Transactions with Unknown Duration for Web Services

Patrick Sauter
University of Ulm
ps9@informatik.uni-ulm.de

Ingo Melzer
DaimlerChrysler AG
paper@ingo-melzer.de

Abstract: Since the convergence of transactional Web Services and workflow management, human interaction can be a determining factor for the length of a distributed business-to-business transaction. Such transactions of unknown duration (e.g. due to human interaction) can be modeled properly neither as a short-running WS-AtomicTransaction nor as a long-running WS-BusinessActivity. Our proposal is to add the concept of ready-to-commit timeouts to the exclusive locking model of the WS-AtomicTransaction protocol by making a few minor extensions. The result is a simple and straightforward implementation strategy for transactions with unknown duration based on the Web Services Transaction Framework.

1 Introduction

Web Services were originally designed to simplify computer-to-computer communication. During the last few years, an extensive set of standards and specifications has emerged as modular building blocks for adding functionality to a Web Service. An important feature of Web Services in a business-to-business (B2B) environment is transaction support, because failed agreement on a transaction outcome (e.g. overbooked airline seats or a greater number of accepted orders for Christmas trees than there are in stock) is a highly undesirable scenario for any business.

A solution to this problem might be to directly implement every business transaction (e.g. book order) as a two-phase commit (2PC) ACID-compliant (Atomicity, Consistency, Isolation, and Durability [Gr81]) transaction, e.g. by means of the WS-AtomicTransaction [La03b] specification. In a distributed Web Services environment, however, this leads to extensive locking of resources which, in turn, is also undesirable, particularly if the locks are not released within a reasonable period of time. A solution which does not cause high lock rates might be to use so-called “long-running transactions”. For example, the WS-BusinessActivity [La04] specification uses compensating actions (i.e. explicitly coded business logic) instead of locking resources. As a result, a Web Services programmer must determine *at design-time* whether a service is short-running (and therefore has to be implemented using WS-AtomicTransaction) or long-running (and should be implemented using WS-BusinessActivity).

Since the publication of the Business Process Execution Language for Web Services (BPEL4WS, cf. [Th03]) specification, the Web Services technology has also gained ground in areas in which not only technical criteria such as processor and network speed determine the duration of an operation. Steps in a BPEL4WS workflow might in some cases have to wait for human input, e.g. approval by clicking an “OK” button or selecting an alternative from a drop-down menu. This might take far less than a minute, but maybe even several days, depending only on the user.

This is a rather special situation for transactions: Although the transaction could technically be finished within a few milliseconds, the commit message is postponed for an unknown timespan. Please notice that although it might be tempting to model the process as a (long-running) BusinessActivity, we will show in section 4.2 that this approach has several severe disadvantages and sometimes does not make sense.

Our proposition is that human interaction sets new requirements to transaction models for today's Web Services. Therefore, the contribution of this paper is

- the demonstration that both WS-AtomicTransaction and WS-BusinessActivity do not provide sufficient support for transactions with unknown duration and
- the recommendation of both a reasonable workaround and a more powerful but yet simple timeout-based solution on top of WS-AtomicTransaction.

2 Related Work

2.1 Approaches to Distributed Transactions in General

Transactions have long been considered mainly in the context of tightly coupled database systems. As enterprise information systems became more and more distributed, several new transaction models were suggested that were tailored towards the requirements of distributed, multi-participant transactions in loosely-coupled, failure-prone environments.

For loosely coupled systems (such as Service-Oriented Architectures (SOA), the concept behind Web Services), a general distinction has to be made between short-running and long-running transactions. Short-running transactions (such as WS-AtomicTransaction) are typically implemented using the two-phase commit (2PC) protocol. The implementation of long-running transactions is covered by a set of *advanced transaction models*. Among these are *open nested transactions*, *sagas*, and *chained transactions*. For a detailed discussion of the differences and applications of the various advanced transaction models for distributed long-running transactions, cf. e.g. [GR92, JG03]. The WS-BusinessActivity specification uses *open nested transactions* as the underlying transaction model. In short, an open nested transaction is a tree (of arbitrary height) of so-called “subtransactions”. The nodes in the tree may commit independently of each other without having to wait for the root transaction to commit (as this is the case for a *closed* nested transaction, which is more widely known) [GR92].

