CompTalks – From a Meta-model Towards a Framework for Application-Level Interaction Protocols

Eryk Ciepiela, Maciej Malawski, and Marian Bubak

Academic Computer Centre CYFRONET AGH, Nawojki 11, 30-950 Kraków, Poland
Institute of Computer Science AGH, Mickiewicza 30, 30-059 Kraków, Poland
eryk.ciepiela@cyfronet.pl

Abstract. This work presents CompTalks – a novel concept and meta-model for specifying application-level communication protocols. The goal is to enable custom fine-grained and elaborate message exchange between distributed yet tightly-coupled parties. Hence, the concept of a conversation protocol is introduced. Its reference implementation – the CompTalks Framework – is a Java-based middleware toolkit that supports development, testing, analysis, validation and running highly interactive services. An important feature is the ability to verify the developed protocols at compile time by using a Petri Net-based analyzer. The framework was successfully applied to develop a protocol for GSEngine which serves as the runtime system of the ViroLab virtual laboratory, enabling development and execution of complex collaborative applications.

Keywords: Service-Oriented Architecture, interaction, conversation, application-level protocols, application framework, Java, Petri Net.

1 Introduction

The subject of the research discussed in this paper is focused on paradigms and implementation aspects of stateful and highly interactive services in Service-Oriented Architectures. Our work was motivated by the need for development of a new protocol between the GSEngine [1] client and server which constitute the core of the runtime system of the ViroLab virtual laboratory [2, 3]. The requirements included interactive and collaborative execution of complex applications (experiments) on the server together with online streaming of input and output data together with user interaction message exchange with the client. The rationale for our research is that a whole class of stateful and interactive services is not adequately addressed by the state-of-the-art paradigms and technologies applied in modern service-oriented distributed systems.

Currently available service models and frameworks assume the services are loosely coupled with clients and that their interfaces comprise a set of advisably idempotent operations which are to be invoked in a blocking and synchronous manner. Message-Oriented Middleware (MOM) enables asynchronous messaging, however it only offers predefined styles of messaging (e.g. queues and publish-and-subscribe topics)

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 278–288, 2010.
© Springer-Verlag Berlin Heidelberg 2010

and is therefore intended for loosely-coupled parties. In both approaches consecutive invocations of operations or message passing proceed in no implicit session context, which leaves system designers with few options to explicitly preserve such context. As a result, session tracking code often becomes mixed with service business logic code, preventing it from being system architecture-agnostic and thus reducing its reusability and maintainability.

Current approaches do not suffice for use cases that need finer-grained interaction when:

- Input data is provided to the service in parts – e.g. in a continuous way, in batches, periodically etc.
- The service asks its client to make decisions that determine further processing – e.g. the service performs some steps, but there are check points when it has to ask a decision-maker for further processing instructions, validation, confirmation etc.
- Input data is heavyweight and is the subject of a chain of service operation invocations which are not known beforehand – e.g. input data is sent once, before the client decides which processing chain to apply to this data.
- The service has to ensure richer client experience and responsiveness – e.g. inform about processing status, provide the client with intermediary and partial results and other relevant information from the client's point of view.

In order to address these issues, we propose a new concept of a Conversation Protocol as a way for modelling interaction between services and their clients that is conceptually simplistic, yet generic and scalable enough to be capable of describing complex stateful interactions schemes. Its goal is to natively support session context preservation, asynchronous communication, highly interactive data and control flows. The proposed Conversation Protocol Meta-Model further allows for formal modelling of conversation protocols, thus enabling analysis, simulation and validation e.g. by using the Petri Nets [15] model. The above mentioned concepts are implemented in our Java-based CompTalks Framework which serves as a basis for the GSEngine protocol.

The paper is organized as follows: Section 2 discusses the state-of-the-art architectures and technologies. The Conversation Protocol concept is introduced in Section 3 with further details including formal modelling of conversation protocols enabling analysis, simulation and validation that use the Petri Nets model (Section 4). Section 5 the presents Java-based CompTalks Framework which is a reference implementation supporting Conversation Protocols. Performance tests are reported in Section 6. We conclude with the evaluation of the advantages and limitations of the proposed solution and an outline of the future prospects for CompTalks in Section 7.

