Towards Context-Aware Transaction Services

Romain Rouvoy[1], Patricia Serrano-Alvarado[2], and Philippe Merle[1]

[1] INRIA Futurs - Jacquard Project,
LIFL - University of Lille 1,
59655 Villeneuve d'Ascq Cedex, France
{romain.rouvoy, philippe.merle}@inria.fr
[2] ATLAS-GDD Team,
LINA - University of Nantes,
44322 Nantes Cedex 03, France
patricia.serrano-alvarado@univ-nantes.fr

Abstract. For years, transactional protocols have been defined for particular application needs. Traditionally, when implementing a transaction service, a protocol is chosen and remains the same during the system execution. Nevertheless, the dynamic nature of nowadays application contexts (e.g., mobile, ad-hoc, peer-to-peer) and context variations (semantics-related aspects) motivates the need for transaction service adaptation. Next generation of transaction services should be adaptive or even better self-adaptive. This paper proposes CATE: (1) a component-based architecture of standard 2PC-based protocols and (2) a Context-Aware Transaction sErvice. Self-adaptation of CATE is obtained by context awareness and component-based reconfiguration. This allows CATE to select the most appropriate protocol with respect to the execution context. We show that using CATE performs better than using only one commit protocol in a variable system and that the reconfiguration cost is negligible.

1 Introduction

The dynamic nature of nowadays application contexts (e.g., mobile, ad-hoc, peer-to-peer) and context variations (semantics-related aspects) justifies the need for application adaptation [1]. Next generation of applications should automatically tune themselves and apply optimizations in order to maximize performances, to evolve, to face different contexts or to adapt the execution process according to context variations.

Component-based models are a good solution to make possible software adaptability [2] mainly because component-based architectures facilitate static and dynamic configuration. Implementing component-based adaptive applications is a very active and consolidated research/industrial issue [3, 4]. Nevertheless, there has been little work on adaptability of middleware services, such as persistence, replication, transaction, or communication [5, 6, 7].

In distributed transaction management, commit protocols ensure atomicity, which means that all transaction operations success (commit) or none of them

F. Eliassen and A. Montresor (Eds.): DAIS 2006, LNCS 4025, pp. 272–288, 2006.

(abort). The most used commit protocol is Two-Phase Commit (2PC) [8]. There exists a number of 2PC optimizations and some of them are so widely used that, as 2PC, are part of transaction processing standards. 2PC variations are proposed to optimize transaction execution costs, to address particular transaction semantics (e.g., read-only), to execute on different network topologies, etc. For instance, the 2PC Presumed Commit protocol (2PC-PC) [9] is well suited for high transaction commit rates, whereas 2PC Presumed Abort (2PC-PA) [9] is more appropriate for high transaction abort rates.

Traditionally, transaction service implementations are tailored for a particular application context. A transactional protocol is chosen and remains the same even if the application context changes. This may lead to unexpected poor performances. To deal with context variations of transactional applications, the transaction management system should be adaptive or even better self-adaptive. We consider self-adaptation as the ability of being aware of the application context changes and the capacity of reacting to them. This paper proposes CATE, which is composed of (1) a component-based architecture of standard 2PC-based protocols and (2) a Context-Aware Transaction sErvice. Self-adaptation of CATE is obtained by a context-aware mechanism and component-based reconfiguration. This allows CATE to select the most appropriate protocol with respect to the execution context. The implementation performance results show that using CATE performs better than using only one commit protocol in a variable system and that the reconfiguration cost is negligible.

This paper is organized as follows. Section 2 briefly introduces the atomic commit protocols used in this work. Section 3 introduces the component-based implementation and the evaluation of the 2PC, 2PC-PA, and 2PC-PC protocols. Section 4 presents our Context-Aware Transaction Service, its implementation and some empirical measures obtained when using it. Finally, Section 5 presents some related work, and Section 6 concludes and gives future work.

2 Overview of Commit Protocols

In database systems, correct concurrent data access is ensured using transactions. Transactions are characterized by the well-known ACID (Atomicity, Consistency, Isolation and Durability) properties, which are guaranteed by transaction services. While we consider that consistency, isolation and durability properties are supported by the application resource managers (e.g., Database), this paper focuses on the atomicity property. In particular, our work focuses on the self-adaptability of the atomicity property. For the purposes of this paper, we concentrate on some standard 2PC-based protocols, which are the 2PC, 2PC-PA and 2PC-PC protocols.

