The Web Services-BusinessActivity–Initiator (WS-BA–I) Protocol: an Extension to the Web Services-BusinessActivity Specification

Hannes Erven*, Georg Hicker*, Christian Huemer* and Marco Zaptletal°
* *Business Informatics Group,* ° *Electronic Commerce Group*
Institute of Software Technology and Interactive Systems, Vienna University of Technology, Austria
hannes@erven.at, *georg.hicker@reflex.at*, *huemer@big.tuwien.ac.at*, *marco@ec.tuwien.ac.at*

Abstract

The Web Services Transaction protocol family includes the WS-AtomicTransaction and the WS-BusinessActivity specifications in order to carry out distributed transactions in a Web Services (WS) environment. The WS-AtomicTransaction specification defines all necessary interfaces to carry out transactional work. In contrary, the WS-BusinessActivity specification for long-running transactions intentionally left the interface between initiator and coordinator undefined. This allows vendors to integrate WS-BusinessActivity coordinators into their business process engines. However, it requires proprietary protocols between initiator and coordinator. We propose an extension protocol to the WS-BusinessActivity specification that explicitly defines this interface between initiator and coordinator. This extension allows coordinators and initiators from different vendors to interoperate transparently. Accordingly, participants no longer need to trust an initiator-selected and likely initiator-run coordination service, but may use commonly trusted, third-party coordination services.

1. Introduction

The *WS-BusinessActivity* specification [1] was created to enable web services to participate in long-running, loosely coupled transactions. This type of transaction is commonly used when modelling business processes, as for example in workflow systems where the workflow execution engine is responsible for coordinating the participants as defined by the rules of the business processes.

At present, hardly any web service consuming software is backed by a workflow engine that provides *WS-BusinessActivity* coordination services. Hence there is a need for independent, easy to use third party tools realizing business activity coordination services. These will enable environments without workflow engines to create, coordinate and participate in long-running transactions.

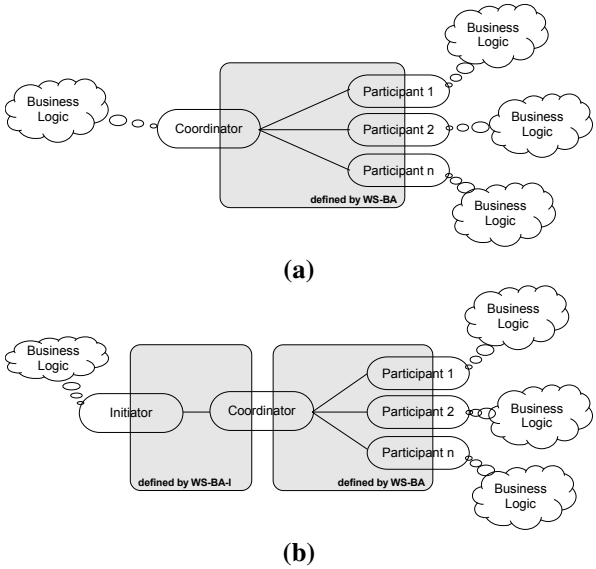

Figure 1. *WS-BusinessActivity* and *WS-BusinessActivity–Initiator* overview

The *WS-BusinessActivity* specification leaves the choice of the coordination service to the application. This is depicted in Figure 1a. The lack of a well-defined interface between application and coordinator requires a coordination service that is either built into the application or tightly coupled to it. In other words, the business logic is tightly coupled to the coordination service. A participant has to trust this proprietary coordination service.

A "trusted third party" model, similar to the one used for PKI certificate authorities, adds transaction

2007 IEEE International Conference on Web Services (ICWS 2007)
0-7695-2924-0/07 $25.00 © 2007 IEEE

security to participants and initiators. However, this model requires a common protocol between initiator and coordination service. In such a model, a coordination service records all state changes and notifications. This means that a transaction's progress is always reliably recorded by an independent party.

In this paper we introduce the *WS-BusinessActivity–Initiator* protocol to be implemented between an initiator and a coordination service. In Figure 1b we illustrate how *WS-BusinessActivity–Initiator* and *WS-BusinessActivity* work together. The *WS-BusinessActivity–Initiator* specification proposes a protocol that enables an initiator to enlist participants, to check on their current state and to inform the coordination service about decisions it has made. The current proposal is designed as a "pull" protocol. Accordingly, the initiator is a simple client that does not offer any services to the transaction coordinator or participants. This enables even applications that do not run in web service containers and/or are behind firewalls or on NAT networks to create and control *WS-BusinessActivity* transactions. The approach we present in this paper has been implemented and contributed to the Apache Kandula project [2]. A WSDL description specifying the operations required by the *WS-BusinessActivity–Initiator* protocol is found at [3].

