Extended WS-Agreement Protocol to Support Multi-Round Negotiations and Renegotiations*

Christoph Langguth and Heiko Schuldt

Databases and Information Systems Group
University of Basel, Switzerland

Abstract. WS-Agreement is a well-established and widely adopted protocol that helps service providers and consumers to agree on constraints under which a service is made available. However, the original protocol is limited to a simple interaction pattern for establishing agreements: the requester suggests the Quality of Service (QoS) details, the responder either accepts or declines. This is no longer sufficient when several rounds of negotiations are needed before both parties agree on the QoS level to be provided, or when an already established agreement needs to be changed based on mutual consent (renegotiation). This paper presents an extension to WS-Agreement which jointly addresses these limitations.

1 Introduction

As Service-oriented Architectures (SOA), and in particular Web Services (WS), are constantly gaining in popularity and adoption, the research focus is broadening more and more towards non-functional service properties. A prime example is Quality of Service (QoS): in a production system, customers may demand for guarantees about a service's QoS, for instance because a best-effort-only provision of the service could lead to untolerable latencies when the system gets heavily loaded. If guarantees can be given, they are generally expressed in a Service Level Agreement (SLA), which can be considered a binding contract between provider and consumer.

WS-Agreement [2] is a well-established specification that allows to express and manage such SLAs. It defines a standardized protocol for managing agreements, while being flexible concerning their actual (domain-specific) content. However, it has a few shortcomings that we point out in the following use case.

Consider the example of a workflow engine W that orchestrates a process consisting of several service invocations – say, service S_1 followed by S_2. If each individual service provider participating in the workflow could assure proper QoS guarantees (e.g., on the availability of local resources to guarantee a certain execution time; or to guarantee an upper bound on the cost incurring during execution), the predictability of the individual service calls – and therefore also of the process as a whole – would be greatly improved. Thus, W could support the negotiation of QoS agreements for the overall process with its own client C.

* This work has been partly supported by the Hasler Foundation in the *COSA* project.

However, as the workflow engine needs to act as a kind of mediator and has to rely on QoS guarantees of the actual service providers, it has to individually negotiate agreements with the service providers as well. The QoS guarantees of the overall process need to be derived from "internal" negotiations with the service providers. In the case of long-running workflows encompassing resource-intensive services, each QoS agreement with a service provider might include several parameters (run-time, local resources, execution cost) that need to be jointly negotiated in a single agreement.

We have used the term *negotiation* in its literal sense, as this is a desirable feature: C could send an agreement offer to W, which, in trying to find a configuration that can satisfy the SLA terms, in turn needs to negotiate with providers offering S_1 and S_2. Clearly, if an agreement responder is able to send "counter-offers" to proposals, instead of just rejecting them (and forcing a subsequent attempt with a different proposal), the negotiation process is significantly facilitated. One shortcoming of WS-Agreement is that the specification does not allow for such multi-round negotiations, but only considers "one-shot" agreement creation, i.e., the responder has to immediately accept or reject an offer.

Furthermore, existing and valid agreements cannot be modified once established, except by terminating the existing SLA and creating a new one. Yet, the likeliness of "things going wrong" increases with the complexity of a workflow. Suppose that the provider of S_1 realizes it cannot assure the QoS it committed to. Being able to renegotiate with W (which may in turn trigger other renegotiations with providers of subsequent workflow activities) might limit the negative effects, whereas being forced to terminate the SLA inevitably leads to all agreements – and thus the workflow execution as well – having to be terminated.

This scenario stems directly from our ongoing research on DWARFS[4], which uses Advance Reservations for delivering QoS guarantees at workflow level, particularly for complex, long-running and resource-intensive scientific workflows. We consider the limitations of the WS-Agreement specification crucial. Thus, we devised and implemented an extension which: a) allows for multi-round agreement negotiations, as opposed to the existing single-shot "offer-accept-or-reject" creation; b) supports renegotiations, i.e., changing SLA terms while an agreement is in effect; c) is symmetric concerning the options that the involved parties have to operate on an agreement (e.g., allows also the agreement responder to terminate it); d) strives for maximum possible downward-compatibility.

