# Nonblocking Scheduling for Web Service Transactions

Mohammad Alrifai[1] Wolf-Tilo Balke[1] Peter Dolog[2] Wolfgang Nejdl[1]
[1] *L3S Research Center*
*University of Hannover, Germany*
*{alrifai, balke, nejdl}@l3s.de*
[2] *Department of Computer Science*
*Aalborg University, Denmark*
*dolog@cs.aau.dk*

## Abstract

*For improved flexibility and concurrent usage existing transaction management models for Web services relax the isolation property of Web service-based transactions. Correctness of the concurrent execution then has to be ensured by commit order-preserving transaction schedulers. However, local schedulers of service providers typically do take into account neither time constraints for committing the whole transaction, nor the individual services' constraints when scheduling decisions are made. This often leads to an unnecessary blocking of transactions by (possibly long-running) others. In this paper, we propose a novel nonblocking scheduling mechanism that is used prior to the actual service invocations. Its aim is to reach an agreement between the client and all participating providers on what transaction processing times have to be expected, accepted, and guaranteed. This enables service consumers to find a set of best suited providers fitting their deadlines. Service providers on the other hand can benefit from the proposed mechanism due to the now possible intelligent scheduling of service invocations for best throughput. In fact, our experiments show a significant improvement in terms of overall throughput, service chain completions and resources' utilization.*

## 1. Introduction

Web service-based business processes are generally composed of invocations to internal business processes outsourced by loosely coupled providers. A key requirement for the success of Web service-based integration of business applications is to ensure a correct and reliable execution of the composed process with regards to partners' transactional requirements. Web services technology provides standard interfaces like WSDL and XML SOAP messaging for the integration of enterprise applications, however, do not fully support transactional management yet. Web service providers have to control their local resources and ensure their consistency and integrity. Web service requests are treated by local transaction managers as independent transactions and strict ACID properties are enforced by special error recovery and concurrency control components.

In contrast to classical database systems, however, a Web service-based transaction is generally long-running and involves some independent services that might need to be synchronized manually. To optimize their throughput (and thus their profits) service providers, therefore, cannot grant locks on their resources for some consumers for long time spans. Hence, strict locking-based distributed transaction protocols are difficult to apply in Web service environment.

More optimistic transaction models with *relaxed isolation* property have been recently adopted for Web service transactions where the intermediate results of previous activities are already made visible to the new consumers, even if they are not yet committed ('dirty read'). Providers leverage compensation mechanisms to recover from any failures or cancellations of executed operations (see e.g. [9, 15, 17, 18] and [4, 5]).

Ensuring the correctness of the execution of concurrent transactions under relaxed isolation is left to the local transaction managers of the service providers. However, conventional scheduling algorithms typically do not take into account the timing characteristics of transaction execution and individual time constraints are ignored when scheduling decisions are made (i.e. which transaction is allowed to commit and which transactions have to be delayed). This often leads to blocking transactions later by earlier transactions that still not committed, though they could already be willing to commit their individual execution. This has a great impact on Web service-based business transactions as those are usually combined with time constraints. For example, consider a Web service-based process for arranging a business trip to attend some meeting on a specific day. The time factor in such transactions is very valuable and the initiators of these transactions usually have a deadline for committing these transactions and might not be interested in a successful termination after this deadline.

In this paper, we propose a nonblocking scheduling mechanism for enhancing the transaction management of Web services to cope with these limitations. Our approach takes into account the deadline constraints of both the consumers and the providers of Web services while scheduling the execution of a global transaction at every participating site. The proposed solution can be applied in a fully decentralized fashion for open dynamic

environments where global transaction manager does not exist. In particular, our contributions are the following:

- Minimizing the number of aborted transactions due to missed deadlines (advantage for consumers)
- Maximizing resource utilization, i.e. the service throughput (advantage for providers)

This paper is structured as follows: Section 2 describes a motivating scenario and in Section 3 a formal description of the problem is introduced and related work is reviewed. Section 4 describes and discusses our approach for addressing the problem in details. In Section 5 we describe our implementation and the experimental results that show the significant improvement in terms of service chain completions, throughput, and resources utilization. Section 6 concludes our paper and gives an outlook on future work.

## 2. Motivating Scenario

The following scenario (Figure 1) shows a completion dependency between two concurrent business transactions that leads to a blocking situation. An airline company offers flight tickets to a travel agency, who in turn sells them to customers. Assume the agency requests 10 seats for a particular flight by invoking the reservation service. In the meantime to increase the utilization of its resources, the travel agency already makes the tickets available for customers instead of safely locking them until a commit is issues by the airline. A customer may now already request a ticket through this travel agency. Upon receiving the request, the scheduler of the local transaction management and based on the current status of concurrent active transactions decides upon the right scheduling of the request such that the consistency of all transactions is maintained. In our scenario the 10 tickets may be available though not yet committed, which means that the customer's transaction has to be serialized after the airline's transaction, i.e. before the customer's request can be committed the airline has to commit the transaction first. The agency thus reserves the ticket for the customer on an optimistic base. The customer transaction now might also involve some other activities served by other providers like reserving a hotel room, registering for a conference, thus he/she proceeds in invoking all corresponding web services. At the end of the process the transaction coordinator tries to commit all activities and starts to run a 2PC protocol to insure that really all participants are ready to commit. This is important, because if some individual activity like the flight booking fails, also other activities like the hotel reservation might be affected. The coordinator therefore sends a "ready to commit" message to Web services all providers including the travel agency's Web service. However, in case the offer from the airline company is still not committed, the travel agency cannot immediately issue such a message. Aborting the transaction at this late stage is too expensive, therefore, the customer request has to be delayed until the final confirmation from the airplane company is received.