2 State of the Art

The Web Services [4] approach considers a service interface as a set of operations which typically are to be invoked sequentially in a blocking, synchronous and asymmetric request-response manner. Another shortage of the Web Services Description Language (WSDL) [5] is that it does not cover the intended sequence in which operations need to be invoked in order to carry out given use case scenarios.

Such directions are not formalized, forcing developers to refer to additional, usually plaintext, documentation, which may lead to improper use of services.

The Web Services Conversation Language (WSCL) [6] proposes a specification for describing the flow of documents exchanged between a Web Service and its client. Such descriptions are intended to be interpreted by Web Service infrastructures and development tools. Although the specification enables describing conversation protocols, it is dedicated to loosely-coupled parties interacting via document exchange. Hence, it is focused on building another abstraction layer over services communicating with SOAP-like [7] protocols rather than on communication protocols allowing for finer-grained interaction and tighter coupling of parties.

Some Web Service-oriented approaches support operations on the state associated with a service. In such cases, the result of an operation depends on prior operations. Web Services Resources (WS-R) [8], Web Services Transfer (WS-T) [9] and Representational State Transfer (REST) [10] follow similar paradigms [11] for managing state and offer advanced access mechanisms, e.g. notifications about state changes or accessing the state in parts. Such an approach can be regarded as state model-centric and document-based; therefore it is poorly suited for interaction-oriented applications, which require a means for specifying custom interaction schemes that reach beyond those offered by the aforementioned specifications.

The Extensible Messaging and Presence Protocol (XMPP) [12] is an XML-based protocol for near-real-time messaging, presence, and request-response services [13]. Contrary to WSCL it enables finer-grained interaction and bidirectional stream-like communication. It originally served as a streaming medium for Instant Messaging, the message exchange cannot be managed by any protocol rules thus cannot support custom application-level conversation protocols.

The Blocks Extensible Exchange Protocol (BEEP) [14] is an attempt to provide a generic application-level meta-protocol for connection-oriented, asynchronous interactions. It enables defining various styles of message exchange (request-response, asynchronous calls and streaming) but keeps it asymmetric with only the server being capable of streaming and generating responses and the client limited to sending requests.

In order to enable interactive connectivity to grid infrastructure nodes several utilities like *glogin* [21] emerged that patch grid toolkits in order to maintain direct client-service bidirectional data streaming channel. This constitutes powerful foundation for interactive message exchange but still needs high-level model for specifying application-level protocols.

3 CompTalks Conversation Protocol Concept

The CompTalks Conversation Protocol proposes a novel and more general way for defining and describing service interfaces. In CompTalks, an interface to a service is regarded as an interaction protocol that specifies messages and rules of message passing between a service and its clients. More precisely, the description of an interface consists not only of a set of messages the service can exchange, but also contains information about the allowed sequences of messages sent between a service and its client. Allowed sequences are specified through a state machine with

transitions denoting messages sent to or from a service. Interactions between endpoints in CompTalks reach beyond the asymmetric client-server request-response model by supporting stateful, asynchronous, non-blocking and symmetric communication in order to enable elaborate, interactive and finer-grained message exchange schemes.

In CompTalks a *conversation* is defined as a message exchange that follows some rules that are agreed a priori by communication endpoints. We distinguish *basic conversation rules* that constitute a rudimentary contract between communicating endpoints, guaranteeing effectiveness, predictability and determinism of conversation. Over that, application-specific rules, expressed in the form of a *conversation protocol,* define allowed messages and message sequencing.

Basic CompTalks conversation rules are codified as follows:

1. They assume a *medium* for carrying messages comprises two streams – one for each direction between the client and the server. Messages are delivered in the order in which they were inserted into the stream. Latency and jitter in message delivery is allowed. Depending on use case reliability has or has not to be ensured. These requirements (including reliability) are met e.g. by the Transmission Control Protocol (TCP) – therefore, the Internet TCP/IP protocol stack can be successfully used as a medium.
2. On each side of a medium there is exactly one *endpoint*. Each endpoint specifies the *messages* it can receive. A message, in turn, specifies in which *conversation state* it can be sent, which conversation state ensues after it is sent and whether it hands over *control* over the conversation. Control denotes whether the endpoint, after receiving such a message, is allowed to send subsequent messages or should expect another message. Therefore, the conversation defines a *conversation state diagram* constructed by nodes, denoting conversation states, and edges, representing messages sent between endpoints. The role of messages is to transfer business-logic data as well as to carry conversation state transitions between endpoints, hence synchronizing conversation state in endpoints. A message can be sent only if the sending endpoint has control over the conversation and the conversation state diagram allows for sending such a message in a given state.
3. The conversation state diagram has to be deterministic; that is, there should be only one endpoint which has control over the conversation in a given state, no matter which transition sequence has led to this state.
4. Endpoints can pass messages to each other in an asynchronous and non-blocking way: the sending operation returns once the message is successfully committed to the medium. Messages delivered by the medium are placed in the buffer queue and processed sequentially by the recipient endpoint in the same dedicated thread. Therefore, the conversation is driven by a pair of *conversation threads*, one per each endpoint.
5. Conversation thread being a mediator in message delivery is the entity that takes care of timing and/or reliability aspects, i.e., it can decide whether to suspend the endpoint processing by holding up sending operation return or whether to postpone or give up sending messages according to some timing policy (e.g. jitter compensation policy).

6. The conversation is initiated by starting both endpoints' conversation threads. Both endpoints are initially in the *init* state. The client's conversation thread starts the exchange by sending the first message to the server.
7. The conversation stops as soon as it reaches the *final* state and all messages are processed by conversation threads.
8. Messages can also set up *sub-conversations* whose life-cycle is contained in the scope of a single state of the *parent conversation*. Sub-conversations can be created by an endpoint only if it currently has control over the conversation. A *sub-conversation handle* can be passed to a remote endpoint along with a message. The recipient endpoint can use this handle in order to start a sub-conversation. The sub-conversations allow for modularization and decomposition of the complex conversation schemes into fine-grained, easily maintainable parts and consequently enables tree-like scaling to larger and more complex conversation patterns.

4 Conversation Protocol Meta-model

An important feature of CompTalks is that it proposes a way of modelling conversation protocols. Once modelled, a protocol can be subjected to analysis, simulation and validation.

The conversation Protocol Meta-Model defines a *Conversation* that comprises *Client* and *Server Endpoints*. Each endpoint specifies a set of *Messages* it accepts. Each message, in turn, is defined by its *Signature*, *Required State*, *Implied State* and *Control Passing Flag*. This can be formalized using a UML class diagram, as presented in Fig. 1.

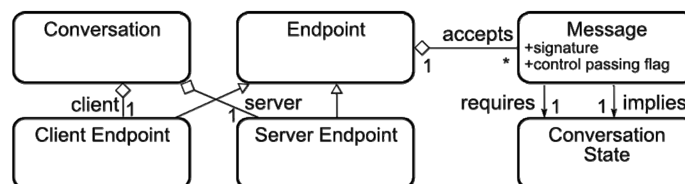

Fig. 1. Conversation Protocol Meta-Model expressed as a UML class diagram

For example let us examine a simple conversation depicted as a UML object diagram in Fig. 2 (a). The conversation starts when the client sends the *foo* message to the server. The *foo* message conducts *text* data of type *String* and causes transition from *init* to *intermediate* conversation state. It also hands over the control over the conversation to the server. After successfully committing the *foo* message to the medium the client enters the *intermediate* state and loses control over the conversation. After receiving the message, the server takes control over the conversation and enters the *intermediate* state where it processes the received data. Having control and being in the *intermediate* state, the server is able to send the *bar* message, which carries *text* data of type *String*, causing transition from the *intermediate* to the *final* state and handing control over to the client. Once the server successfully commits the *bar* message to the medium, it enters the *final* state and hence the conversation thread on the server side ends. Once the client receives the message, it similarly enters the *final*

state and processes the received data. After the data is processed, the conversation thread on the client side ends. A sequence diagram describing such a course of this sample conversation is shown in Fig. 2 (b).