All of these advanced transaction models have the following properties in common:

- Relaxation of the isolation property: Intermediate results of a (sub)transaction might become visible to other participants before the overall transaction has committed.
- Relaxation of the atomicity property: In the event of failure, several (possibly atomic) (sub)transactions might have already committed and these intermediate results have to be compensated by invoking explicitly coded business logic.
- Moreover, they all lack the ability to exclusively lock a resource for only a limited period of time – an important requirement for transactions with unknown duration, as we will show in section 5.

An approach which tackles some of the issues related to transactions with unknown duration for Web Services is the Tentative Hold Protocol (THP, cf. [RS01]), a W3C Note by Intel. As the name already indicates, the THP does not lock resources exclusively, but only tentatively. A tentative lock may be cancelled at any time which causes the previous tentative lock holder to be informed of the cancellation. Since THP locks are not exclusive, the same issue arises as with no locking at all: What will happen if a user clicks the "OK" button to commit the order transaction of his shopping cart, but just a few milliseconds before, another participant has ordered the articles and thus none of them are left? This situation will be discussed in detail in section 5.

2.2 Existing Web Services Specifications for Distributed Transactions

The overall importance of the Web Services technology is continually increasing. In many fields, there are already several competing specifications for a single type of service, e.g. security, discovery, notifications, and also transactions. There are three important rival specifications covering the field of transactions for Web Services:

- the Web Services Composite Application Framework [Li03] by Sun, Oracle, Arjuna, Iona, and Fujitsu,
- the Business Transaction Protocol [Ce02] by OASIS, and
- the Web Services Transaction Framework (WSTF) by IBM, Microsoft, and BEA which consists of WS-Coordination [La03a], WS-AtomicTransaction [La03b], and WS-BusinessActivity [La04].

In this paper, we focus on the latter specification, because the WSTF is currently the most widely accepted approach with some early implementations already available (e.g. [http://www.alphaworks.ibm.com/tech/wsat/]). However, some of the particular features of WS-CAF and BTP are compared to those of the WSTF in section 5. For a detailed discussion of the WSTF, cf. [Cu03, Ca04]. In short,

- WS-Coordination defines the protocol for distributing the coordination context of a transaction (e.g. a unique transaction ID) to the participants. For example, WS-Coordination specifies the interface of a transaction manager (a so-called *coordinator*) for creating a new or joining an already existing transaction. Both WS-AtomicTransaction and WS-BusinessActivity are so-called *coordination types* that are built on top of WS-Coordination:

- A WS-AtomicTransaction is a short-lived (though not necessarily ACID) transaction implementing the two-phase commit protocol in terms of Web Services. Typically, it is used for locking resources exclusively and sending the `Rollback` notification in the event of failure.
- In contrast, a WS-BusinessActivity is a long-running transaction that may consist of several AtomicTransactions; it does not lock any resources itself (only the invoked AtomicTransactions hold locks for short periods of time). In the event of failure, it will invoke explicitly coded compensating actions.

As a result of this distinction, the implementor of a transaction-enabled Web Service has to decide at design-time whether the transaction is short-running (i.e. a WS-AtomicTransaction) or long-running (i.e. a WS-BusinessActivity) by answering the question: "Will the overall transaction always finish within a few milliseconds?" However, as already mentioned in the introduction, transactions can sometimes be of unknown duration. Figure 1 depicts the classification of WS-AtomicTransaction and WS-BusinessActivity regarding compensatability and transaction duration – with the latter including unknown duration. In this paper, we will focus on the last row, i.e. transactions with unknown duration.

	Compensatable		**non-compensatable**
	rollback directly implemented	**compensating actions** (i.e. explicit business logic)	
short-running	**WS-AtomicTransaction**	*not required; every short-running AtomicTransaction must be able to process the* `Rollback` *notification*	*not allowed; abort must be implemented by every AtomicTransaction*
long-running	*impossible; a Business-Activity may consist of several AtomicTransac-tions of which some might have already committed*	**WS-BusinessActivity**	**WS-BusinessActivity**; in case of error always send a `Fault` (i.e. "cannot compensate") notification
unknown duration (e.g. human approval)	discussed in section 5.1 and 5.2; *but not WS-Atomic-Transaction (as shown in section 4.1)*	discussed in section 5.1 and 5.2; *but not WS-BusinessActivity (as shown in section 4.2)*	*special case! – e.g. in case of human approval: not sensible; otherwise should be called "human notification"*

Figure 1: Classification of WS-AtomicTransaction and WS-BusinessActivity and their uncovered areas of transactions with unknown duration.