The remainder of this paper is structured as follows: Section 2 analyzes the original WS-Agreement specification and its limitations. Section 3 provides detailed information about our extensions to the specification. Section 4 presents related work. Finally, Section 5 concludes.

2 The Original WS-Agreement Specification

Figure 1 shows a high-level overview of the lifecycle of an agreement (for the moment, consider only the "non-bold" part of the figure). The lifecycle states can be related to the port types defined in the specification as follows. An agreement passes through the **negotiating** state by two possible sequences of actions:

- The agreement initiator invokes the `createAgreement` operation of the `AgreementFactory` port type. This operation either returns an EPR to an `Agreement` resource, which corresponds to the **accept** transition, or it throws a fault, which corresponds to the **reject** transition.
- The agreement initiator invokes the `createPendingAgreement` operation of the `PendingAgreementFactory` port type, passing it an EPR to an `AgreementAcceptance` resource. The `PendingAgreementFactory` decides on whether it accepts the offer, and calls back the respective operation (`accept` or `reject`) on the `AgreementAcceptance`. This callback invocation directly corresponds to the transition of the same name.

If the agreement ends up in state **void**, the semantics are the same as if the agreement never existed, i.e., a contract has never taken place. Once an agreement is in state **effective**, the specification allows for two further transitions: The agreement comes to its "normal" end of lifetime, i.e., it *successfully* passes its expiration time. The transition is implicit in the specification, and is taking place when "[...] an agreement is no longer valid, and the parties are no-longer obligated by the terms of the agreement" [2]. This transition is made explicit in the figure, leading to the **completed** state. The second possible transition can only be triggered by the agreement initiator by calling the `terminate` operation on the `Agreement` resource. This leads to the **terminated** state, signifying that the agreement was terminated *unsuccessfully.*

There are several implications of this protocol design. First, the negotiation is in fact a simple request/response operation, in which the agreement initiator sends an offer that the responder either accepts or rejects. Second, only the initiator may terminate an agreement. There is no possibility for the agreement responder to terminate an agreement. Third, an agreement may not be modified after its creation – neither by the initiator nor by the responder. Fourth, when an agreement is created using the callback mechanism (`PendingAgreementFactory/AgreementAcceptance`), no provision is taken for the possible case of the responder not answering.

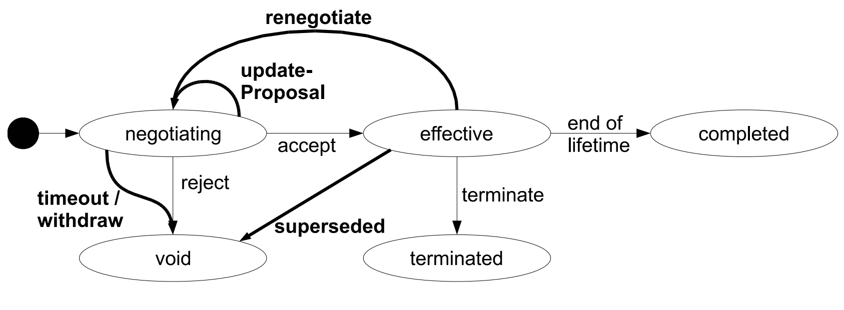

Fig. 1. Lifecycle of an Agreement (extensions in **bold**)

Port Type	**Operations**	**Resource Properties**
`PendingAgreementFactory`	createPendingAgreement	Template
`Agreement`	*accept*	Name
	reject	Id
	terminate	Context
	updateProposal	Terms
	extendDecisionDeadline	*AgreementState*
	withdraw	*ServiceTermStateList*
	renegotiate	*GuaranteeTermStateList*
		AgreementServiceReferenceList
`AgreementAcceptance`	accept	*Name*
	reject	*Id*
	terminate	*Context*
	updateProposal	*Terms*
	extendDecisionDeadline	*AgreementState*
	withdraw	
	renegotiate	

Operations/Resource Properties: unchanged, *reused*, **new**

Table 1. Modified Port Types in the extended WS-Agreement Protocol

The first three issues share one common aspect: they can only be fully addressed if the asymmetry of the protocol is broken. By asymmetry we mean that according to the specification, only the responder keeps track of the agreement, while only the initiator may operate on it (e.g., call the `terminate` operation).