By preserving the order of completion, consistent outcome of concurrent transactions can be achieved. However, the amount of time the customer's commit request has to be delayed is unpredictable and strongly depends on the dominant transaction(s). This can be seen as a live lock, because at the end the whole transaction of the client is blocked. Assuming that the customer has a deadline for committing the transaction (the date of the trip is too close or the deadline for confirming the Hotel

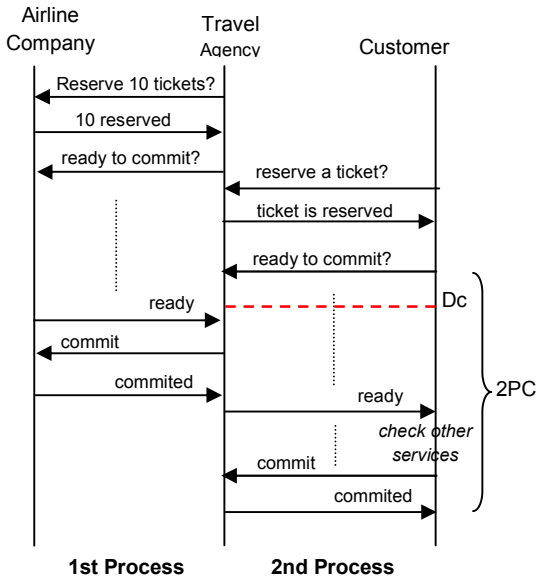


**Figure 1. Two inter-dependent business processes**

booking is approaching), the client would not be interested in a late successful completion after the deadline is missed (see the dashed line in Figure 1).

The duration of the 2PC protocol (or any consensus-based commitment protocol) is strongly influenced by the underlying concurrency control mechanism, which is applied by the transaction managers of the service providers. In this paper, we propose an approach for intelligently scheduling concurrent transaction requests prior the actual execution to minimize the waiting times during the execution of the 2PC protocol and thus avoiding transaction blocking.

## 3. Background and Related Work

A web services-based transaction T is a partially ordered set of n tasks $T = \{t_1, \ldots, t_n\}$. To execute T, a concrete Web service implementation $WS_i$ for every abstract service $t_i$ in T is needed. As a result of relaxing the isolation property completion dependencies among

transactions in the local schedule of every provider occur dynamically. There is a completion dependency between two transactions $T_1$ and $T_2$ if $WS_1$ and $WS_2$ invoked by $T_1$ and $T_2$ respectively, are semantically in conflict, i.e. access the same resource held by a common provider and at least one of them is influencing the result of the other (for example by writing a data item used to compute the output of the other one).

To maintain transactional consistency of concurrent executions, local transaction schedulers of service providers use special concurrency control mechanisms to ensure the serializability of the produced schedules. Serializability with ordered termination has been identified as the correctness criterion for the applied concurrency control mechanism that counts for addressing both atomicity and isolation [14]. Local schedules, therefore, preserve the order of transactions commitments according to the completion dependencies. By $Q_i$ we denote the ordered list of transactions that have an access to the *i*th resource. The first transaction in $Q_i$ is the only one that has an exclusive lock on the resource while all other post-ordered transactions in the queue have a shared lock. Any transaction T that has an exclusive lock is allowed to commit immediately, i.e. if provider receives a ready-to-commit request from T, it replies immediately with a ready message. The commitment of Transactions holding shared locks has to be delayed until exclusive locks are acquired.

## 3.1 Waiting Times during 2PC

Adopting the concept of shared locks under relaxed isolation by optimistic transaction schedulers allows providers to increase the throughput of their Web services. However, a main problem that remains is that transactions still need to acquire the exclusive locks of all accessed resources before being able to commit. And the fact that Web service-based transactions are usually long-running and completion dependencies among transactions occur dynamically renders the problem even more challenging. The example in Figure 2 shows how the required time for acquiring exclusive locks is crucial to the length of the 2-phase-commit protocol execution. In the first phase the transaction coordinator sends a call for commit to all participants and waits until it receives a positive (ready) or negative (failed) answer on his request. The length of the waiting time (Wc in Figure 2) depends on the pre-ordered transactions that have to commit before this client can commit. This is the time necessary for acquiring an exclusive lock on the requested resource. After granting the exclusive lock, providers usually have to wait for getting the final decision (commit or compensate) from the coordinator in the second phase. This waiting time (Wp in Figure 2) can also be long since the coordinator cannot start the second phase before it receives the answer from all providers (i.e. all exclusive locks are acquired).

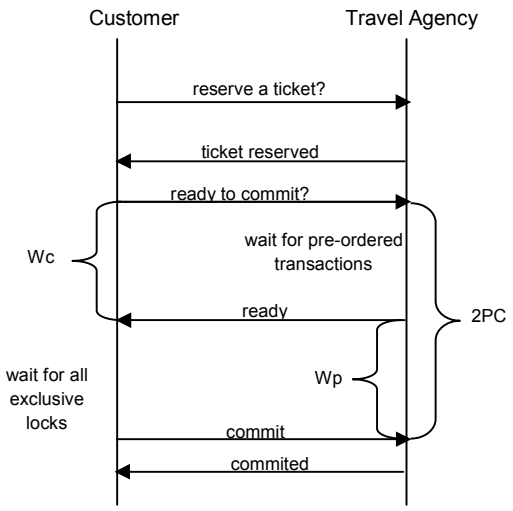


**Figure 2. Waiting times on both sides during the execution of the 2PC protocol**

## 3.2 Blocking vs. Nonblocking Schedules

Given that business transactions are usually combined with time constraints (e.g. customer's deadline for committing the transaction or provider's deadline for releasing the exclusive lock on local resource), long waiting times during the execution of a 2PC protocol increases the probability of violating these constraints, hence, aborting the already successfully executed transaction by any partner because of missed deadlines.