The remainder of this paper is structured as follows: section 2 elaborates related work. In section 3 we briefly introduce the most important concepts of the *WS-BusinessActivity* standard. Afterwards, section 4 motivates the need for a standardized protocol between initiator and coordinator. In section 5 we introduce this protocol in detail and illustrate it by a simple example. A short summary in section 6 concludes the paper.

2. Related work

For many years transaction processing has attracted a lot of attention. There exist various implementations of the basic concepts and techniques as described by Gray and Reuter [4]. Transaction processing is not only of great importance in database systems, but also in distributed systems. The Object Management Group has released a specification that provides transaction synchronization across the elements of a distributed client/server application [5]. In this paper, we focus on the transaction processing of a certain type of client/server applications, i.e. Web Services.

The *WS-BusinessActivity* specifcation [1] together with the *WS-AtomicTransaction* [6] and the *WS-Coordination* specifications [7] forms the *WS-Transaction* family, which is also referred to as the Web Services Transaction Framework (WSTF). Within this set of specifications, *WS-Coordination* provides the foundation for implementing transactional web service interactions. It defines the coordination context of a transaction and protocols for registrating services therein. *WS-AtomicTransaction* and *WS-BusinessActivity* build on *WS-Coordination* by specifying differing types of transactional behavior. *WS-AtomicTransaction* focuses on transactions according to the ACID principle: **A**tomicity, **C**onsistency, **I**solation, and **D**urability. It implements the two-phase commit protocol. *WS-BusinessActivity*, on the other hand, focuses on long-running transactions allowing human interactions and mixed outcomes.

This differentiation corresponds to the two kinds of business transactions for e-business applications as proposed in [8] and [9]. Atomic business transactions, on the one hand, are small scale interactions that are composed of services that agree on enforcing a common outcome. This means, that either all services commit or all services roll-back ensuring an atomic outcome. Long-running business transactions, on the other hand, aggregate several atomic transactions. They feature the same behavior as open nested transactions by allowing their sub-transactions to commit independently [10]. The superimposed business logic is then in charge of deciding on the overall outcome of the long-running transaction.

In [11], Papazoglu stresses the importance of transactional capabilities in terms of collaborative business processes. He proposes a business transaction framework (BTF) orchestrating loosely coupled web services into a single business transaction backed by transactional support. The BTF is responsible for coordinating distributed autonomous business functionality and guarantees coordinated and predictable outcome for the participating business partners. The work details the functional criteria and the required components of a BTF and compares standards such as the *Business Process Execution Language* (BPEL) [12], *WS-AtomicTransaction*, *WS-Coordination*, the *Business Transaction Protocol* (BTP) [13] and the *ebXML Business Process Specification Schema* (BPSS) [14] against it.

The Business Transaction Protocol (BTP), standardized by OASIS, is an analog initiative for coordinating service interactions. It aims at defining a protocol for representing and seamlessly managing complex, multi-step business-to-business (B2B) transactions over the Internet. The BTP is not restricted to the web service environment, but might be layered over other frameworks like RosettaNet [15] or ebXML [16]. However, in the web services world, the WSTF currently seems to be better supported than BTP.

Vogt et al. outline in [17] that *WS-Coordination* and *WS-BusinessActivity* violate the SOA paradigm by making assumptions about the internal structures of

the participants. They underpin their assertion by an analysis based on a proof-of-concept implementation of *WS-Coordination* and *WS-BusinessActivity* . As a result of this analysis they discover a tight-coupling between the entities participating in the transaction (coordinator, initiator, and participant). Their approach introduces a transactional middleware that separates the client's business logic from the coordination logic. However, the middleware is still required to deal with business messages which prohibits a complete seperation of concerns. It follows, that their approach does not allow for a third-party coordinator service as proposed in our work.

The work by Sauter and Melzer [18] focuses on a comparison of *WS-BusinessActivity* and BPEL with respect to long-running transactions. They show the differences as well as the similarities between these standards and discuss that they are not contradicting approaches as sometimes argued. Furthermore, their work proposes to integrate *WS-BusinessActivity* into BPEL to allow for distributed coordination. Therefore, BPEL's compensation handler is replaced by a coordination handler that coordinates nested scopes via SOAP messages.