3 A Motivating Example

Web Services are to become the most important technology in business-to-business environments. In this paper, we will use the example of an online bookstore as a typical B2B service provider. The bookstore wants to offer its corporate customers a Web Services interface (cf. Figure 2). The usual book order transaction proceeds as follows: A back-office (e.g. workflow) system of the customer triggers the invocation of a book ordering routine. Since a book order is a transaction, the customer system first has to register (in Figure 2: step 1) with the bookstore's transaction manager (coordinator) using its `CreateCoordinationContext` interface as described in [La03a].

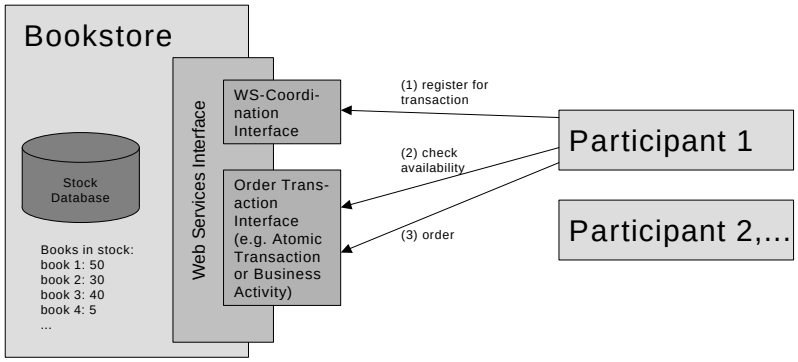

Figure 2 depicts the overall architecture of the bookstore Web Service.

The customer system is returned a `CoordinationContext` (e.g. including a transaction ID) and thus becomes a transaction participant. Next, the participant system checks the availability (step 2) of the articles it wants to order. If some of the articles are out of stock, a user has to be asked if the purchase should be carried out anyway. The missing articles have to be ordered at another bookstore or can be delivered only after a back order of the bookstore. The user will also be asked for approval if the total amount of the order (including the articles that were not available) is greater than, say, 20 €. Purchases with a total amount of less than 20 € are considered uncritical and are typical candidates for process automation and are carried out immediately. Either in this case or if the user approves the purchase, the transaction commits and the order is sent (step 3).

As a result, there are two factors determining the length of the book order transaction: The total amount of the order and the user. If the total amount of the order is less than 20 € and all books are available, the estimated duration might be, say, 100 milliseconds. If this is not the case, the time it takes until the user clicks the “OK” button on his worklist is the determining factor. This might be anything between a few seconds (if he is at his desk) and several days (if he is on vacation).

So, in this example, there is a high level of uncertainty about the transaction length and we therefore consider it well-suited. However, there are three noticeable assumptions:

1. Books of the same type are not distinguishable – unlike e.g. airplane seats.
2. Whenever a book is considered to be locked by the bookstore's Web Services interface (i.e. the application server), the book cannot be locked or ordered by any other of the bookstore's ordering systems, e.g. a JINI interface.
3. Human interaction is required only at a single step in the ordering process, so there is no complex chain of user inputs.

The main issue discussed in this paper is how to ensure that the system behaves correctly even if several participants are trying to buy certain quantities of the same book at the same time, e.g. when multiple participants are waiting for human approval. The decisive step for this situation is the time between checking the availability of a book and the time when the order is committed. Altogether, we will discuss four possible implementations of the bookstore example and their suitability for transactions with human interaction in the next two sections.

4 First Approach: Using the Web Services Transaction Framework without Adaptations

4.1 The Straightforward WS-AtomicTransaction Implementation

The WS-AtomicTransaction specification essentially describes the implementation of the two-phase commit protocol for Web Services. This typically implies that exclusive locking is involved between the completion of the prepare phase and the completion of the commit phase. Locking resources is questionable, because other participants are prevented from accessing the locked resources. Therefore, locks should be released after a very short period of time. Given these time constraints, it is not feasible to lock resources and then wait for the user. Instead, the user interaction must take place *before* the resources have been locked. Figure 3 shows the order in a workflow-style notation. Notice that the user input takes place before the actual AtomicTransaction starts.