Figure 3 illustrates a transaction with three Web services in three different possible schedules based on the offered spans for holding exclusive locks by service providers. In Figure 3.a we see that the offered spans (i.e. time span between lock acquisition L and lock release R) do not violate the client's deadline (Dc) for committing the whole transaction. Furthermore, all offered spans are overlapping, i.e. locks can be acquired (and released) before the earliest provider's deadline is missed. Consequently, such schedule ensures that time constraints of all participants in this transaction are met and the transaction is not going to be blocked at commit time. In Figures 3.b there is a conflict between the client's deadline and the expected time for acquiring the exclusive lock for $WS_2$ ($L_2$), which means that the transaction is going to be blocked until the deadline is missed. In Figure 3.c all offered spans respect the client's deadline but there is a conflict between the deadlines of the providers of $WS_1$ and $WS_3$ ($L_3 > R_1$), which leads to blocking the provider of $WS_1$ until the lock of $WS_3$ is acquired and the coordinator is able to commit.

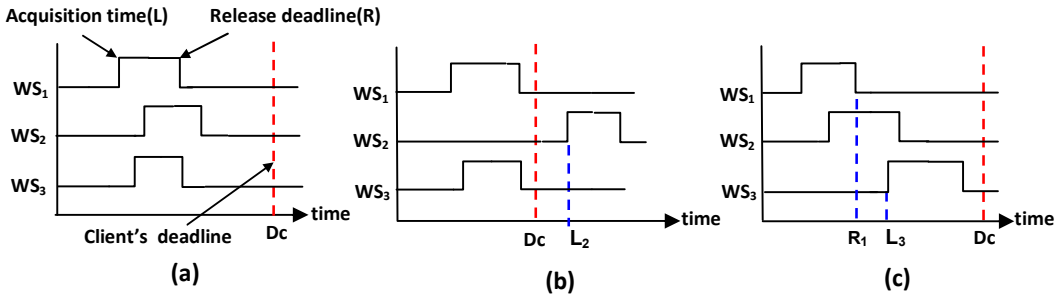


**Figure 3. Different synchronizations of locks acquisition for a transaction with three Web services:** a) conflict-free, b) conflict with client's deadline and c) conflict between deadlines of $WS_1$ and $WS_2$

We consider any global schedule that leads to violating time constraints of transaction partners (like in Figures 3.b and 3.c) as a blocking schedule. In the following we give a formal definition of blocking and nonblocking global schedules (we refer the reader to [24] for formal definitions of transaction schedules).

**Definition 1**: (Global Schedule)
Let $G$ be a set of global transactions and $T^1$, …, $T^n$ be sets of local subtransactions at provider sites $P_1$, …, $P_n$. Let $S_1$, …, $S_n$ be local schedules such that $T^i \subseteq S_i$ and $G \cap S_i \neq 0$ for $1 \leq i \leq n$. A global schedule is a schedule $S$ for $G$ such that its local projection equals the local schedule in each provider site, i.e., $\prod_i(S) = S_i$ for all $i$, $1 \leq i \leq n$.

**Definition 2**: (Blocking vs. Nonblocking Schedule)
Let $S$ be a global schedule, $T$ be a transaction composed of $m$ Web services and $Dc$ be the deadline for commitment. Let $L_i$ be the expected time for acquiring the exclusive lock and $R_i$ the deadline for releasing the lock according to the projection of $S$ at local sites $\prod_i(S)$ for all $i$, $1 \leq i \leq m$. A global schedule $S$ is a nonblocking schedule iff: $L_i < Dc \wedge R_i > L_j$ for all $i, j$, $1 \leq i, j \leq m$, and a blocking schedule otherwise.

Our contribution in this paper is proposing a scheduling mechanism for long-running global Web service transactions, which produces a nonblocking global schedule in the sense of the given definitions, i.e. for every global transaction $T \in G$, lock holding spans:

- do not violate the client's deadline, and
- do not mutually conflict

### 3.3 Related Work

Two different open specifications have been introduced to handle ACID properties of web service transactions: Web service Atomic Transaction [6] for short transactions that require strict ACID properties and Web service Business Activity [5] for long-running flexible transactions that require relaxed atomicity and isolation properties. The latter relies on the notion of compensations [9, 15, 17, and 18] which are triggered whenever a subset of a transaction fails. Compensations are introduced either at the client level as part of the business process execution [10] or on both, client and participant sides [11]. The compensation approach is very flexible and reduces especially long waiting times when long running transactions are locked and wait for each other. On the other hand, if the environment triggers too many compensations, cascading compensations may result in long and costly replacements of concrete Web services, making Web service transactions too long.

Ensuring a fault-tolerant execution of Web service transactions and reliable composition of transactional composite Web services has been the focus of recent research work (e.g. [8], [12], [16] and [19]). However, maintaining consistency of the concurrent transactions is neither addressed by these papers nor by the existing industrial specifications. Several distributed transaction protocols for concurrency control of Web services have been recently proposed (e.g. 0, 0, or 0). The proposed solutions ensure global consistency by ensuring the serializability of local schedules. Serializability with ordered termination has been identified as the correctness criterion for the applied concurrency control mechanism that counts for addressing both atomicity and isolation [14]. However, conventional scheduling algorithms typically do not take into account the timing deadlines for transaction execution of service consumers nor of service providers. Consequently, post-ordered transactions in the transactions schedule are blocked at commit time (for example during the execution of a 2PC protocol) by their pre-ordered ones. Given that Web-services-based business transactions are usually long-running process, blocked transactions which have some deadlines (e.g. the date of the business trip) are likely to abort because of the missed deadlines. The higher the degree of concurrency in the

system, the higher the number of aborted transactions because of missed deadlines. The problem becomes even worse if the aborted transaction has post-ordered transactions in the schedule, which leads to cascading aborts.