Owing to its intrinsic scalability, CompTalks allows for creating more robust and complex conversation protocols. It supports conversation state diagrams of unbounded finite sizes and provides mechanisms for nesting an unbounded finite number of sub-conversation within the scope of a parent conversation. The conversations are then organized in a tree-like structure, where conversation lifecycle of child conversation is contained in the scope of one of the states of the parent conversation, while sibling conversations' lifecycles are independent and proceed in parallel.

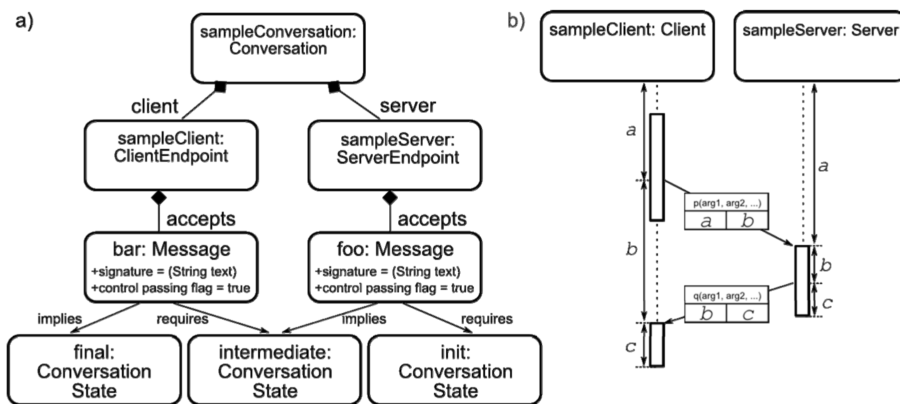

Fig. 2. Sample conversation model that conforms to the Conversation Protocol Meta-Model expressed as a UML object diagram (a) along with a sequence diagram of the course of this conversation (b)

4.1 Methods for Analysis and Validation of Conversation Protocol Models

The Conversation Protocol, modelled in the aforementioned way, can be subject to analysis, simulation and, eventually, validation in terms of conformance to CompTalks conversation rules. This section shows how to transform the conversation protocol into a powerful and thoroughly explored Petri Net [15] model which enables studying static and dynamic properties of asynchronous and concurrent systems.

Transformation from the Conversation Protocol Model to the Petri Net model involves tree steps:

1. Transforming each message specification into a subnet of places and transitions according to the following rules:
    a. Message *m* accepted by the server, with conversation transition from state *p* to *q* and with control passing, results in the subnet depicted in Fig. 3 (a)
    b. Message *m* accepted by the server, with conversation transition from state *p* to *q* and without control passing, results in the subnet depicted in Fig. 3 (b)
    c. Message *m* accepted by the client, with conversation transition from state *p* to *q* and with control passing, results in the subnet depicted in Fig. 3 (c)

d. Message *m* accepted by the client, with conversation transition from state *p* to *q* and without control passing, results in the subnet depicted in Fig. 3 (d)

2. Merging all subnets obtained in step 1.
3. Setting initial marking by placing tokens in *client state initial*, *server state initial* and *client control* places.

According to the transformation procedure, for the conversation composed of *m* messages and *s* states the resulting Petri Net is consisting of *2*s+m+2* places and *2*m* transitions, therefore the size and complexity of a net is merely linearly correlated with a number of states and messages of a conversation. The sample protocol described in the previous section can be represented as a Petri Net, as shown in Fig. 3 (e).

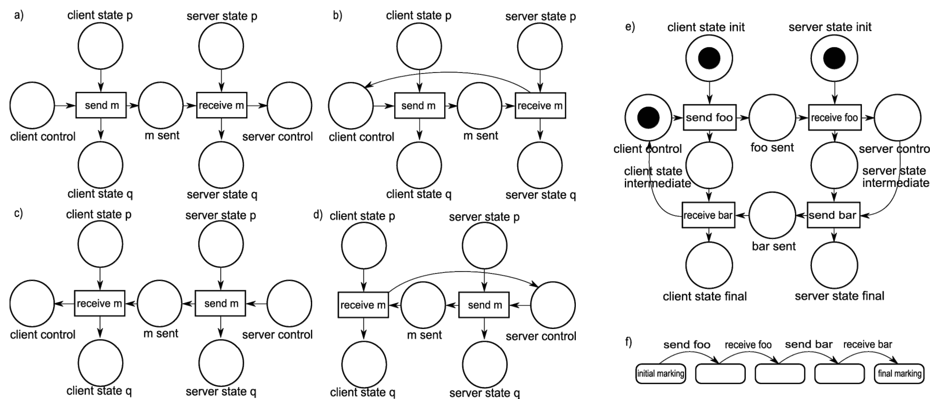

Fig. 3. Conversation Protocol Model to Petri Nets model transformation rules (a-d); sample protocol model transformed to the Petri Nets model (e) and its marking graph (f)