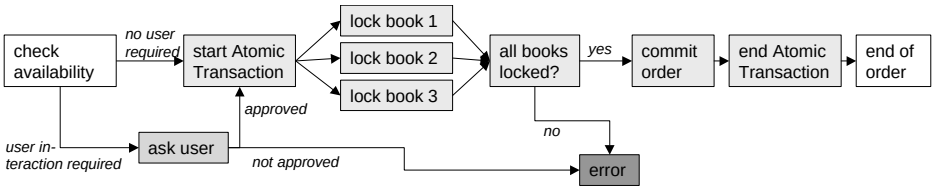

Figure 3 depicts what the workflow of the pure AtomicTransaction implementation of a book order might look like.

The main problem about this approach is the following: Assume that the outcome of the activity “check availability” is “no user required”, i.e. all books are available and the total amount of all books does not exceed 20 €. Although the AtomicTransaction starts almost immediately, another participant might have started an order of the remaining books in stock in the meantime. The participant system is now unable to order the books whose availability it has checked only a few milliseconds before. Eventually, an error occurs. To anyone willing to use the bookstore's Web Services interface, this behavior seems to be an obvious bug (cf. e.g. [Ne86, pg. 424]).

4.2 The Straightforward WS-BusinessActivity Implementation

BusinessActivities are long-running transactions that may consist of several short-running AtomicTransactions. A BusinessActivity itself cannot hold a lock on a resource, only its constituent AtomicTransactions may acquire exclusive locks on resources. The main difference in behavior between an AtomicTransaction and a BusinessActivity (apart from the duration) is its recovery concept. Whenever an AtomicTransaction fails, the `Rollback` notification is sent, i.e. the 2PC's commit phase is simply not completed, the uncommitted changes are discarded and all locks are released immediately. In contrast, when a BusinessActivity fails, there are no locks that could be released. Instead, all its AtomicTransactions which have already committed must be undone, i.e. compensated.

Since these AtomicTransactions have already reached the `Ended` state, explicitly coded business logic has to be called to reverse the effects of the already committed AtomicTransactions.

For the bookstore example, this implies the following implementation (cf. Figure 4): The book order BusinessActivity essentially consists of one AtomicTransaction which orders the available books without considering whether all of the books were available or the total amount exceeded 20 €. The user will be asked whether he is fine with the outcome only *after* the AtomicTransaction has committed. If he is not, the order has to be cancelled or, in case the books have already arrived after a long approval time, the parcel has to be sent back as the compensating action.

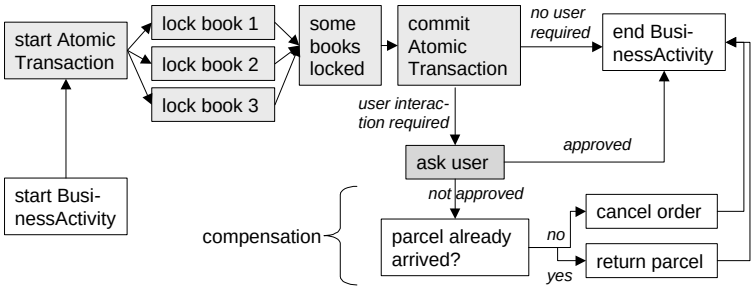

Figure 4 shows the pure BusinessActivity implementation with compensating actions.

In contrast to the pure AtomicTransaction approach of section 4.1, the human interaction takes place *after* the availability has been checked and the books have been locked. The user can be sure that no error will occur because of other participants snatching away his books. However, the greater usability of this optimistic approach is outweighed by its high shipping charges: Simply ordering articles and then returning them in case the user disapproves can be very expensive and is not feasible for most B2B scenarios.

In other words, the initially most obvious approach of modeling a transaction with *unknown* duration simply as a *long-running* transaction such as a BusinessActivity does not always make sense. Moreover, in this particular example, sending the books to any prospective purchaser in advance makes these already dispatched books unavailable for other participants. This is essentially the same as if an AtomicTransaction acquires a long-running lock on the books. So, the pure BusinessActivity approach is undesirable in many respects and not suitable for transactions with unknown duration such as the bookstore example.

5 Second Approach: Ready-to-Commit Timeouts

Since the two previously described approaches both have severe disadvantages that make them unsuitable for most B2B scenarios with human approval, we will now discuss the requirements for the desirable solution that does not face the problems encountered in section 4.1 and 4.2.