3. The *WS-BusinessActivity* protocol

The *WS-BusinessActivity* standard [1] specifies the coordination type *business activity* as defined in the extendable *WS-Coordination* framework [7]. This coordination type defines long running transactions which could consist of several atomic transactions. In this coordination type processing of requests could take some time, since human interaction or the assembly of parts could be necessary. This means an un-do cannot be realized by a roll-back function. Instead an un-do requires a sophisticated compensation mechanism involving business logic. For example the cancelation of a booking requires a compensation mechanism that may include the billing of cancelation fees.

The following attributes characterize transactions of the *business activity* coordination type [1]:

- All state changes made during a transaction are recorded. Especially application state and coordination metadata are collected to enable later compensation.

- Messages have to be acknowledged so that coordinator and participant have the same perception of state.

- Each message has to be transmitted individually. This means that it is not allowed to combine messages.

The *WS-BusinessActivity* specification allows nesting business activities. Consequently, a business activity may be included in another one. Nested transactions allow intercepting and handling errors that should not lead to a failure of the parent transaction. A popular example for a nested structure is the supply of product components. Consider, components are urgently needed. A nested transaction is initiated to contact the contractor delivering at the lowest price. If the contractor is not able to deliver in time, the nested transaction does not necessarily fail, since another contractor may be able to supply the components in time.

WS-AtomicTransaction is based on the classic ACID properties. In contrast, *WS-BusinessActivity* had to soften this classical transaction paradigm. It supports so-called long running transactions that do not lock datasets. This allows e.g. waiting for user input or assembly of parts. In all these cases locks are not possible because they would block other business processes. Therefore results and changes during *business activities* are immediately visible to the outside of a transaction. Locks on datasets are held as short as possible or, better, are not held at all. Since locks are sometimes critical to the overall system, *WS-BusinessActivity* may be combined with nested atomic transactions.

WS-BusinessActivity specifies two coordination types: (*AtomicOutcome* and *MixedOutcome*. Each of them may be use in the two coordination protocols described in the following subsections.

3.1. *BusinessAgreementWithParticipantCompletion* protocol (BAwPC)

We depict the lifecycle of a transaction for the BAwPC protocol in Figure 2 [1]. It shows the states of the transaction and the messages that lead to state changes. A participant registered for this protocol decides when he has done all work for an activity. The participant informs the coordinator about finishing his work by means of a *Completed* message.

The coordinator may reply with a *Close* or with a *Compensate* message upon receipt of a *Completed* message. A *Close* message signals to the participant that the transaction is ended in a positive way. In case the coordinator wants to undo work and to restore the initial original conditions of data, he sends a *Compensate* message.

A problem concerning the message order is the fact that the coordinator may receive a message which is not effective to the already reached state. In this case the coordinator has to revert to its prior state and handle the received message accordingly. Moreover a party, regardless whether coordinator or participant, has to be

prepared to receive duplicate protocol messages and has to react according to the specification. The party has to either resend a prior message or ignore the duplicate as defined in Appendix A of the specification [1].

The participant signals an error by sending a *Fault* message to the coordinator. The coordinator acknowledges this message by a *Faulted* message. After sending this message the protocol instance is terminated and no more messages can be exchanged.

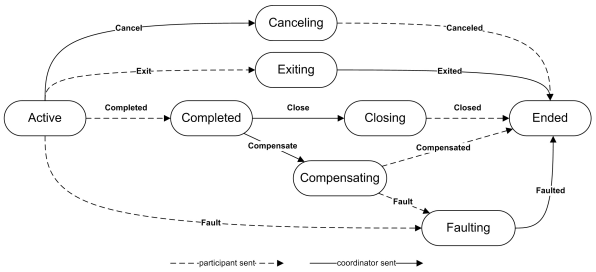

Figure 2. Lifecycle of the *BusinessAgreement-WithParticipantCompletion* protocol

The coordinator can end a transaction by sending a *Cancel* message while the transaction is in *Active* state. However the specification does not illustrate how the coordinator should decide to end the business activity. This task has to be handled by the business logic. In any case, the participant has to answer with a *Canceled* message.

3.2. *BusinessAgreementWithCoordinatorCompletion* protocol (BAwCC)

In the *BAwPC* decribed before the participant does decide when the activity is terminated. The coordinator makes this decision in the *BAwCC*. This difference becomes obvious in Figure 3 which depicts the lifecycle of a transaction in the *BAwCC* protocol [1]. In this protocol the coordinator sends a *Complete* message which is followed by a *Completed* message of the participant. As a consequence, an additional state *Completing* is required between the states *Active* and *Completed*. Once having reached the state *Completed* the rest of the lifecycle is the same as for the *BAwPC* protocol described before.