In this paper we introduce an enhancement to the transaction management for Web services to cope with these limitations. The proposed scheduling mechanism takes into account the time characteristics of transactions to produce a non-blocking global schedule. It can be executed in cooperation between transaction coordinators and local transaction schedulers of service providers to ensure global consistency and correctness of the produced schedule. The goal of our approach is similar to the goal of the research work introduced in the area of real-time databases [e.g. 20, 21 and 22]. However, the introduced solutions for real-time databases still tend to solve conflicts by aborting or delaying some transactions on the favor of other transactions in a priority-based manner. Our approach on the other hand avoids such conflicts by producing a conflict free schedule that meets all partners' deadlines prior the actual execution to better suit the Web service environment. We will also show that our approach features improved throughput independently on the level of concurrency and execution time of Web services.

## 4. Nonblocking Transaction Scheduling

Figure 4 gives an overview of our approach. The Web service-based transactional process is composed at design time by specifying abstract services and predefined deadlines. The scheduling can be then conducted by the transaction manager of the client's application i.e. the transaction Coordinator in the terminology of [23]. The Coordinator takes the abstract transactional process as input and produces as output a concrete execution plan (i.e. a concrete service implementation for each abstract service) that does not suffer the problem of long waiting times (i.e. Wc and Wp). All participants in the produced plan agree on a common and acceptable time span for holding exclusive locks during the execution the commit protocol (e.g. 2PC protocol), hence, the output of this scheduling process is always a non-blocking schedule.

### 4.1 The Transaction Scheduling Algorithm

In the following we sketch the overall algorithm to show how the Coordinator can communicate with the candidate providers to agree in advance on an acceptable timing for the acquisition and release of exclusive locks. We assume that service providers advertise information about the expected processing time (e.g. average or maximum execution time based on statistical information) or provide methods to enquire about it and that a discovery component exists and is able to find *equivalent* Web service implementations (see e.g. [7]).

***Transaction Scheduling Algorithm:***

**Input:**

1. $T = \{t_1, t_2, \ldots, t_n\}$, a transaction composing $n$ tasks
2. $Dc$ = client's deadline for transaction commit

**Start:**

1. Coordinator collects *offline* information about service implementations (e.g. from UDDI) including maximum expected execution time $E$ for each task t from $T$

2. Coordinator requests *online* offers from local schedulers of service providers, i.e.:
   $L$ = expected time for acquiring exclusive lock
   $R$ = provider's deadline for releasing the lock

3. Providers: based on the position $k$ of $T$ in the local schedule $Q$ (refer to Section 3 for the definition of $Q$), every provider prepares its offer as follows:
   $L_k = E + 0$ , if $k = 0$
   or $L_k = R_{k-1}$ , otherwise
   $R_k = L_{k+1}$ , if $\exists\ T_i \in Q,\ j > k$
   or $R_k = m$ , otherwise
   where $m$ is a predefined maximum acceptable time span for holding an exclusive lock

4. Coordinator constructs an execution plan using *ProviderSelectionAlgorithm* (Section 4.2)

5. Coordinator computes the mutually agreed upon time span for holding the exclusive locks as follows:
   - Deadline for acquiring all locks:
     $L = \max(L_i)$ , $\forall\ t_i \in T$
   - Deadline for releasing all locks:
     $R = \min(R_i)$ , $\forall\ t_i \in T$

6. Coordinator establishes service level agreements with the selected providers on the specified values in 5 and starts the actual execution of $T$

Step 3 of the Transaction Scheduling Algorithm, (which is conducted by the service providers) shows how the service providers construct their offers for a service request (i.e. how the values of L and R are computed). Service providers use their local transaction schedulers to allocate the submitted request in the current schedule. They base their offer on the existing agreements with active transactions. If there is no pre-ordered transaction in the local schedule, the exclusive lock can be granted as soon as the service is executed; otherwise, the provider offers a shared lock and the estimated time to acquire an exclusive one is set to the (agreed) commit time of the closest pre-ordered transaction in the queue. The provider's deadline for releasing the lock depends on the list of post-ordered transactions. If there is a post-ordered transaction in the local schedule, the provider sets the deadline to the (agreed) time of granting an exclusive lock