On the one hand, the disadvantages of the pure AtomicTransaction approach of section 4.1 have shown that locks should be acquired *before* the user is asked. On the other hand, the pure BusinessActivity approach of section 4.2 is equivalent to locking resources for a very long period of time which isn't desirable either. Our proposition is that locking resources is necessary, but because of the associated disadvantages, locks should be granted only for a limited period of time.

This section discusses the possible implementation strategies for the bookstore example with *ready-to-commit timeouts* – as opposed to an `Expires` timeout (cf. section 5.2). A ready-to-commit timeout refers to limiting the timespan between the prepare phase and the commit phase of the 2PC protocol, i.e. the usually very short moment in which the resources are locked. This timeout might now be granted to a participant for, say, 60 seconds. If no user interaction is required, the system will certainly be able to finish the transaction within less than one second and therefore the minute-long timeout is meaningless – the transaction commits and the books are ordered. Similarly, if user interaction is required, the user is assured that no other participant can snatch away his books within this period of time – he may safely complete the order. In case that either the ready-to-commit timeout of 60 seconds expires or the user votes to disapprove the order, the locks are released, and other participants may order these books.

To minimize the number of error messages displayed to a user, it might be one possible strategy to deal with a late order approval after 61 seconds (i.e. the user clicking the "OK" button after the ready-to-commit timeout has expired and the locks have already been released) as follows: All information on the order is now available and thus no additional user interaction is required if still all of the books are available. It might then be feasible to simply try to order the books with a single AtomicTransaction. Only if some of the previously locked books are not available any more (or the price has risen), an error message has to be displayed. This error message could also contain information on the cause of the error, e.g. "Sorry, your timeout has expired and your order cannot be repeated at present. Please try again." The user would probably not consider this message to be a bug.

Notice, however, that more complex situations in which some previously unavailable articles are now available or the question of what will happen if the user wants to make changes to the order are not discussed exhaustively in this paper. We will also neglect the possibility to keep the lock on an article *even after* the timeout period has expired as long as no other participant requests some of these locked articles.

The most important advantages of timeouts for this scenario are the following:

- If we assume that most transactions which require human interaction will be approved or disapproved within 60 seconds, no validating check whether the desired quantity is still available is needed, thereby minimizing overhead.
- Using timeouts makes the order process starvation-free, because no participant may hold a lock forever.
- Moreover, because locks can be granted exclusively, the user is given a guarantee that no unexpected failure condition may appear due to the unavailability of a book before the timeout has expired.

Similar to the WSTF, the Web Services Composite Application Framework (WS-CAF) does not contain a dedicated timeout mechanism for limiting the duration of the `Prepared` (ready-to-commit) phase. The Business Transaction Protocol (BTP), however, features an `<inferior-timeout>` tag that specifies the number of seconds a participant is guaranteed to remain in the `Prepared` state as long as no `Commit` notification arrives. After the timeout has expired, the BTP participant is allowed (though not required) to finish the transaction by invoking either `Commit` or `Cancel` depending on the Boolean value contained in the `<default-is-cancel>` tag. That is, the bookstore's basic desired behavior could well be implemented on the basis of BTP by setting `<default-is-cancel>` to `true`. However, we will continue to focus on the possible implementation strategies for the described behavior by means of the WSTF only. Therefore, the main question now is: How can we ensure that the WSTF transaction is rolled back (and the locks are released) after the timeout has expired?

5.1 AtomicTransaction with a Workaround

The 2PC protocol requires all its participants to agree on the transaction outcome. If at least one participant does not agree to commit and instead votes to roll back the transaction, the entire transaction will be aborted and all the other participants must roll back their uncommitted work. This leads to the following feasible implementation of the desired behavior: During the execution of the prepare phase in which the books are locked, another activity is invoked. In contrast to the other prepare phase activities, this "dummy" activity does not acquire any locks, but its only purpose is to wait for the timeout to expire and then send the `Rollback` notification (cf. Figure 5).