4. Motivation for the *WS-BusinessActivity–Initiator* protocol

The *WS-BusinessActivity* protocol specification concentrates on the interaction between the coordinator and the participant roles. It does intentionally not specify how the transaction can actually be managed by the

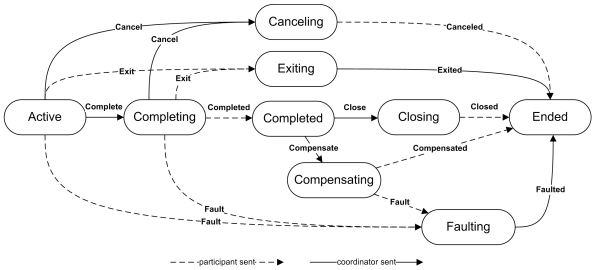

Figure 3. Lifecycle of the *BusinessAgreement-WithCoordinatorCompletion* protocol

corresponding business logic. The authors of the *WS-BusinessActivity* specification expect workflow engines to provide proprietary interfaces for workflows to manage their tasks with *WS-BusinessActivity*. While *WS-AtomicTransaction*'s *CompletionProtocol* also defines an interface between the initiator and coordinator roles, it does not have an equivalent in *WS-BusinessActivity*.

The tight coupling between coordinator and initiator violates common software architecture principles, since distinct roles always should have an explicit communications protocol defined between them. A clear separation of concerns vastly enhances software modularity and reuseability.

The *WS-BusinessActivity* protocol also does not specify how the initiator's business logic may map business partners to transaction participants. Hence, the initiator is not able to assess the meaning of a transaction's state to the business task he is performing.

As a solution, we propose the *WS-BusinessActivity–Initiator* protocol that defines the additional interface. It is designed as a "pull" based protocol where the initiator frequently polls the coordinator for the context's and his participants states. The protocol design offers the following advantages:

Firstly, it relieves the business logic's container of providing a web service interface to participants and/or the coordinator. Accordingly, even environments that cannot provide SOAP endpoints may take on the initiator role.

Secondly, it does no more require that coordinator and/or participants are able to actively contact the entity containing the business logic. The business logic may sit on a restricted, NAT- or firewalled network or even be temporarily disconnected.

Thirdly, it requires that every participant of a transaction is individually "invited" by the business logic. Thereby it maintains a business-partner to transaction-participant relationship.

Fourthly, it allows independent, third parties to offer coordination-only services. The third party could

also reliably record the transaction's progress for later assessment.

In a nutshell, one could say the protocol "brings *WS-BusinessActivity* to the desktop" since it allows creating desktop applications that manage *WS-BusinessActivity* transaction contexts.

In order to coordinate the transaction's progress, the initiator role needs to be able to perform the following tasks: (i) notify the coordinator when inviting new participants; (ii) decide whether the context's overall goals can be achieved by querying the participants' states; (iii) send the *Complete* and *Cancel* commands to individual participants; (iv) in *mixed* outcome contexts, direct each participant to either *Close* or *Compensate*; and (v), in *atomic* outcome contexts, decide whether to *Close* or to *Cancel/Compensate* all work.

The *WS-BusinessActivity–Initiator* protocol defines the interface that allows a business logic implementation to perform those tasks by sending its decisions to the coordinator service. The coordinator service actually executes these decisions by sending the appropriate messages to the participants.

5. The *WS-BusinessActivity–Initiator* protocol

In order to create a new transaction context the initiator contacts the *activation service*. In our proposed extension the initiator shall immediately register itself for the *WS-BusinessActivity–Initiator* protocol by contacting the *registration service* associated with the context. The coordinator will only allow one initiator per context and will reject all further registration requests for that protocol.

When successfully registered, the initiator uses the *WS-BusinessActivity–Initiator protocol service* to communicate with the transaction context. It enables the initiator to perform the tasks necessary to manage the transaction's progress. The inititor may request coordination context data for additional participants. Furthermore, it may also query the current states of the enrolled participants. It may ask the coordinator to send *WS-BusinessActivity* messages to the participants directing them through the *WS-BusinessActivity* protocol's states. When the initiator is managing a subcontext, it may ask the coordinator to send messages to its supercontext. These concepts realized in our *WS-BusinessActivity–Initiator* protocol are detailed in the following subsections.

5.1. Registration and Participant Identification

In a business activity, participants are invited by the initiator to join the transaction by passing a *WS-Coordination CoordinationContext* object with an application message to them. The *CoordinationContext* object includes the registration service's endpoint address and the transaction identifier.