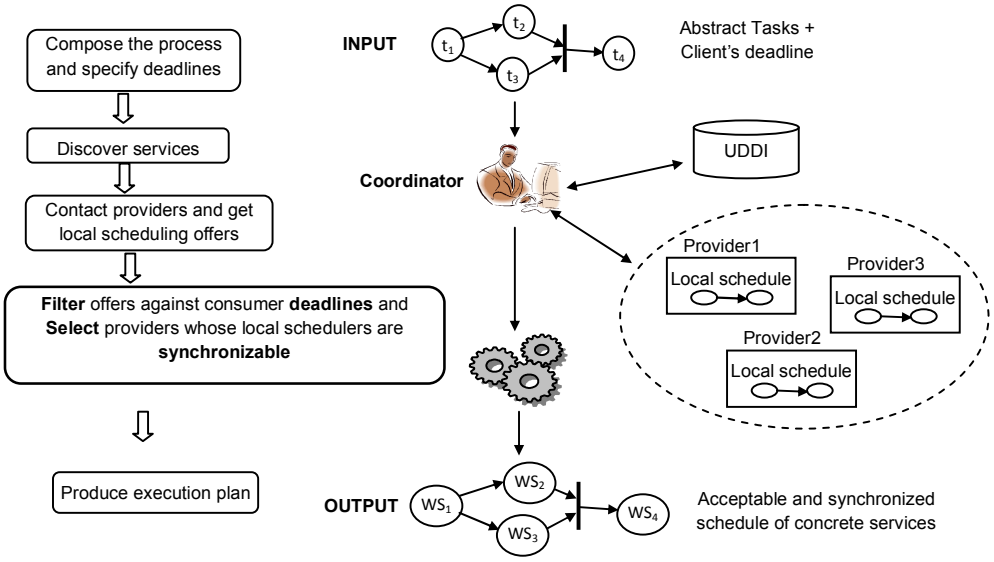


**Figure 4. Overview of the proposed approach**

for the closest post-ordered transaction, otherwise, the provider sets its deadline to some arbitrary value (e.g. the maximum accepted holding time). After receiving all offers and selecting the candidate providers, the Coordinator defines the start and end time of the 2PC protocol, i.e. the required time of acquiring all exclusive locks and the expected time of releasing them (Step 5) and sends this information to the providers.

## 4.2 Provider Selection

Step 4 of the Transaction Scheduling Algorithm is the most challenging part in this algorithm. Given all required information from local schedules at the different provider sites, the Provider Selection Algorithm is used to efficiently find a correct and possibly optimal combination of Web service providers such that no timing conflicts between their local schedules occur, and hence, no transaction blocking is needed. This is not a trivial task, especially when the number of available providers for each service is relatively high. For example, a transaction T involving N Web services, with M available providers for each service would result in $M^N$ possible execution plan for T. And given that the scheduling phase should be executed prior the actual start of the transaction as a short and atomic process, the provider selection process is crucial to the performance of the whole scheduling mechanism. In this section we provide the Provider Selection Algorithm to solve this problem.

Consider the example shown in Figure 5 to explain the given algorithm. A transaction consists of three Web services $WS_1$, $WS_2$ and $WS_3$. There are three different providers for each service. Figure 5.a shows the collected offers (time spans for holding the exclusive locks) from all providers after sorting them in an ascending order with respect to the expected time of lock acquisition.

***Provider Selection Algorithm:***

1. **Input:**
2. $Dc$ = client's deadline for committing transaction $T$
3. $F = \{F_1, F_2, ..., F_n\}$, a set of collected offers,
4. $F_i$ := list of provider offers for task $t_i \in T$, $1 < i < n$
5. each offer in $F_i$ has the form (P, L, R), where
   P is a reference to the provider,
   L and R: lock acquisition and release times
   $F_i$ is a sorted list in ascending order with respect to $L$
6. **Start:**
7. **do**
8. /* find $L_{max}$ : the time of acquiring the last lock */
9. **Let** $L_{max}$ = Max $(x | x = F_v[0](L)$ , $1 \le v \le n)$
10. $Plan[v] \leftarrow F_v[0]$
11. **foreach** $t_i$ in $T$ , $i \neq v$ **do**
12. **while** $(F_i[0](L) \le L_{max} \wedge Plan[i]$ isEmpty) **do**
13. **if** ( $F_i[0](R) > L_{max}$) **then** $Plan[i] \leftarrow F_i[0]$
14. **else** $F_i = F_i - F_i[0]$
15. **end while**
16. **if** ($Plan[i]$ isEmpty ) **then**
17. $F_v = F_v - F_v[0]$
18. $Plan = \{\}$, ***goto*** 21
19. **end if**
20. **end for**
21. **while** ($Plan == \{\} \wedge L_{max} < Dc$)
22. **return** $Plan$

The algorithm goes in several rounds until an acceptable plan is found or the client's deadline Dc (the red line) is reached. In each round the algorithm considers only the offers in the top of each order-list of provides' offers ($P_1$,

$P_4$ and $P_7$ in Figure 5.a), and determines the offer with the latest time ($L_{max}$) for acquiring the lock among them (the green dashed line). The algorithm then cancels all offers that: 1) have conflict with the offer of $L_{max}$ and 2) lies to the left of $L_{max}$ on the time line. The algorithm then checks if there is at least one provider for each service whose offer overlaps with this offer. In the given example (Figure 5.a) the first round fails because $WS_1$ has no valid offer. The second round then starts (Figure 5.b) by determining the new $L_{max}$ value and repeating the previous steps. In this round the offers $P_4$, $P_5$ and $P_7$ are canceled because they conflict with $P_2$ ($L_{max}$) and lie to the left of the dashed line ($L_{max}$). The algorithm however

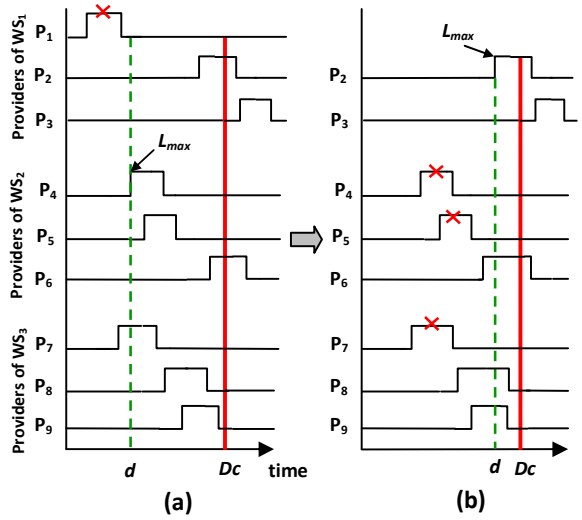


**Figure 5. Example of Provider Selection Algorithm**

succeeds in finding a conflict-free offer for each service in this round, i.e. $P_2$, $P_6$ and $P_8$ for the services $WS_1$, $WS_2$, and $WS_3$ respectively.

