations and WS-RP attributes for managing the lifetime of a service instance [7]. Since both the attribute access and the lifetime management is independent from the particular service implementation, clients can access and utilize this functionalities in all cases. The according query and update operations become automatically part of the service interface, as defined in the according standards.

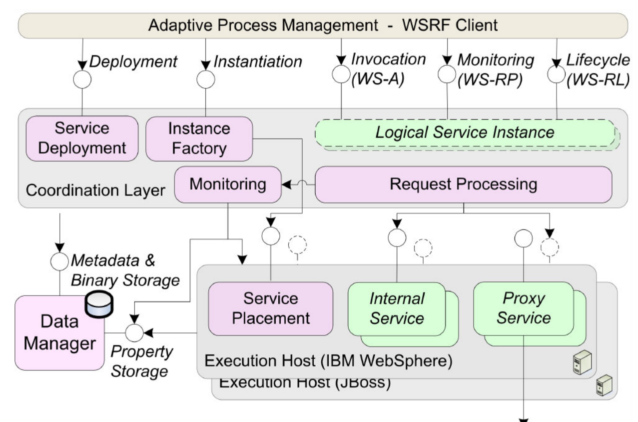

Figure 2: Service Infrastructure

A first implementation of this infrastructure concept was realized and tested in the *Adaptive Services Grid* project [12]. Fig. 2 shows how the logical service instances for the client are managed by a coordination layer, which schedules and manages the incoming requests for the available set of physical service instances. New service implementations can be deployed at runtime, in which case the coordination layer chooses the right execution host for the binary (*service placement*). Service access is monitored by the request processing components, which supports the unified gathering of monitoring information for all service types. Service implementations can use a central storage facility to provide attribute values to the client or to save their own

state between invocations. The atomic services can either implement some functionality by them self or act as proxy for external functionality.

A. *Experiment as a service*

Based on our research results from the ASG project, we identified the tackled problems to be similar in the context of our new DCL infrastructure:

- Due to the usage of standardized SOAP-based protocols, client and infrastructure implementation should be enabled to rely on different implementation platforms.
- Stateful interaction with services should be made available to the client in an interoperable and standardized manner.
- For scalability and maintainability reasons, new service implementations should be deployable during runtime. They should be able to utilize additional execution resources on demand, without effecting active requests and their clients.
- Clients and services should be loosely coupled. Service access and state data access should therefore not relate to a particular execution host.
- Open access to researchers, students and guest users at the same time demands a prioritization of specific requests according to the users' identity.

In order to check the Service Infrastructure concepts against a virtual laboratory application scenario, we compared the old DCL infrastructure concepts with the stateful service concepts of the ASG infrastructure. The resulting mapping is shown in the following table.

TABLE 1: MAPPING OF DCL CONCEPTS ON STATEFUL SERVICES

Distributed Control Lab	Stateful Service Concept
Experiment controller daemon	Execution service implementation
Experiment compiler daemon	Compile service implementation
Experiment simulation daemon	Experiment service implementation
Job for an experiment type	Logical service instances of compiler and execution service
Compiling a job	Operation on a logical instance of the compiler service
Running a job	Operation on a logical instance of the execution service
Job results	Resource properties of the logical instances for the execution service
List of all available experiments	List of all usable execution services
Status of users job	Resource property of logical service instance
Queue per physical experiment	Queue per physical service instance

Each DCL experiment controller, the software component to execute jobs on physical experiment hardware, can be represented by an experiment execution service (or control service) implementation. It provides the necessary interfaces to execute jobs and query job results.

Since most of the experiments expect the source code of a control application as input, we also introduced the notion of a compile service implementation. It is specific to an experiment type and transforms source code to an executable binary, which can be directly passed to an execution service of a given experiment type.

The decoupling of compilation and job execution improves the scalability of single experiment types. Execution services act as proxy for the physical hardware, and can therefore not be duplicated to multiple physical instances on multiple computers. In contrast, the compiler service acts as self-contained functional unit, and can be replicated over multiple execution hosts.