3 Extended WS-Agreement Protocol

Overcoming these limitations is possible by promoting the agreement initiator role to actually (also) *represent* the SLA. This is already partially done in the original specification by the definition of the `AgreementAcceptance` port type, albeit it is defined to only provide callback methods. As an example, consider the second issue: if an `AgreementAcceptance` also provides a `terminate` operation, then it becomes possible for the responder to terminate an agreement.

3.1 Modifications of Port Types

In that spirit, we have modified the WS-Agreement port types as shown in Table 1. The `Agreement` and `AgreementAcceptance` port types now present a much more symmetric interface: both provide exactly the same operations, and the `AgreementAcceptance` "mirrors" to a large extent the resource properties (RP) of the `Agreement` port type. While we describe the operations in more detail in Section 3.2, in the following we make a few important observations.

First, the `AgreementFactory` port type is not used. Even in the specification, this port type merely presents a simpler alternative to the use of the `Pending-AgreementFactory` in conjunction with the `AgreementAcceptance`. Since we heavily rely on the latter, the `AgreementFactory` is simply superfluous. While the `AgreementState` port type is not mentioned explicitly in Table 1, it is actually being used: according to [2], this port type "is not meant to be used as is but instead, its resource properties may be composed into a domain-specific

Agreement port type". In our proposal, the `Agreement` and `AgreementAcceptance` both represent the agreement, therefore we integrate the additional RPs into these port types. As we do not want our system to permanently "synchronize" system state from one port type to another, the `ServiceTermStateList` and the `GuaranteeTermStateList` RPs are only available at the service provider side (`Agreement`), while the `AgreementState` RP is part of both `Agreement` and `AgreementAcceptance`.

The full state diagram that the modified port types allow is shown in Figure 1. We have not introduced new states, but several new transitions. Most notably, the agreement can stay in state **negotiating** for arbitrarily long, by repeatedly following the **updateProposal** transition. Thus, the agreement responder no longer needs to immediately accept or reject an agreement, but may instead send a counter-offer to the initiator, who then has to accept or reject it, or in turn come up with another proposal. This "ping-pong" style is the reason why the operations and RPs of `Agreement` and `AgreementAcceptance` are to a large extent symmetric: both agreement partners now have equal roles in the negotiation. The concept of having this alternation of responsibility to decide about the "fate" of an agreement means that while the agreement is in the **negotiating** state, there is always one involved party waiting for a decision from its partner. To cope with the possible case of that partner not answering anymore, we have introduced a *decision deadline*: this is the time by which a decision must have been taken, or the agreement is cancelled by the waiting party (transition **timeout** into state **void**).

3.2 Operations in Detail

The following list gives an overview of the newly introduced operations. In the interest of space, operations which were merely "copied" from the `Agreement` to the `AgreementAcceptance` or vice-versa, are not discussed, as their semantics remains the same. Figure 2 shows sequence diagrams of how these operations may play together during the initial negotiation and renegotiation of agreements.[1]

- `updateProposal`: This operation allows for a multi-round negotiation process. If one of the agreement partners receives an agreement proposal that it neither accepts nor rejects, it responds by sending a proposal which would be accepted instead, by invoking the `updateProposal` operation at the partner's endpoint. The method has one parameter, which is the proposal, and it does not return any value. An invocation of the `updateProposal` operation should always be responded to by invoking either the `accept` or `reject` operation, or by sending yet another proposal.
- `extendDecisionDeadline`: As described earlier, there is a deadline to prevent timeouts and to give agreement partners the possibility to limit the time for decision making. It is initially defined by the service provider, but

[1] To avoid clutter, illustration of the decision making process is limited to the initiator's side. The analogous parts on the responder's side have been omitted.

either of the negotiation parties can request an extension by invoking the `extendDecisionDeadline` operation on its partner's endpoint. The operation does not expect any parameters. If the invoked party is willing to accept the extension, it returns the new deadline or else throws a fault.
- `withdraw`: If one of the parties has sent a proposal to the other partner and has not yet received an answer, it still has the possibility to abort the negotiation process by calling the `withdraw` operation at the partner's endpoint.
- `renegotiate`: Either partner can request renegotiation of an existing agreement. This is described in more detail in Section 3.3.