For the obtained Petri Net, a marking graph can be computed and subsequently analyzed. The conversation is considered valid if and only if all the following constraints are met by the marking graph:

1. Each *send X* transition is followed only by *receive X* transition.
2. Each *receive* transition is followed only by zero or more *send* transitions.
3. The only dead marking is the *final marking* with tokens placed in *client state final*, *server state final* and *client control* places.
4. The *final marking* is reachable from each marking that is reachable from the *initial marking*.

As the marking graph for the sample protocol shown in Fig. 3 (f) meets all above-listed requirements the conversation is considered valid.

Such analysis and validation on the model level is useful in ensuring correctness, as it takes place on an early stage of design and at a high level of abstraction. The presented method is abstract and can be implemented using notations specific to a particular framework, e.g. the reference implementation of the CompTalks Framework presented below.

5 CompTalks Framework

The CompTalks Framework is a Java-based reference implementation of CompTalks. The conversation protocols are specified by a pair of Java interfaces which extend *ServerEndpoint* and a *ClientEndpoint* interfaces respectively and specify methods responsible for message interception. Such methods are to be annotated with a *Transition* annotation, which, in turn, specifies the required state of a message (*from* field), its implied state (*to* field) and the control passing flag (*control* field). As an illustration, the CompTalks specification of the sample conversation protocol from Fig. 2 is shown in Fig. 4.

```
public interface SampleServer extends ServerEndpoint<SampleClient> {
   @Transition(from = Transition.INITIAL_STATE, to = "intermediate",
   control = true)
   public void foo(String text);
}
public interface SampleClient extends ClientEndpoint<SampleServer> {
   @Transition(from = "intermediate", to = Transition.FINAL_STATE,
   control = true)
   public void bar(String text);
}
```

Fig. 4. *SampleServer* and *SampleClient* interfaces (irrelevant code fragments omitted)

Applied annotation approach allows for keeping the specification of the protocol in a single source code entity, namely Java interface, and in a single binary entity of Java class file what keeps source code self-documenting on the one hand, and binary file self-contained and easily distributable on the other.

A pair of interfaces, being a specification of a conversation protocol, can be subject to conversation protocol model analysis and validation. The CompTalks Framework offers an analyzer, that, given a pair of compiled interfaces and using the Java Reflection API, determines whether the model is valid according to the Petri Net method presented in the previous section.

Endpoint interfaces' definitions should be provided to the client and the server sides and realized by concrete endpoint implementations, such as the ones presented in Fig. 5. Implementation can be examined in terms of conforming to a given model, e.g., whether implementation ensures that messages are sent only in allowed conversation states etc. This kind of validation may be performed by applying bytecode analysis techniques to the code of implementation classes. Such two-level validation (model validation followed by implementation validation) allows for rapid and early evaluation of software correctness. Moreover, as validation takes place at compile time, it greatly tightens the development-test-feedback loop and eliminates performance-consuming correctness checking at runtime.

The CompTalks Framework is currently available for use and offers support for two types for message passing media: TCP-based and TCP/TLS-based media, where the latter uses TLS [16] for the purpose of authentication and maintaining the confidentiality of exchanged data. Both basic and complex data types in message signatures including Java basic types and Plain Old Java Objects (POJOs) are supported and serialized to XML through standard Java API for XML Binding (JAXB) [17]. Conversation Protocol Model Analyzer is able to examine protocols

```
public class SampleServerImpl implements SampleServer {
   private SampleClient sampleClient;
   public void setRemoteEndpoint(SampleClient remoteEndpoint) {
      this.sampleClient = remoteEndpoint;
   }
   public void foo(String text) {
     // entered 'intermediate' state
      this.sampleClient.bar("some business logic data");
     // entered 'final' state
   }
}
public class SampleClientImpl implements SampleClient {
   private SampleServer sampleServer;
   public void setRemoteEndpoint(SampleServer remoteEndpoint) {
      this.sampleServer = remoteEndpoint;
   }
   public void start() {
     // entered 'init' state
      this.sampleServer.foo("some business logic data");
     // entered 'intermediate' state
   }
   public void bar(String bar) {
     // entered 'final' state
   }
}
```