As described in Table 1, every job for an experiment type can be represented by creating a logical instance for an execution service. The mapping between logical instances and physical instances is managed by the coordination layer (see figure 2). Each client therefore can operate its own logical instance (or session) for an experiment. The central request processing module queues the incoming requests for the available physical instances.

The standardized support for service attributes allows a unified representation of job results. Each physical instance can store job results from the experiment run as attribute values. The infrastructure relates such saved attribute values to the logical instance triggering the operation, and stores the value and related logical service instance identifier in a central database. If the client now queries its logical instance for some current attribute value, the coordination layer can provide the latest data made available by the execution service or the compile service. This decouples write and read operations for experiment data, and also decouples clients from particular execution hosts for compilation or experiment services.

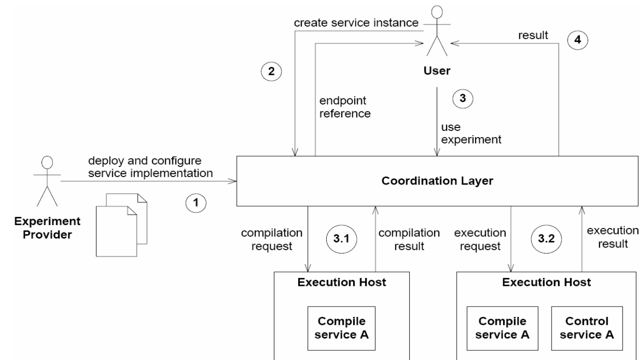

Figure 3: Workflow in the Distributed Control Lab

Figure 3 shows an example workflow for the service-based DCL infrastructure. After the experiment service implementation has been deployed and configured (step 1), the user creates a logical service instance at the Coordination Layer (step 2). In the next step (3) the experiment is executed by invoking the described service operations. The first time an experiment is used, the Coordination Layer dynamically deploys a physical service instance in an appropriate execution host. Afterwards the user code is potentially compiled and finally executed (step 3.1 and 3.2). In the last step the user can acquire experiment results via resource properties of the according logical service instance.

B. *Instance destruction*

A special activity to be considered by the infrastructure is the concurrent usage and destruction of a logical service

instance. The typical case is the cancellation of a long-running experiment job by destroying the according logical instance. With SOAP as basic access protocol, the service consumer will send a destruction request message to the infrastructure, while the response message to the original service request is still pending.

With the destruction request in place, the infrastructure coordination layer now has to decide how the still running request is handled. This is mainly a decision based on the nature of the according service. One typical approach in HTTP-based Web service systems would be the cancellation of the transport layer connection from infrastructure to atomic service, which has the disadvantage of an exception in the service implementation.

In the case of the DCL service infrastructure, physical service implementations therefore describe in their meta-data which kind of cancellation they support. Simulation services support the silent invalidation of the logical instance on a destruction request. In this case, the service call is completed by the physical instance and the returned value is discarded. Execution services demand the explicit retrieval of a cancellation call, in order to cancel the experiment run on the connected hardware. They will therefore not return a response message for the pending call. Since the destruction of logical instances has no influence on the local reference in the service consumer, it might happen that requests are still performed outside of the logical instances lifetime. This results in an error response stating the non-existence of the logical instance.

It must be noted that service implementations shall not be able to influence the life time of their logical instances, in order to keep the strict separation of logical instance coordination and physical execution layer. With such an approach, all life time management remains on the level of logical instances, enabling the flexible assignment of resources.

In the following section, we will now describe the implementation of the updated DCL based on the stateful service concept.

III. IMPLEMENTATION DETAILS

In the current implementation of the DCL infrastructure, new available experiments must be announced by providing an implementation of execution and compile service for a particular experiment type. The experiment provider uploads a *service package* as binary file, containing the service implementation and a *deployment descriptor*. The deployment descriptor contains meta-data such as scheduling configurations, a description of the experiment for the frontend display, properties of the service and the experiment type that is used to group compile and execution services. During the registration process, the WSDL of the services is augmented with the necessary operations defined by the WSRF standards for property and life-time management.