Thus, by ordering the offers of each group of providers and eliminating all offers that conflict with $L_{max}$ in each round, we argue that the proposed algorithm is able to find a conflict-free execution plan (if any) as early as possible without the need to try all possible combinations of service providers.

## 4.4 Discussion

In the following we discuss the proposed approach from different perspectives.

**Blocking prevention vs. resolution:** In contrast to the proposed solutions in the literature for real-time databases, which attempt to solve the blocking problem *at run-time* in a priority-based manner (by either aborting or delaying transactions with lower priority), our approach aims at avoiding blocking *prior* the actual execution. This is necessary since aborting long-running transactions in such a late stage (at commit time) is very expensive and might require cascading aborts of other transactions.

**Decentralization**: The proposed scheduling mechanism does not assume (and does not require) the existence of a global transaction manager, since such an assumption is unrealistic in the environment of autonomous and loosely coupled Web services. The proposed mechanism can be deployed in a fully decentralized fashion since it does not require the global knowledge over all running transactions when the scheduling decisions are made. Information about potential completion dependencies among concurrent transactions is processed and kept locally by the service providers. Such information is taken into account when scheduling offers are made by the providers, but not directly communicated to the transaction coordinators. Moreover, our approach preserves the autonomy of service providers since it does not influence the locally applied scheduling mechanism. Instead, it exploits the provided information from the local schedulers to intelligently select service providers and construct a globally agreed upon schedule that meets all participants' deadlines and timing requirements.

**Correctness issues:** Global serializability as a measure of the correctness and consistency of concurrent execution of transactions can be achieved through local guarantees [24]. The correctness of the produced global schedule in our approach is ensured by the applied concurrency control mechanism of local schedulers [e.g. 1, 2 and 3] as long as their local schedules belong to the COCSR family, i.e. commit-order preserving conflict serializable schedules [24]. Our algorithm focuses on selecting an acceptable execution plan out of all possible and correct plans that meets the time constraints of the transaction partners and avoids unnecessary blocking at commit time.

It is important to ensure that the local scheduling offers are kept valid during the provider selection process. Thus, the scheduling phase has to be executed in a timely manner and service providers should not accept any further scheduling requests until the end of this phase. We propose to execute the scheduling steps in an atomic manner, e.g. according to the AtomicTransaction specifications [6]. If the Coordinator fails in finding an acceptable execution plan or one of the selected providers rejects its offer, the whole AtomicTransaction is canceled, and providers can accept new scheduling requests.

Given that the Coordinator successfully completed the scheduling phase and all involved parties agreed on an acceptable 2PC length, every provider should insure that the promised time for granting an exclusive lock is held. This means that providers cannot offer exclusive locks on the respective resources for new transactions unless such locks will be released before the promised time expires.

**Performance issues:** The benefits of the proposed scheduling and pre-selection of providers do not come

without extra costs. However, we believe that the extra overhead of the pre-processing can be neglected compared to the cost of the unnecessary abort of transactions at the commit time because of missed deadlines. This is especially important, given that the compensation of some already successfully executed services is sometimes too expensive and often leads to cascading aborts of dependent transactions. Moreover, collecting the offline information about service implementations (e.g. expected execution length) as well as the online information (e.g. local schedule offers) can be performed as part of the service discovery and the negotiation on service level agreements with the selected providers. The required computations and processing of the collected scheduling offers and selecting service providers can then be performed locally at the Coordinator's machine. And given that the execution time of the Web service transactions is usually long (hours or even days) the required time for pre-processing can be, therefore, neglected.

**Implementation issues:** We assume in this paper that service providers advertise information about the expected processing time (e.g. average or maximum execution time based on statistical information). This can be realized for instance by extending the WSDL descriptions of the Web services to include such information and publish them in UDDI repositories. To

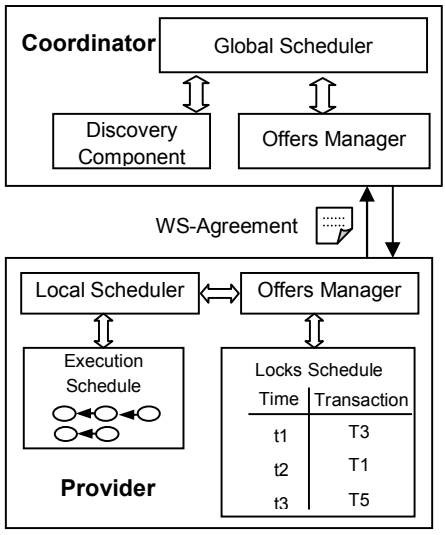


**Figure 6. The proposed Architecture**

support the proposed scheduling mechanism service providers need to provide Web services for requesting local scheduling offers by the transaction coordinators. Figure 6 sketches the proposed architecture and different components on both sides. The Execution Schedule and Exclusive Locks Schedule are used by the Offers Manager component at the service provider side to keep track of active agreements and help those making appropriate offers such that their local resources utilization is maximized. This component also facilitates monitoring any violation of established agreements.

For the scope of this paper we adopt the use of the Open Grid Forum's *WS-Agreement Specification* [25] for specifying and establishing agreements between the service coordinator and each service provider, which are called *Agreement Initiator* and *Agreement Responders* in the *WS-Agreement* terminology, respectively.