3.3 Renegotiation

As we have mentioned, the possibility to renegotiate an existing agreement is of fundamental importance. As shown in Figure 2, invocation of the `renegotiate` operation creates two new resources (`Agreement` and `AgreementAcceptance`).[2] The partners then carry out the renegotiation in exactly the same style as a normal multi-round negotiation, using these new endpoints. Once that negotiation process comes to an end, the following actions depend on its outcome. In case of a successful negotiation, the newly created agreement enters the **effective** state, superseding the existing one, which gets discarded (transition **superseded**). If the negotiation process is not successful, the existing agreement stays in place unchanged (and the newly created one is **void**).

3.4 Compatibility with the Original Specification

With the exception of the (dispensable) `AgreementFactory` port type, our extension is completely downward-compatible with the existing WS-Agreement specification. In other words, if none of our extensions are being used, i.e., the responder immediately accepts or rejects the first proposal, and no renegotiation takes place, the interaction patterns correspond exactly to the original specification. The only additional data that needs to be communicated is the decision deadline; it was added to the `wsag:Context` (which allows for `xs:any`). Thus, we ensure interoperability with endpoints not using our extension – but it should be noted that this will not hold anymore if one party *requires* our extension.

4 Related work

The shortcomings of WS-Agreement that our extension deals with have been addressed at various levels. A formal analysis of the structure of an agreement and its internal state is presented in [3]. The authors analyze in detail *when* an agreement may need to be renegotiated, and propose an extension to the agreement representation that allows for (in-place) modification of existing agreement

[2] The figure only shows the case of the initiator triggering a renegotiation. The symmetric case is, of course, also possible.