A. Experiment usage

To use an experiment, a logical service instance for both compile and execution service has to be created by the front-end. The created instances allow the usage of an experiment by invoking the standardized `ExecuteExperiment` or `CompileExperiment` method on the logical instance. The DCL coordination layer ensures that a working physical service instance exists for any logical instances being successfully created. If necessary, it places a new service by loading the according service package to a remote host. Results of the experiment runs are centrally stored and can be accessed via the resource properties of the logical service instance. Experiment hardware cannot be shared among multiple jobs. Therefore, the coordination layer has to support the serialization of invocations for physical service instances.

Listing 1 shows a sample implementation of an execution service for the Lego NXT robot experiment. Users can write control programs for robot movement, and submit it to the infrastructure in order to see the resulting physical activities of the experiment hardware. In the implementation, the class `NxtExecuteService` derives from the `WebService` base class, indicating the implementation of a new Web Service. Each execution service must implement the `ExecuteExperiment` method, which receives the program image to be executed as binary array. The method is called by the coordination layer, based on the next request to be handled from the queue of pending logical instance calls.

```
[WebService(Namespace="http://hpi-web.de/esvc")]
public class NxtExecuteService : WebService
{
  [WebMethod]
  public void ExecuteExperiment(byte[]
       experimentProgram)
  {
    WebCam.Record();

    NXTBluetooth.UploadProgram(experimentProgram);
    NXTBluetooth.StartProgram();

    // save experiment result
    PropertySupport.SetInstanceProperty
      ("Video", WebCam.GetVideo());
    PropertySupport.SetInstanceProperty
      ("DisplayOutput", NXTBluetooth.GetLog());
  }
}
```

Listing 1: Experiment implementation

As first step in the example implementation, a camera recording is started to save a video of the robot's movements during the job execution. Then the binary program image is transferred to the NXT via a Bluetooth connection. After the execution of the job, which is indicated over the Bluetooth connection, results of the experiments are saved in the infrastructure. The DCL implementation automatically determines the related logical instance and can therefore provide a generic attribute access library for experiment services. This simplifies the programming model, since the integrators of new experiments don't need to consider the logical instance handling of the coordination layer.

Listing 2 shows a shortened example for the implementation of a client using an experiment and fetching the results afterwards. In the listing, logical instances for compile and execution service are created first. Each service instance is represented by an endpoint reference object, in accordance to the WSRF specification. The `ExecuteExperiment` method returns after the finalization of the control program, or after the maximum time allowed for execution has been expired. The result of

the experiment run can be acquired by accessing the according service attributes with the WS-RP operations.

All DCL experiments are currently accessible over the Internet. Therefore it must be ensured that only authorized users can access the experiments. The DCL therefore requires authentication data to be present in a SOAP request as described by the *WS-Security Username Token Profile* [3]. Furthermore, clients and experiment developers are free to use other standardized mechanisms to protect the message body.

```
// retrieve service instances
EndpointReference compileService =
   serviceFactory.CreateServiceInstance(
        CompileServiceId);
EndpointReference executeService =
   serviceFactory.CreateServiceInstance(
        ExecuteServiceId);

// compile code and execute experiment
byte[] compilationResult =
   compileService.CompileExperiment(code);
executeService.ExecuteExperiment(compilationResult)
      ;

// retrieve experiment result
byte[] experimentResult =
   executeService.GetResourceProperty("Result");
```

Listing 2: Experiment client

B. *Scheduling*

In order to schedule Web service requests in our infrastructure according to an assigned priority, two problems needed to be solved. First, the priority decision value that is encoded in the SOAP message needs to be accessed. As most Web service stacks abstract from the communication handling, the access to priorities is usually not possible before the request processing starts. This prevents a re-ordering of incoming requests according to some priority setting. The problem was solved by intercepting the SOAP processing in the Web service stack, and analyzing the incoming raw XML data in a custom preprocessing handler. This step also covers the reaction on WSRF-compliant request messages, for example for the attribute access, which is not covered by the service implementation itself. The solution provides fast access to the priority values and allows the correct routing of the result messages.