The initiator must be able to map the business-level partners to transaction-level participants. In order to facilitate the identification of participants, the method `createCoordinationContext` opens a registration ticket in the context that is bound to an initiator provided "match code". Thus, the "match code" extends the *CoordinationContext* object as defined in *WS-Coordination* (see [7]). The initiator passes this object including the "match code" on to the business partner, who in turn registers using the data from the received *CoordinationContext* object. The *registration service* automatically associates the registering participant with the match code. When issuing subsequent commands, the initiator may now address individual participants with their match codes.

Although the *WS-BusinessActivity* specification does not explicitly define whether a particular "invitation ticket" (*CoordinationContext* object) may be used for multiple registrations, this specification restricts registrations to one single participant per "invitation". The "identification" design goal implies that every participant is assigned to a unique match code, so a particular *CoordinationContext* object must not be used for multiple registrations. Should a business partner listed as a participant require in turn to enlist multiple participants to complete his work, he may create a subcontext (which is presented as one single participant to its supercontext) to manage those participants.

5.2. Transaction State Reporting

The initiator may at any time ask the coordinator for a complete list of participants enrolled in a transaction. Each participant in this list is characterized as follows:

- the participant's match code
- the protocol the participant registered for
- the participant's current state
- the participant's result state

Based on this information the initiator application may decide what commands to issue. The "participant result" offers the initiator information about what outcome the participant has agreed on. When the participant has reached the *Ended* state, the result is either *Canceling*, *Closing*, *Compensating*, *Faulting*, *Faulting/Canceling* or *Faulting/Compensating* depending on the previous state the participant visited. When the participant has reached the *Completed*, *Closing*, *Compensat-*

ing or *Faulting/Compensating* state, the result is *Completed* as the participant has not yet confirmed anything else. In all other states, the result is *Active*.

5.3. Commands sent to Participants

The set of available commands depends on the business activity's outcome assertion:

- *mixed* outcome allows for the independent management of the participants.
- *atomic* outcome mandates that all participants are directed to an equal target state: either they all close, or they all *Cancel* or *Compensate*. Participants who leave by sending *Exit* or *Fault* are not considered.[1]

The following sections describe the protocol commands available in each of the outcome models. Should the initiator request a message to be sent to a participant for whom the message is not appropriate in his current state (e.g. sending *Complete* to a non-*CoordinatorCompletion* protocol participant) the coordinator will silently skip this participant. All operations except `getCoordinationContextWithMatchcode` return an updated participant list.

The following commands are valid for all business activities:

- `listParticipants`: retrieve a list of a transaction's participants (see subsection 5.2).
- `completeParticipants`: ask the coordinator to send a *Complete* message to the specified participants. This command only applies to participants who registered for the *CoordinatorCompletion* protocol.
- `getCoordinationContextWithMatchcode`: retrieve a new CoordinationContext object a new participant can use for registration. The match code specified within the coordination context is used to identify the participant throughout the transaction. The coordinator will always include the match code when information about a context is given. Furthermore, the coordinator always expects the match code when referring to a particular participant. The coordinator must ensure that match codes are unique in their contexts and that each match code must exactly refer either to one participant or to a pending registration. In *atomic* outcome contexts, this method cannot be called after the initiator made its final decision.

[1] Although one could argue that *Fault* should generally drive compensation, handling the faulted participants as if they exited enables the initiator to invite other business partners who may replace the failed participant.

5.3.1. Commands Valid in Business Activities with Atomic Outcome. This protocol extension allows the initiator to decide on the transaction's final outcome and tell the coordinator to execute the final decision. The initiator may call only one of the following methods per transaction.

- `closeAllParticipants`: decide to commit all work. This command is only valid when each participant has reached either the *Completed*, *Exiting* or *Exited* state. The coordinator will tell all *Completed* participants to close.
- `cancelOrCompensateAllParticipants`: decide that all work shall be aborted. The coordinator will direct participants to either *Cancel* or *Compensate*, depending on their state. There is a small chance that a participant being in the *Completing* state is sent a *Cancel* command at the same time as he completes his work and reports *Completed*. According to the specification, the coordinator must discard his last message sent and accept the participant's message. To enforce the already made decision, the coordinator must immediately issue a *Compensate* command to this participant. Should the participant be unable to compensate and send *Fault*, the overall transaction result is undefined and requires human intervention.

Once the final decision is made, the coordinator will refuse to register new match codes and will also reject participants who try to register for an already open match code. Subsequent calls are rejected with an *Invalid_State_Fault*.