Fig. 5. Sample realizations of *SampleServer* and *SampleClient* interfaces (irrelevant code fragments omitted)

modelled as a pair of endpoint interfaces in terms of validity, based on the Petri Net formalism. The current distribution of CompTalks Framework includes command-line tools for running server and client endpoints and a set of sample demo protocols.

6 Validation and Tests

The CompTalks framework has been developed in the course of the ViroLab [2] Virtual Laboratory [3] project. CompTalks-powered GSEngine [1] forms the core of the Virtual Laboratory and enacts scientific workflows called *experiments*. These experiments are expressed as Ruby [18] scripts and are capable of accessing resources (such as data sets and services), interact with human actors and provide rich end-user experience. Enactment of scripts by a remote GSEngine service involves complex client-server interaction, including: streaming standard input and output, serving input forms to end users, uploading and downloading data and script files, tracing the status of experiments, sending results, notifications etc. All these features have been successfully implemented with CompTalks. Satisfactory stability and performance of the solution has allowed for employment of CompTalks in the production setup of the Virtual Laboratory.

Estimation of the performance of the CompTalks framework was carried out by measuring the time and memory required by the client in order to carry out a number of simultaneous benchmark conversations. Since the sample conversation discussed in this paper represents the simplest bidirectional message exchange, and its reference implementation consumes as little time and memory as possible, it is a useful benchmark for estimation of the overhead introduced by the framework itself.

The test results, presented in Fig. 6, were obtained on a testbed constituted by the 64-bit Sun Blade server, with dual-core Intel Xeon 5150 2.66GHz CPU running Ubuntu OS version 4.0.3 (Linux version 2.6.15-28-amd64-server) and HotSpot Java Virtual Machine version 1.6.0 update 10. Both client and server were running on the same machine and were communicating through the *localhost* network interface (RTT latency measured using ping equal to 0,006±0,001ms), via a plain TCP medium. The obtained results show that time and memory consumption is linearly dependent on the number of simultaneous conversations. Time consumption estimation is 235ms (bias) plus 1,6ms per each conversation. Memory consumption is estimated as 14,8MB (bias) plus 0,09MB per each conversation.

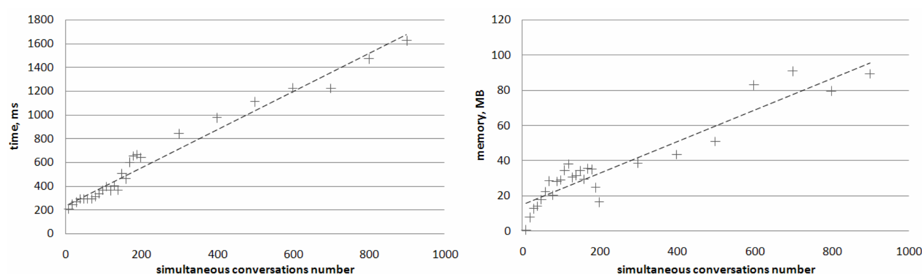

Fig. 6. Performance test results: time and memory required by the client to simultaneously carry out a number of benchmark conversations in the testbed environment

7 Conclusions and Future Prospects

The research described in this paper has shown that vital design and implementation issues related to a certain class of services, thus far unaddressed, can be remedied by using the proposed conversation protocols model. The example of GSEngine and ViroLab shows that this concept is applicable, and establishes the framework as a convenient and productive software development facility as well as effective middleware technology.