Some of the experiment hardware requires a recovery phase between successive jobs. This is implemented by according queuing strategies in the coordination layer implementation.

Using our central scheduling approach, we are able to tolerate crash-faults, by having execution hosts installed on redundant computers. Before executing a job, the coordination layer checks whether the chosen physical service instance is still operational. If this is not the case, an existing physical service instance on another host is used. If no more physical instances are available, the coordination layer can also decide to deploy the service implementations to another empty machine. The infrastructure supports the addition of new execution hosts at runtime, which allows the immediate reaction on failures without down time for the whole infrastructure.

In a future step, we plan to delegate jobs to multiple physical service instances in parallel and choose a result according to a voting mechanism. This mechanism is independent from the location of the execution host, and can therefore support fail-over scenarios between multiple interconnected virtual labs. For the sake of extensibility, the coordination layer itself is not aware of the deployment format of a service package. At the moment, our infrastructure contains two different types of service containers – one type to process .NET Web services, and one type for JAX-WS Web services. Since all incoming requests contain the information about the logical service instance, successive jobs need not to be processed by the same physical instance. This supports both load balancing and fault tolerance for a particular experiment type, under the assumption of reliable central data storage.

C. *Performance data model*

In order to rely the dynamic resource usage mechanisms on according runtime information, we developed a generic data model for performance measurements in service infrastructures. The obtained data is used to identify performance bottlenecks with simulation and compilation services in the processing of user requests. It also supports the dynamic scheduling of requests to different physical instances.

Our data model is a combination of existing specifications from different standards, especially [9-11]. It focuses only on properties measurable on the level of the service infrastructure itself. Other typical performance counters from hardware and operating systems, such as CPU load or process working set, are not comparable between heterogeneous execution hosts. Therefore, they cannot be utilized for an overall ranking and analysis of services in the infrastructure and were intentionally omitted.

The model distinguishes between performance values *per request*, *per logical instance* and *per service implementation*. Every logical instance allows the retrieval of performance values of all these classes through the WS-RP operations (see section II). The validity scope is expressed by the namespace of the `QName` (e.g. `{'http://dcl/perf/callscope','state'}`).

One example are implementation-scope values, which are retrievable through all logical instances of this particular implementation.

Performance values that are valid *per request* are either measured in the coordination layer or in the runtime environment for the physical instances:

- In the coordination layer
  - Request / response retrieval time
  - Request / response forwarding time
  - Response time (duration)
  - Request status
    - *Received*
    - *Processed*
    - *Finished*
    - *Failed*
  - Request / response size
- In the service container
  - Request processing start / end
  - Utilized CPU time

A single request has the status ‚received' if the coordination layer received the request message completely, but did not forwarded the request to a physical instance so far. The request is then in the status 'processed' as long as the physical instance did not send a response message. The state 'finished' expresses the fact that the processing is completed. Requests with the status 'failed' can occur if the service infrastructure had no physical instance available, or if the response message was identified as SOAP fault message.

The state change to 'finished' or 'failed' is triggered by a generic SOAP interceptor in the service container, which also reports the container-side performance values. The state therefore does not imply any information about the possible availability of experiment results, since this is an implementation-specific issue.

The request processing time, as well as the utilized CPU time is relevant for the dynamic usage of physical instances of the same implementation on different machines. The current implementation of our concept obtains these values also in the SOAP interceptor.

In the second class of monitoring parameters, all values are available *per logical instance*. As a specific characteristic, these values are always also available *per implementation*, basically as an aggregated version. All the values collected in the coordination layer are:

- Number of successful requests
- Number of failed requests
- Successability rate
- Request throughput
- Average / maximum response time
- Maximum request / response size
- Processing time
- Life start / end

The successability rate expresses the relation between the number of successfully processed requests (status 'finished') and the overall number of requests, similar to [10].

It must be noted that the concept of accessibility from this standard ([10]) is not used, since it would demand some message retrieval acknowledgements from the service consumer. This shows again the focus on technology-independent performance metrics in our model.