## 5. Implementation and Experimental Results

To evaluate our proposed protocol we implemented a prototype and ran several experiments, which we describe in this Section. We built our prototype on top of an implemented version of the WS-Transaction specification [5] with an extension to support concurrency control and handle distributed deadlocks as described in our previous work [1]. We extended the given architecture by implementing the components shown in Figure 6 to support the proposed scheduling mechanism.

For experimental evaluation purposes we simulated the environment of concurrently running web service-based transactional processes. In each experiment we ran a number of concurrent transactions each of them is assigned to a coordinator and is executed in a different thread. Every transaction is composed of a (randomly set) number of tasks; each of them can be accomplished by one of several available Web services from different providers. Every call to a service starts a new thread, which performs some transactional operations (read/write) on some local resources (text files) assigned to some specific provider.

**Table 1. Simulation setup**

| | |
|---|---|
| num. of concurrent transaction | 10 to 100 |
| num. of Web services per transaction | 3 to 5 |
| num. of providers per service | 10 |
| Service execution length | 5 to 60 sec |
| User interaction length | 5 to 60 sec |
| Transaction length | 30 to 400 sec |
| Distribution of transaction lengths | Normal |
| Transactions arrival average | 1 trans per sec |
| Max Client's deadline for committing whole transaction | 60 sec after execution |
| Max Providers' deadline for holding lock on one resource | 20 sec |

To simulate variant execution lengths of the Web services, we delay the return of the results by a randomly set amount of time and to simulate the time needed by the client to process the results (human interaction) we introduce a randomly set delay at the client side between

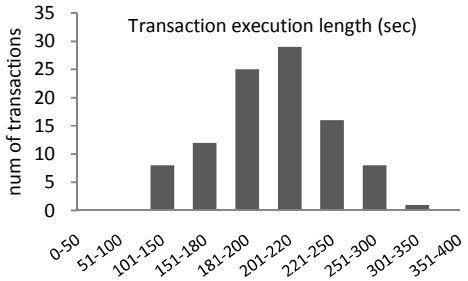


**Figure 7. Ditribution of transactions' lengths**

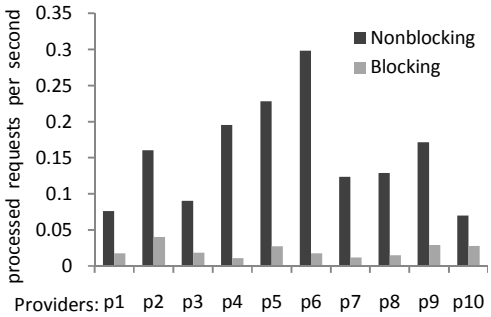


**Figure 8. Resources utilization**

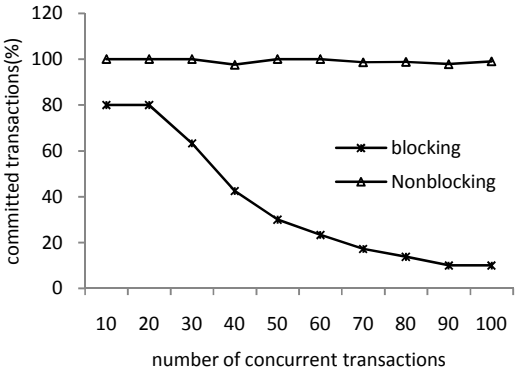


**Figure 9. Committed transactions before deadlines**

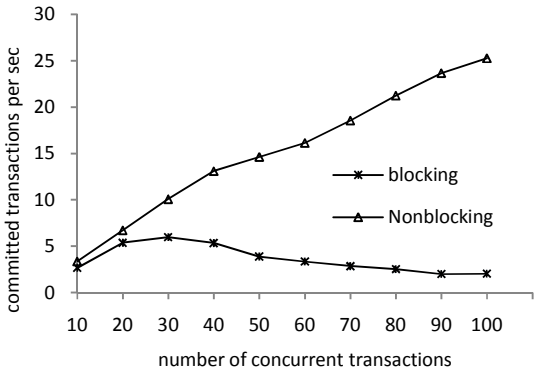


**Figure 10. Overall throughput**

service invocations. All random numbers including transactions' execution length (see Figure 7) follow the normal distribution. Table 1 summarizes the different parameters of the simulation setup in our experiments.

We ran all experiments on a machine with a 2GHz Genuine Intel CPU, T2500, processor and 2GB RAM equipped with Microsoft Windows XP Professional Version 2002. The JVM used is J2SE 1.5.

In the experiments we compared between the conventional (blocking) scheduling and our proposed nonblocking scheduling mechanism. In the former case each task in a transaction is assigned to a random service provider and the service is invoked directly according to the given workflow. Service providers ensure the serializability of their local schedules and preserve the commit order of concurrent transactions. In the latter case, transaction Coordinator constructs an execution plan and pre-selects service providers based on their current offers using the proposed algorithms in Section 4. Figure 8 shows the advantage of the proposed approach from the provider's perspective. The number of processed requests per second at each provider was measured. Using the nonblocking scheduling has improved the utilization of local resources at all provider sites significantly. In Figure 9 we measured the percentage of successfully completed transactions when applying the proposed nonblocking scheduling in comparison with the normal scheduling without pre-selection of providers. In this experiment, transactions that fail to commit before the deadlines are not restarted. We ran the experiment in different runs with different concurrency levels represented by the varying number of concurrent transactions between 10 and 100. By increasing the level of concurrency the probability that completion dependencies between transactions occur, increases and hence the probability that transactions are blocked at the commit time increases as well. The results of the experiment (Figure 9) show that with nonblocking scheduling almost all transactions terminated successfully before the deadlines. This is a significant improvement from the client's perspective since the number of aborted transactions because of missed deadlines is minimized.