If the initiator invokes one of those methods while his preconditions are not met, the coordinator silently ignores the request and returns the current participant list.

5.3.2. Commands Valid in Business Activities with Mixed Outcome. The *mixed* outcome assertion allows for the independent management of a transaction's participants. The initiator is required to tell the coordinator what command to give to the participants at any protocol state where more than one command is appropriate or the timing of the message is important (e.g. *Complete*). The coordinator will handle simple request-response type message flows automatically, e.g. upon receipt of an *Exit* notification the coordinator automatically replies with *Exited*.

Each of the following commands takes a list of participant match codes as argument. The coordinator makes sure that the requested command is valid in the participant's current state and silently discards inappropriate commands. Once the coordinator has completed his task, he returns a list of the participants including their current states.

- `cancelParticipants`: tells the coordinator to send *Cancel* to the specified participants.
- `closeParticipants`: tells the coordinator to send *Close* to the specified participants.
- `compensateParticipants`: tells the coordinator to send *Compensate* to the specified participants.

5.4. Managing Subcontexts

A subcontext encapsulates a whole transaction scope and presents it as a single participant to its parent transaction. This allows for the separation and parallelization of tasks a business activity needs to perform without loosing transactional semantics or control over the progress the participants make.

The initiator of a subcontext requires business logic to translate commands from the supercontext into commands he may issue to his local participants. This specification proposes that the subcontext coordinator registers himself for the *CoordinatorCompletion* protocol[2] in the super context and forwards any commands received to the subcontext initiator via a pull based query protocol.

Upon receipt of a valid message from the supercontext, the subcoordinator performs the internal state transition associated with the command and reliably lists the message for the initiator to query:

- received *Complete*: store and transist to *Completing*
- received *Cancel*: store and transist to *Canceling*
- received *Close*: store and transist to *Closing*
- received *Compensate*: store and transist to *Compensating*
- received *Exited*: store and transist to *Ended*

The initiator may instruct the subcoordinator to send messages to the supercontext. Upon receipt of such a command, the subcoordinator makes sure it is allowed in the current state and performs the state transition associated with the outgoing message.

[2]The *CoordinatorCompletion* protocol is a superset of the *ParticipantCompletion* protocol, hence one registration is sufficient.

- report *Exit* and transist to *Exiting*
- report *Completed* and transist to *Completed*
- report *Compensated* and transist to *Compensated*
- report *Fault* and transist to *Faulting*

In *atomic* outcome subcontexts, the coordinator automatically handles the following events:

- when receiving *Close* from the supercontext, all local *Completed* participants are directed to close. When all of those participants reported *Closed*, also report *Closed* to the supercontext.
- when receiving *Compensate* from the supercontext, all local *Completed* participants are directed to compensate. When all participants reported *Compensated*, also report *Compensated* to the supercontext; if one or more participants fail, also report *Failed* to the supercontext.
- when receiving *Cancel* from the supercontext while in state *Active*, reject new participants and direct all existing participants to cancel. When all participants reported *Canceled*, also report *Canceled* to the supercontext.

In the following we consider special cases for cancelling and compensation in subcontexts. In the *Completing* state, the *WS-BusinessActivity* protocol allows both coordinator and participant to initiate a state transition: the coordinator may send *Cancel*, and the participant may send *Exit*, *Completed* or *Fault*. If both parties send contradicting messages at the same time, the coordinator must discard its own message and accept the participant's one.

If the participant reported *Completed*, it is required to run compensation, but - unlike cancellation - compensation is allowed to fail. This race condition allows a situation to occur where the coordinator (from its point of view) sent a timely *Cancel* command, but the participant completed anyways and later failed on compensation, leaving the requested work permanently done instead of canceled.

This is especially bad in a subcontext. Because when the coordinator of the subcontext is told to cancel and forwards this command to the local participants, some may successfully cancel while others already have completed. It cannot report *Completed* since some parts of its work have successfully been canceled, and on the other hand cannot report *Canceled* since some work has actually been performed. The subcoordinator must hence immediatly try to compensate the *Completed* participants, and if they all successfully *Compensated*, may report *Canceled* to the supercontext. If one single participant fails during compensation, the whole subcontext

must report *Fault* to the supercontext to further escalate the issue.