For the computation of the request throughput, we divide the number of all successful requests by the life time of the instance / implementation. The average response time per implementation is obtained accordingly.

The monitoring values in the third class are only available *per implementation*:

- Status
  - *Available*
    - *Busy*
    - *Free*
  - *Not available*
    - *Stopped*
    - *Failed*
- Overall duration of *available* status (up time)
- Overall duration of *failed* status (outage time)
- Overall duration of *not available* status (down time)
- Availability
- Reliability
- Number of physical / logical instances

A 'busy' implementation is in general available by its logical instances, but has some pending requests at the time of querying. A 'free' implementation has no pending requests. This distinguishing can be used for choosing between different implementations before logical instance creation, and is based on the assumption that state changes are comparatively infrequent.

The non-availability of an implementation, even though the logical instances are still provided, can be transient (reconfiguration, failures) or permanent (de-installation). In both cases, logical instances remain accessible on the coordination layer, to provide an endpoint for late result or performance data retrieval.

Up time, outage time and down time are updated on every status change of the implementation. The availability value describes the portion of time where the implementation was 'available'. The reliability value is computed according to [11], based on the mean time between failures for the implementation.

The overall monitoring model was implemented as part of the updated DCL infrastructure. SOAP interceptors on the execution hosts report the current values to the coordination layer by asynchronous messaging protocols. The coordination layer aggregates the data sets and offers them of the logical instances to interested clients.

IV. OPERATIONAL EXPERIENCES

Within the VET-Trend project, we started a first pilot effort for testing experiment integration with the Technical University Darmstadt and at the Hasso Plattner Institute. TU Darmstadt is operating a remote laboratory for reconfigurable hardware modules, which can be programmed and tested by according tools. In order to perform the integration, TU Darmstadt provides an experiment and compilation service interface for their experiments, which is called by our infrastructure.

Practical tests showed that the usage of SOAP messaging to query information about experiment runs is the most time consuming task in the new infrastructure. Since most of the job-related information remains constant during their life-time, we integrated several caches in parts of the infrastructure. With this technique, the number of fully processed service invocations at the coordination layer was dramatically reduced.

From a technological perspective, we successfully interconnected an ASP.NET frontend with the coordination layer written in Java 6. Current execution host are programmed both in Java and .NET, and initial experiments already showed the possibility also for other platforms. In general, the usage of mature Web service standards and the consideration of WS-I regulations has shown to be helpful in order to achieve true interoperability in a heterogeneous middleware environment.

V. RELATED WORK

Many universities around the world provide virtual and remote laboratories with a variety of experiments. Nevertheless only a few of these labs rely on Web service communication between users and experiments. In contrast to our approach, which uses an extended Web service infrastructure with integrated support for stateful service, load balancing and fault tolerance these features cannot be found in most other approaches.

The MIT iLab project [14][15] provides an open source framework with common functionality for the operation of virtual/remote laboratories. iLab relies on a three-tier Web architecture including client applications, intermediate service brokers and lab servers. Users do not communicate with experiments directly, but use a Web service interface provided by service brokers. A service broker is a generic actor provided by iLab, which synchronizes access to lab servers and also handles authorization and authentication of users. The service broker forwards experiments usages from users with a computed trust level to the lab servers, which are totally decoupled from user management and synchronization details. iLab uses Web services to interconnect experiments residing in on different campuses.

IsiLab [13] is a web-based remote laboratory for measurement experiments in electronics. Behind a portal tier, which generates web pages for users, a workflow manager coordinates Web service invocations within the engine tier and into the resource tier. In the resource tier, measurement instruments can be accessed via Web services. Their usage is synchronized by an additional instrument reservation Web service. An experiment workflow ensures that all necessary resources are reserved during an experiment execution. Together with the workflow, the WS-execution engine service manages user working sessions. IsiLab has much in common with our approach and besides iLab it is one of the most advanced projects using Web services for remote labs. IsiLab currently is restricted to a web page. Our approach has advantages in the flexibility for end-users. It offers a very convenient way to access experiments directly via the offered stateful services. This allows for an easy integration of experiment access into standard development tools.