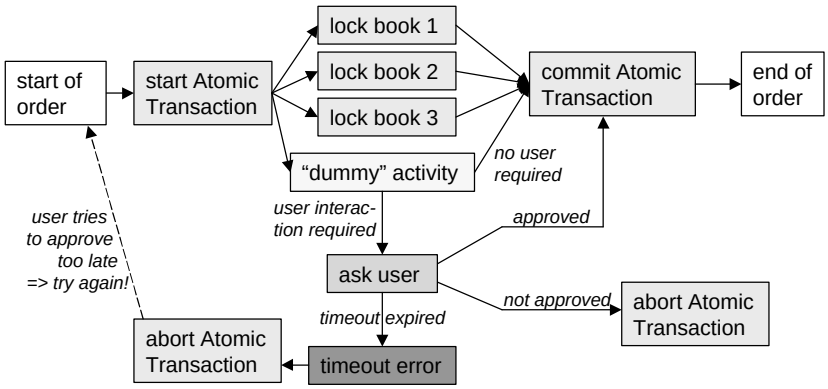

Figure 5 depicts the order process with a "dummy" activity that automatically rolls back the transaction after the timeout has expired.

Notice that if the user has tried to approve the order after the timeout has expired, e.g. after 61 seconds, the "start AtomicTransaction" activity is reached for the second time. In this situation, the condition "user interaction required" holds false if nothing about price and availability of the books has changed from the first time this step was reached.

Although this implementation clearly provides the required behavior, there is some implementation overhead because of the dummy activity that has to be implemented for every AtomicTransaction with human approval. As we have argued, human interaction will become more and more common, and it would thus be better if the timeout mechanism was already implemented by the bookstore's transaction manager. This will be discussed in the subsequent section.

5.2 The Desirable Architecture

Timeouts are a relatively common mechanism in distributed environments. It is therefore not surprising that the WS-Coordination specification [La03a] already contains a timeout mechanism: the optional `Expires` attribute whose semantics can be overridden by every new coordination protocol. Both WS-AtomicTransaction and WS-BusinessActivity actually override the semantics of this attribute. Quoting the WS-AtomicTransaction specification [La03b]: “This attribute specifies the earliest point in time at which a transaction may be terminated solely due to its length of operation. From that point forward, the transaction manager may elect to unilaterally roll back the transaction, so long as it has not transmitted a `Commit` or a `Prepared` notification.”

Notice that a transaction manager might be either the participant's coordinator or the bookstore's coordinator. In the context of the bookstore example, the bookstore coordinator manages all timeouts. So, the value passed in the `Expires` attribute of the `CoordinationContext` specifies the (earliest) point in time at which the bookstore coordinator is allowed to roll back the transaction as long as it has not yet sent the `Commit` notification.

The first idea would be to simply let the `Expires` attribute specify the ready-to-commit timeout. The bookstore's coordinator could be configured to make use of its right to unilaterally roll back the transaction after the timeout has expired, which would be an operable solution to the desired behavior as described in section 5. However, there are several problems related to the `Expires` attribute:

- As its name already indicates, the `Expires` attribute contains an absolute time value, e.g. “2004-10-11T18:30:00.000+01:00”. If the inherent assumption that the coordinator's and participants' clocks are reasonably synchronous does not hold, this creates severe difficulties for relatively short timeouts. For example, if the granted timeout is 10 seconds and the time difference between the two clocks is 11 seconds, this would render any book order impossible.
- The `Expires` attribute is determined before the transaction starts. As a result, the bookstore would not be able to grant a shorter lock on a book that is in high demand and in low stock than on a book still available in abundance.

As a result, we cannot build a good solution to the bookstore Web Service based on the `Expires` attribute. Instead, our proposed solution is based on the following extensions to the WS-AtomicTransaction specification:

- The length of the ready-to-commit timeout is represented in the form of an offset value, e.g. the number of milliseconds, and is determined by the coordinator only after all participants have sent the `Prepared` message, e.g. after all books have been locked. The information on the timeout length then has to be conveyed to the ordering participant, e.g. the user.
- If the ready-to-commit timeout expires without any human interaction, the system changes its state to a dedicated error condition, e.g. `TimeoutExpired`, which allows for a backward jump to the initial `Active` state without forgetting the transaction context.

These two extensions would substantially decrease the effort of implementing a Web Service with unknown duration, e.g. due to human interaction, as shown in Figure 6.