The *WS-BusinessActivity* specification does not specify how subcontexts should handle other situations. In order to keep the time window for such critical states at a minimum length, a subcontext should invite *ParticipantCompletion* protocol participants only after receiving the *Complete* command from its supercontext. This ensures that a *ParticipantCompletion* protocol participant cannot report *Completed* when the supercontext has not told the subcontext to *Complete*. Registering for the *ParticipantCompletion* protocol itself is not an option for the subcontext, since it must not report *Completed* before all required participants themselves reach *Completed*.

In most cases an application will not know the protocol a particular business partner will register for in advance. In order to be on the safe side, all participants should be enrolled only after receiving the *Complete* message from the supercontext.

This convention keeps the time window for a collision short, but does not avoid the problem at all. Removing the possibility to issue a *Cancel* command in *Completing* would be a way to permanently fix the specification and better reflect the actual semantics: in general, a *Completing* participant will already have some work done and hence cancelling requires some kind of compensation to be executed. With the *Cancel* command removed, the initiator could not tell a *Completing* participant to stop processing. Instead he has to wait until they finish and then issue *Compensate*.

5.5. A Sample *WS-BusinessActivity–Initiator* Transaction

Figure 4 illustrates a sample transaction with mixed outcome in which the initiator decides on the outcome. The initiator-object is instantiated by the application and creates a *CoordinationContext* for the transaction. After having registered for the WS-BA–I protocol, the initiator is able to send *CreateCoordinationContext* messages to the coordinator which contain a different match code for each participant. It is the applications task to forward the returned *CoordinationContext* within an application message to its business partners.

After receiving an application message containing a *CoordinationContext* the business partner may register for the transaction as a participant using the received *CoordinationContext*. As shown in Figure 4, the initiating application repeatedly sends *listParticipants* messages to the coordinator to learn how many participants are already enrolled and in which states they currently are.

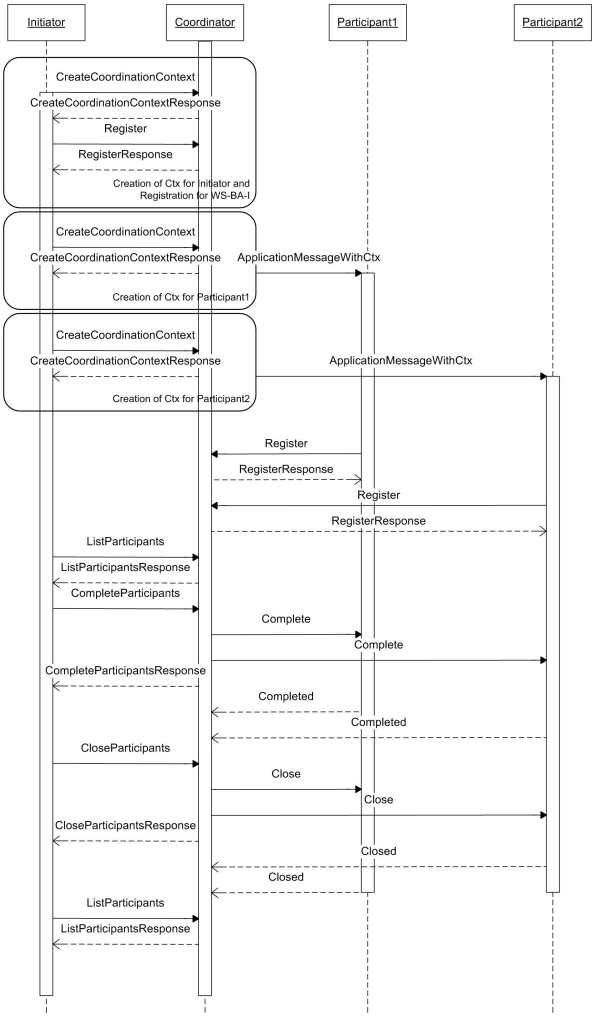

Figure 4. A sample *WS-BusinessActivity–Initiator* transaction

If the application decides to complete the transaction it tells the initiator to send a `completeParticipants` message to the coordinator, who in turn sends *Complete* messages to the specified participants. After having sent out the *Complete* messages the coordinator immediatly returns a *CompleteParticipantsResponse* to the initiator. It contains a *participantList* element that includes the updated participant states.

Since the coordinator does not actively inform the initiator about state changes, they must re-query current states by invoking the *listParticipants* method from time to time. If all participants are in the correct state they may call for closure of certain participants (in case of a transaction with *atomic* outcome this can only be done for all of them). This call behaves like the completion call: a response is generated immediately after all *Close*

messages were sent to the participants, and the initiator has to query for state changes. As soon as all participants have replied with *Closed* and, thus, reach the *Ended* state, all involved parties may forget about the transaction after the configured timeout.