VI. CONCLUSION

The utilization of service-oriented software architectures for remote/virtual laboratories is a promising approach for solving the typical problems of cross-organizational access, scalable behavior and dynamic resource usage. We presented our concept of an ’experiment as a service’, were physical service instances provide access to the experiment hardware and logical service instances represent according user jobs. The application of mature Web service technologies allows establishing a transnational virtual laboratory environment, which integrates experiments and users from different sites all over Europe. First steps toward such an infrastructure already have been taken.

We have successfully used our laboratory infrastructure in courses on embedded systems, held at the Hasso Plattner Institute and the Blekinge Institute of Technology in Sweden. The integration of new experiments from other organizations has just started.

Future work we will concentrate on the integration of further experiments and on the improvement of our service-oriented laboratory middleware according to user and integrator feedback. Beside the batch mode programming of hardware modules, we also identified the need for an interactive mode, which is required to perform test cycles at the downloaded hardware configuration. The interactive mode will be realized as stream-based interaction with an experiment during the execution of a job.

REFERENCES

[1] S. Graham and J. Treadwell. Web Services Resource Properties 1.2 (WS-ResourceProperties). OASIS Open, June 2004.

[2] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing 1.0 - Core. World Wide Web Consortium (W3C), May 2006.

[3] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web Services Security UsernameToken Profile 1.1. OASIS Open, Feb. 2006.

[4] A. Rasche and A. Polze. Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. In International Symposium on Object-oriented Real-time distributed Computing (ISORC), pages 164–171, May 2003.

[5] A. Rasche, M. Puhlmann, and A. Polze. Heterogeneous Adaptive Component-Based Applications with Adaptive.Net. In International Symposium on Object-oriented Real-time distributed Computing (ISORC), pages 418–425, 2005.

[6] A. Rasche, P. Tröger, M. Dirska, and A. Polze. Foucault’s Pendulum in the Distributed Control Lab. In Proceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems, pages 299–306, Oct. 2003.

[7] L. Srinivasan and T. Banks. Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). OASIS Open, June 2004.

[8] P. Tröger, H. Meyer, I. Melzer, and M. Flehmig. Dynamic Provisioning and Monitoring of Stateful Services. In Proceedings of the 3rd International Conference on Web Information Systems and Technologies (WEBIST 2007), pages 434–438, Mar. 2007.

[9] I. Sedukhin. Web Services Distributed Management: Management of Web Services (WSDM-MOWS) 1.0. OASIS Open. Mar. 2005

[10] E. Kim and Y. Lee. Quality Model for Web Services. OASIS Open. September 2005

[11] J. C. Laprie. Dependability. Basic Concepts and Terminology. Springer Verlag, 1998. – ISBN 978-3211822968

[12] D. Kuropka, P. Tröger, S. Staab and M. Weske. Semantic Service Provisioning. Springer Verlag, 2008 – ISBN 978-3-540-78616-0

[13] G. Donzellini and D. Ponta. The electronic laboratory: traditional, simulated or remote? In Advances on remote laboratories and e-learning experiences. University of Deusto, 2007 - ISBN 978-84-9830-077-2

[14] J. Harward, T. T. Mao, and I. Jabbour. iLab. Interactive Services - Overview. http://icampus.mit.edu/iLabs/Architecture, 2006.

[15] The iLab Project. The Challenge of Building Internet Accessible Labs. http://icampus.mit.edu/iLabs/Architecture, 2004.

AUTHORS

Peter Tröger is with the Blekinge Institute of Technology, Department of Systems and Software Engineering (APS), PO Box 520, SE-37225 Ronneby, Sweden (e-mail: peter.troger@bth.se).

Andreas Rasche, Frank Feinbube, and Robert Wierschke are with the Hasso Plattner Institute, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany (e-mail: [andreas.rasche|frank.feinbube|robert.wierschke]@hpi.uni-potsdam.de).

Manuscript received April 2008. Published as submitted by the authors.