Figure 10 shows that the overall throughput (i.e. number of successfully terminated transactions per second in the whole system) using the proposed approach is much higher than using the conventional scheduling. With blocking scheduling the throughput reaches its peak very quickly and starts to decrease while the level of concurrency increases. In this experiment, the total time between submitting and committing transactions including the required time for provider selection process was considered.

## 6. Conclusion and Future Work

In this paper we described a nonblocking mechanism for scheduling global transactions. The proposed solution enhances conventional scheduling algorithms for concurrency control to support the time characteristics of the transactions in Web service environment. By applying the proposed scheduling mechanism unnecessary blocking of transaction commitment during the execution of a 2PC protocol is avoided, thus saving time and costs of later abort or missed deadlines. The proposed approach is beneficial for both the service consumer and service provider. The experimental results showed a significant improvement in terms of number of successfully completed transactions within acceptable time frames as well as in terms of resources utilization. Our future work will focus build a cost-based model for both the client and provider that can help them making decisions regarding the available offers. We also plan to extend the proposed scheduling mechanism especially the Provider Selection Algorithm to allow negotiation-based adjustments to established agreements at run time if required.

## 7. References


[1] M. Alrifai, P. Dolog, and W. Nejdl, "Transaction Concurrency Control in Web Service Environment". In Proc. of the Europ. Conf. on Web Services (ECOWS), Zurich, Switzerland, 2006.

[2] K. Haller, H. Schuldt, and C. Türker, "Decentralized coordination of transactional processes in peer to peer environments", In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*, Bremen, Germany, 2005.

[3] S. Choi, et al., "Maintaining consistency under isolation relaxation of web services transactions", In *Proc. of Int. Conf. on Web Information Systems Engineering (WISE)*, New York, NY, USA, 2005.

[4] OASIS *Business transaction protocol*, 2004, published at http://www.oasis-open.org/committees/ documents.php?wg_abbrev=business-transaction.

[5] Web services Business Activity framework, 2005, published at ftp://www6.software.ibm.com/ software/developer/library/WS-BusinessActivity.pdf

[6] Web services Atomic Transaction, 2005, published at ftp://www6.software.ibm.com/software/ developer/library/WS-AtomicTransaction.pdf

[7] W.-T. Balke and M. Wagner, "Cooperative Discovery for User-centered Web Service Provisioning", In *Proc. of the Int. Conf. on Web Services (ICWS)*, Las Vegas, NV, USA, 2003.

[8] S. Bhiri, O. Perrin and C. Godart, "Extending workflow patterns with transactional dependencies to define reliable composite Web services", In *Proc. of the Int. Conference on Internet and Web Applications and Services*, 2006.

[9] J. Gray, "The transaction concept: Virtues and limitations", In *Intl. Conference on Very Large Data Bases*, 1981.

[10] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. P. Buchmann, "Extending bpel for run time adaptability", In the 9th *Int. Enterprise Distributed Object Computing Conference*, Enschede, The Netherlands, 2005.

[11] M. Schäfer, P. Dolog, and W. Nejdl, "Engineering Compensations in Web Service Environment", In *International Conference on Web Engineering*, Como, Italy, July 2007

[12] B.W. Lampson, "Atomic Transactions," in Distributed Systems: Architecture and Implementation--An Advanced Course, B.W. Lampson (Ed.), *Lecture Notes in Computer Science*, Vol. 105, Springer-Verlag, 198l, pp. 246-265.

[13] M. P. Papazoglou, "Web Services and Business Transactions", *World Wide Web*, v.6 n.1, p.49-91, 2003

[14] Alonso, G. et al, "Unifying concurrency control and recovery of transactions", *Information Syst*ems, Volume 19, Issue 1 (Jan. 1994), P101-115.

[15] Garcia-Molina, H. and Salem, K. Sagas. In *Proc. of the Int. Conference on the Management of Data (SIGMOD),* 1987.

[16] Bhiri, S., Perrin, O., and Godart, C., "Ensuring Required Failure Atomicity of Composite Web Services", In *Proc. of 14th Int. Conference on World Wide Web (WWW'05)*, Japan, 2005.

[17] A. Elmagarmid, "Transaction Models for Advanced Database Applications", Morgan-Kaufmann, 1992.

[18] Gustavo Alonso , et al., "Advanced Transaction Models in Workflow Contexts", *Proceedings of the Twelfth International Conference on Data Engineering*, p.574-581, 1996

[19] Fauvet, M., et al., "Handling Transactional Properties in Web Service Composition", *In Proceedings of 6th International Conference on Web Information Systems Engineering (WISE'05)*, New York, NY, USA, 2005

[20] L. Sha, R. Rajkumar, and J. P. Lehooczky, "Concurrency control for distributed real-time databases", *ACM SIGMOD Rec*. 17, 82-98, Mar. 1988.

[21] Lin, Y., and Son, S., "Concurrency control in real-time databases by dynamic adjustment of serialization order", *IEEE Rea-time Syst. Symp*., 104– 112, 1990

[22] J. R. Haritsa, M. Carey and Y. M. Livny, "Dynamic real-time optimistic concurrency control", *IEEE Real-Time Sys. Symp.*, 94-103, (Dec. 1990)

[23] Web services coordination 2005, published at ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[24] G.Weikum and G.Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann, 2002

[25] Web Services Agreement Specification (WS-Agreement), September 2005, published at http://www.gridforum.org/Public_Comment_Docs/Documents/ Oct-2005/WS-AgreementSpecificationDraft050920.pdf