As we find this concept worth further research, the development of CompTalks is still ongoing and the following challenges are expected to be addressed next:

- Support for distribution, clustering and load balancing between servers by employing a master-worker architecture in order to make the solution scalable in terms of throughput (currently at a prototype stage).
- Enabling the Conversation Protocol Endpoint Implementation Analyzer to examine whether the endpoint implementation meets a given conversation protocol model. This will be based on ASM [19] Java Virtual Machine (JVM) [20] bytecode analysis and manipulation library (currently at a proof-of-concept stage).

The other further areas of research can be targeted at supporting multi-actors, multicast conversation and development of rich, interactive GUI design patterns that would integrate seamlessly with the proposed Conversation Meta-Model.

Acknowledgments. This work was partly funded by the European Commission, Project ViroLab IST-027446 and the related Polish grant SPUB-M, the AGH grant

11.11.120.777, and ACC CYFRONET-AGH grant 500-08. The authors wish to thank Tomasz Gubała and Piotr Nowakowski for their valuable advice.

References

1. Ciepiela, E., Kocot, J., Gubala, T., Malawski, M., Kasztelnik, M., Bubak, M.: Virtual Laboratory Engine – GridSpace Engine. In: Cracow Grid Workshop 2007 Workshop Proceedings, ACC CYFRONET AGH, pp. 53–58 (2008)
2. ViroLab - virtual laboratory for decision support in viral deseases treatment, http://virolab.org/
3. Bubak, M., et al.: Virtual Laboratory for Development and Execution of Biomadical Collaborative Applications. In: Puuronen, S., Pechenizkiy, M., Tsymbal, A., Lee, D.-J. (eds.) Proc. 21st IEEE International Symposium on Computer-Based Medical Systems, Jyvaskyla, Finland, June 17-19, pp. 373–378 (2008) doi 10.1109/CBMS.2008.47
4. World Wide Web Consortium (W3C) Web Services activity, http://www.w3.org/2002/ws/
5. Web Services Description Language (WSDL) Specification 1.1, W3C Note (March 15, 2001), http://www.w3.org/TR/wsdl
6. Web Services Conversation Language (WSCL) 1.0, W3C Note (March 14, 2002), http://www.w3.org/TR/wscl10/
7. Simple Object Access Protocol (SOAP) 1.1, W3C Note (May 8, 2000), http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
8. Web Services Resource (WS-Resource) 1.2, OASIS Standard (April 1, 2006), http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf
9. Web Services Transfer (WS-Transfer),W3C Member Submission (September 27, 2006), http://www.w3.org/Submission/WS-Transfer/
10. Fielding, R.T., Taylor, R.N.: Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology (TOIT) 2(2), 115–150 (2002) doi:10.1145/514183.514185
11. Foster, I., Parastatidis, S., Watson, P., McKeown, M.: How do I model state?: Let me count the ways. Commun. ACM 51(9), 34–41 (2008)
12. Extensible Messaging and Presence Protocol XMPP home page, http://xmpp.org/
13. Extensible Messaging and Presence Protocol (XMPP): Core, IETF RFC 3920 (October 2004), http://www.ietf.org/rfc/rfc3920.txt
14. The Blocks Extensible Exchange Protocol Core, IETF RFC 3080 (March 2001), http://www.ietf.org/rfc/rfc3080.txt
15. Peterson, J.L.: Petri Nets. ACM Computing Surveys 9(3), 223–252 (1977) doi:10.1145/356698.356702
16. Transport Layer Security (TLS), IETF RFC 5246 (August 2008), http://www.ietf.org/rfc/rfc5246.txt
17. JSR-222 Java Architecture for XML Binding (JAXB), http://jcp.org/aboutJava/communityprocess/mrel/jsr222/
18. Ruby Programming Language, http://www.ruby-lang.org/en/
19. ASM - all purpose Java bytecode manipulation and analysis framework, http://asm.ow2.org/
20. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn.
21. Rosmanith, H., Volkert, J.: glogin - Interactive Connectivity for the Grid. In: Juhasz, Z., Kacsuk, P., Kranzlmüller, D. (eds.) Distributed and Parallel Systems - Cluster and Grid Computing, Proc. of DAPSYS 2004, 5th Austrian-Hungarian Workshop on Distributed and Parallel Systems, September 2004, pp. 3–11. Kluwer Academic Publishers, Budapest (2004)