6. Summary and Conclusion

Although *WS-BusinessActivity* was introduced in 2001, it is still expensive for software developers to make their products benefit from it. This is also due to the fact that in *WS-BusinessActivity* a workflow engine or extensive custom programming are needed to implement the coordinator role. The approach presented in this paper - which is also implemented as part of the Apache Kandula project [2] - significantly reduces the overhead involved when implementing WS-BA. Furthermore, the vendor-independent specification allows initiators to interoperate with any WS-BA–I compliant coordinator. It thereby creates an open environment.

With the clear distinction between business logic and transactional logic not only on the participant but also the initiator side, WS-BA–I vastly increases software modularity.

The clear separation of the initiator and coordinator roles enables initiators to select independent third parties who perform a transaction's coordination. We expect third-party coordination services to be offered and later to be certified. Thereby, both initiators and participants are assured of a correct coordination. Future extensions of *WS-BusinessActivity–Initiator* may offer improved reliability, security or performance.

References

[1] L. Cabrera, G. Copeland, W. Cox, T. Freund, J. Klein, D. Langworthy, I. Robinson, T. Storey, and S. Thatte, *Web Services Business Activity Framework (WS-BusinessActivity)*, version 2005-08 ed., Aug. 2005. [Online]. Available: ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf

[2] *Apache "Kandula"*, http://ws.apache.org/kandula, Jan. 2007.

[3] H. Erven and G. Hicker, *WSDL Description File for the WS-BusinessActivity–Initiator Protocol*, Feb. 2007. [Online]. Available: http://www.big.tuwien.ac.at/projects/WS-BAI/InitiatorProtocol.wsdl

[4] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., October 1993.

[5] OMG, "Object transaction service specification," in *CORBAservices: Common Object Services Specification*. Object Management Group, Sept. 2003, version 1.4. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2003-09-02

[6] L. Cabrera, G. Copeland, W. Cox, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, I. Robinson, T. Storey, and S. Thatte, *Web Services Atomic Transaction (WS-AtomicTransaction)*, version 2005-08 ed., Aug. 2005. [Online]. Available: ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf

[7] L. Cabrera, G. Copeland, W. Cox, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey, *Web Services Coordination (WS-Coordination)*, version 2005-08 ed., Aug. 2005. [Online]. Available: ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf

[8] M. Papazoglou, A. Tsalgatidou, and J. Yang, "The Role of eServices and Transactions for Integrated Value Chains," *Business to business electronic commerce*, pp. 207–241, 2003.

[9] J. Yang and M. Papazoglou, "Interoperation support for electronic business," *Commun. ACM*, vol. 43, no. 6, pp. 39–47, 2000.

[10] G. Weikum and H.-J. Schek, "Concepts and applications of multilevel transactions and open nested transactions," in *Database Transaction Models for Advanced Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 515–553.

[11] M. Papazoglou, "Web services and business transactions," *World Wide Web*, vol. 6, no. 1, pp. 49–91, Mar. 2003. [Online]. Available: http://www.springerlink.com/content/h4635u5r9m67t305

[12] *Business Process Execution Language for Web Services*, Version 1.1 ed., BEA, IBM, Microsoft, SAP AG and Siebel Systems, May 2003. [Online]. Available: ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[13] *Business Transaction Protocol*, Version 1.0 committee specification ed., Organization for the Advancement of Structured Information Systems (OASIS), June 2002.

[14] *UN/CEFACT - ebXML Business Process Specification Schema*, Version 1.11 ed., UN/CEFACT TMG, 2003.

[15] *RosettaNet Implementation Framework: Core Specification*, Version 02.00.01 ed., RosettaNet, December 2002. [Online]. Available: http://www.rosettanet.org/rnif

[16] *ebXML - Technical Architecture Specification*, Version 1.4 ed., OASIS, UN/CEFACT, Feb. 2001. [Online]. Available: http://www.ebXML.org/specs/ebTA.pdf

[17] F. Vogt, S. Zambrovski, B. Gruschko, P. Furniss, and A. Green, "Implementing web service protocols in soa: Ws-coordination and ws-businessactivity," in *CECW '05: Proceedings of the Seventh IEEE International Conference on E-Commerce Technology Workshops*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 21–28.

[18] P. Sauter and I. Melzer, "A comparison of ws-businessactivity and bpel4ws long-running transaction," in *Kommunikation in Verteilten Systemen (KiVS)*, ser. Informatik Aktuell. Springer, 2005, pp. 115–125.