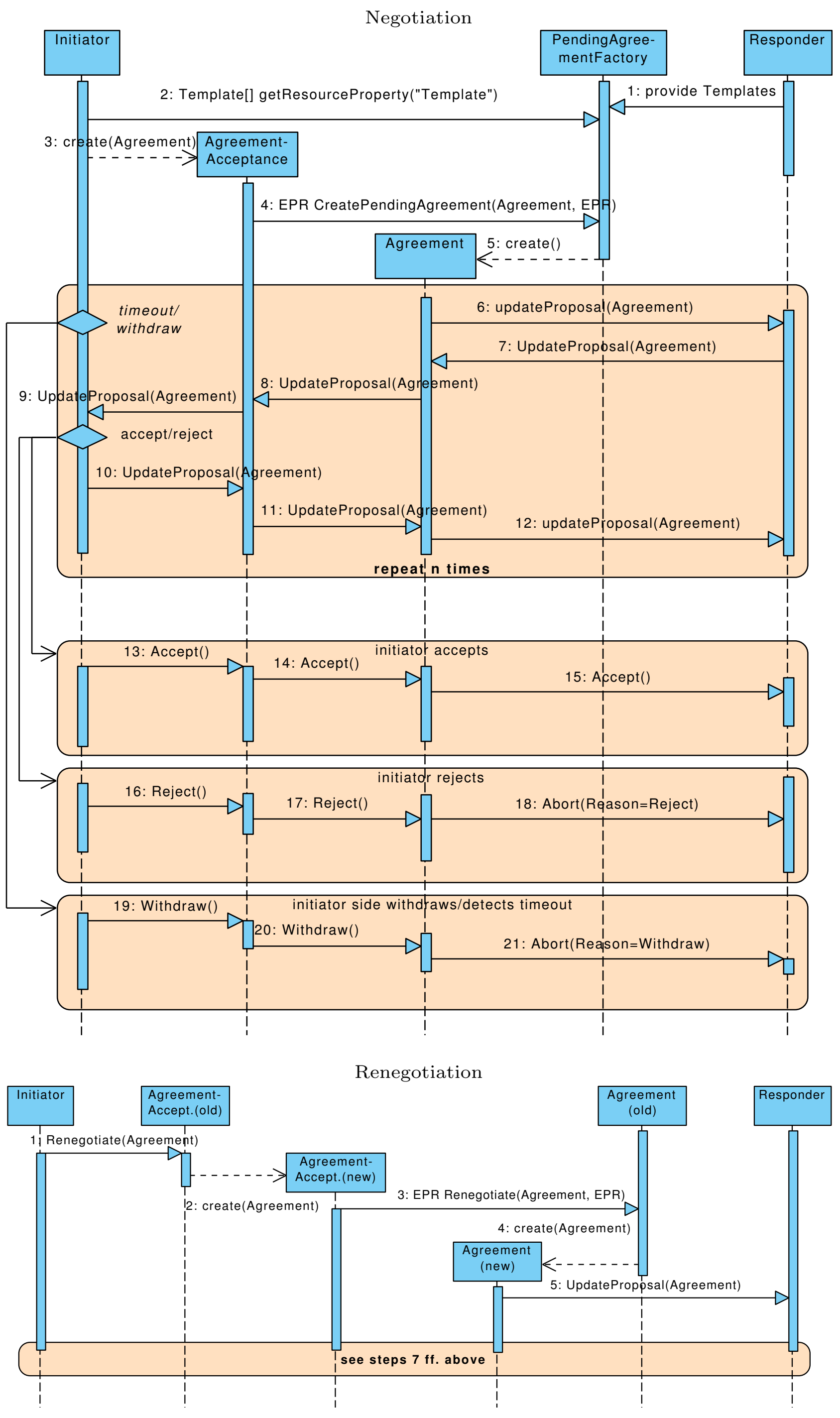

Fig. 2. Sequence Diagrams: Agreement Negotiation and Renegotiation

terms; however, *how* a renegotiation can be performed is not addressed. Multi-round negotiation is discussed in [7, 8] using an approach similar to ours, but performing the negotiation using agreement templates rather than actual agreement offers. A protocol that focuses on renegotiation only is described in [6]. It proposes multiple operations that allow for renegotiation, whereas our approach mostly reuses the multi-round negotiation capabilities for renegotiation; also, the inherent asymmetry is not addressed, thus enabling only the agreement initiator to request a renegotiation. Finally, [5] proposes new port types that allow for a (symmetric) renegotiation, whereas we extend the existing port types. In addition, our proposal does not require any changes to the structure of an agreement representation, but deliberately strives to be "agnostic" of the SLA content.

5 Conclusions

We have described an extension of the WS-Agreement protocol which allows for multi-round agreement negotiation. It also enables renegotiation of existing SLAs, giving both agreement partners the opportunity to trigger such renegotiations. Our proposal requires no modifications to the structure of an agreement and, to a large extent, reuses the operations and resource properties defined in the specification, assigning them in a symmetric fashion to the original port types. This approach allows us to stay compatible with the original specification.

Acknowledgement: The authors would like to thank David Ammann for his support in the implementation and evaluation of this proposal.

References

1. D. Ammann. Design and Implementation of a Negotiation Protocol for Scientific Workflows based on WS-Agreement. Master's thesis, University of Basel, 2009.
2. A. Andrieux et al. Web Services Agreement Specification. Specification, Open Grid Forum, 2007. `http://www.ogf.org/documents/GFD.107.pdf`.
3. G. Frankova, D. Malfatti, and M. Aiello. Semantics and Extensions of WS-Agreement. *Journal of Software*, 1(1), 2006.
4. C. Langguth, P. Ranaldi, and H. Schuldt. Towards Quality of Service in Scientific Workflows by using Advance Resource Reservations. In *IEEE 2009 Third International Workshop on Scientific Workflows (SWF 2009)*, 2009.
5. G. D. Modica, O. Tomarchio, and L. Vita. Dynamic SLAs management in service oriented environments. *Journal of Systems and Software*, 82(5):759–771, 2009.
6. M. Parkin, P. Hasselmeyer, B. Koller, and P. Wieder. An SLA Re-Negotiation Protocol. In *2nd Non Functional Properties and Service Level Agreements in Service Oriented Computing Workshop (NFPSLA-SOC 08)*, November 2008.
7. A. Pichot, O. Wäldrich, W. Ziegler, and P. Wieder. Towards Dynamic Service Level Agreement Negotiation: An Approach Based on WS-Agreement. In *WEBIST (Selected Papers)*, pages 107–119, 2008.
8. W. Ziegler, P. Wieder, and D. Battr. Extending WS-Agreement for dynamic negotiation of Service Level Agreements. Technical Report TR-0172, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, August 2008.