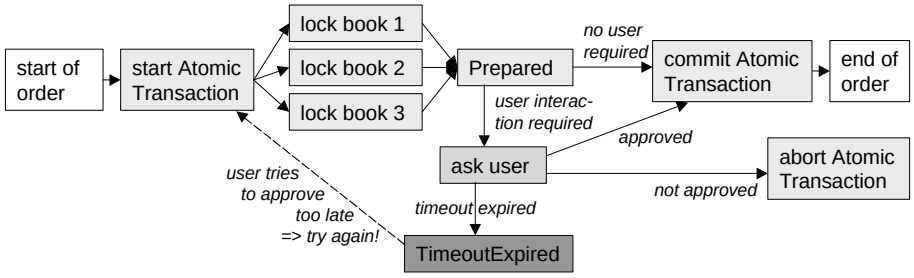

Figure 6 depicts the process of the typical book order transaction based on our proposed extensions to WS-AtomicTransaction.

The result of our two minor extensions to the WS-AtomicTransaction specification is a simple and intuitive implementation strategy to the bookstore example. As already stated, since Web Services increasingly tend to be used as part of BPEL4WS workflows which often require human interaction, this scenario will become more common and a built-in timeout mechanism is required to deal with B2B transactions efficiently.

6 Conclusions

As more and more Web Services are designed not only for computer-to-computer communication but also for workflow management systems with human interaction, transactions with unknown duration will play an increasingly important role. We have demonstrated that transactions with unknown duration cannot be implemented as either short-running AtomicTransactions or as long-running BusinessActivities without running into serious difficulties. Instead, a timeout mechanism is required for the time a resource is exclusively locked. We therefore proposed two extensions to the WS-AtomicTransaction specification: a ready-to-commit timeout duration value that has to be determined by the coordinator after the 2PC's prepare phase has finished as well as a dedicated `TimeoutExpired` condition which allows for a backward jump to retry the transaction in case of a late (order) approval.

The result is the implementation strategy of section 5.2 (cf. Figure 6) in which the entire timeout management has been delegated to the bookstore coordinator. Future work will be conducted in the field of transactions requiring multiple human interaction steps, i.e. timeout-based transactional workflows that generalize the assumptions of the bookstore example.

Acknowledgement

This paper was written as part of a Web Services research project at DaimlerChrysler Research and Technology in Ulm, Germany and a diploma thesis at the Department of Applied Information Processing (SAI) of Prof. Schweiggert at the University of Ulm.

References

[Ca04] L. F. Cabrera, G. Copeland, J. Johnson, D. Langworthy. Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity. January 2004. http://msdn.microsoft.com/library/en-us/dnwebsrv/html/wsacoord.asp

[Ce02] A. Ceponkus et al. Business Transaction Protocol. BTP Committee specification. April 2002. http://www.oasis-open.org/committees/business-transactions/

[Cu03] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, S. Weerawarana. The Next Step in Web Services. Communications of the ACM. Communications of the ACM. Volume 46. Issue 10. October 2003.

[Gr81] J. Gray. The Transaction Concept: Virtues and Limitations. In Proceedings of the 7th International Conference on Very Large Data Bases. Pages 144-154. September 1981.

[GR92] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Series in Data Management Systems. 1992.

[JG03] T. Jin, S. Gosschnick. Utilizing Web Services in an Agent Based Transaction Model (ABT). International Workshop on Web Services and Agent-based Engineering (WSABE-2003), held in conjunction with AAMAS-2003. Pages 1-9. July 2003.

[La03a] D. Langworthy et al. WS-Coordination specification. September 2003. http://www-106.ibm.com/developerworks/library/ws-coor/

[La03b] D. Langworthy et al. WS-AtomicTransaction specification. September 2003. http://www-106.ibm.com/developerworks/library/ws-atomtran/

[La04] D. Langworthy et al. WS-BusinessActivity specification. January 2004. http://www-106.ibm.com/developerworks/library/ws-busact/

[Li03] M. Little et al. Web Services Composite Application Framework (WS-CAF). Version 1.0. July 2003. http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf

[Ne86] P. E. O'Neil. The Escrow Transactional Method. ACM Transactions on Database Systems (TODS). Volume 11, Issue 4. Pages 405-430. December 1986.

[RS01] J. Roberts, K. Srinivasan. Tentative Hold Protocol Part 1: White Paper. W3C Note. November 2001. http://www.w3.org/TR/tenthold-1/

[Th03] S. Thatte et al. Business Process Execution Language for Web Services. Version 1.1. May 2003. http://www.ibm.com/developerworks/library/ws-bpel/