To describe the behavior of these 2PC protocols, we use UML sequence diagrams (see Figures 1 to 3). It allows us to identify four actors: Application, Coordinator, Participants, and Log. Then, the sequences describe the behavior of the 2PC, 2PC-PA and 2PC-PC protocols in terms of communication schema and logging issues. Indeed, the resilience of commit protocols to system and

communication failures is achieved by logging the progress of the protocol in the logs (stable storage) of the coordinator and the participants. There exist two types of log writes: *force* and *non-force*. The first one is immediately flushed into the log, generating a disk access. Non-force writes are eventually flushed into the log. Thus, there exists a window of vulnerability in using non-force writes until they are flushed.

Figures 1 to 3 introduces three commit protocol use cases. Two cases correspond to the situation where the Application orders the Coordinator to commit. In this case, the commit protocol can issue with a Commit (e.g., Figure 1(a)) or a Failure (e.g., Figure 1(b)) depending on the Participants votes. In the third case, the Application orders the Coordinator to abort and the commit protocol issues automatically with an Abort decision (e.g., Figure 1(c)).

2.1 Two-Phase Commit (2PC)

2PC, the most used commit protocol, consists of two phases (see Figure 1). During the *voting phase*, the coordinator sends a *prepare* message to all participants. At the *decision phase*, the coordinator decides to commit (if all the participants vote *yes*) or abort (if at least one participant votes *no*) the transaction and notifies the participants of its decision. When the participants receive the final decision, they send an *acknowledge* message to the coordinator and release all resources held by the transaction. When the coordinator has received all the acknowledgements from the participants that voted yes, it ends the protocol and forgets the transaction. In 2PC, the coordinator force writes a *decision* record and non-force writes an *end* record at the end of the protocol. Participants force write their votes and the coordinator's decision. Write operations are logged before sending the corresponding message.

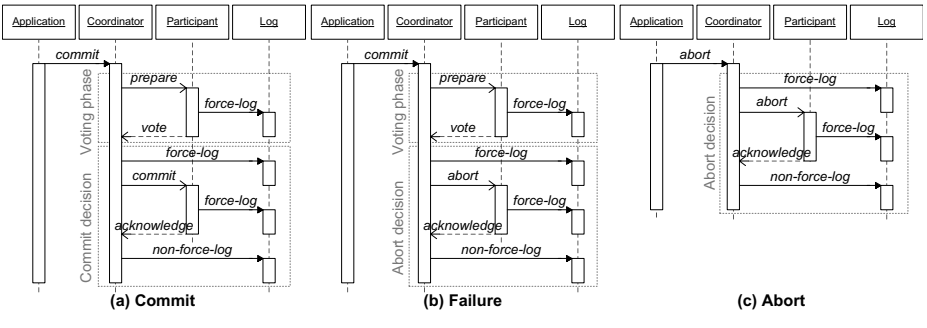

Fig. 1. The 2PC protocol

2.2 2PC Presumed Abort (2PC-PA)

2PC-PA reduces the cost associated to aborted transactions. When the coordinator decides to abort a transaction, it discards all information related to the

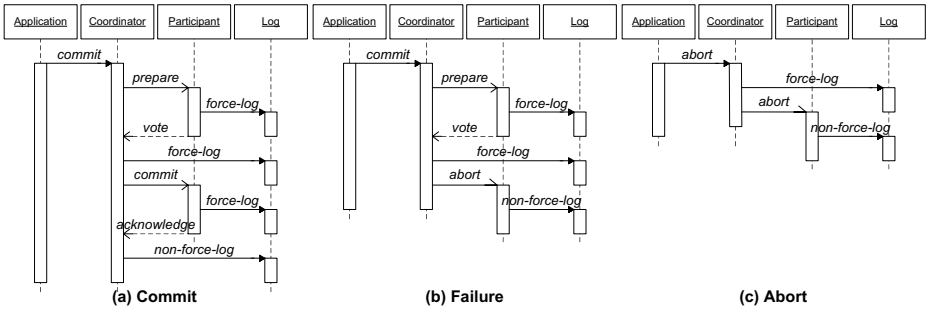

Fig. 2. The 2PC-PA protocol

transaction and sends an *abort* message to all the participants without logging the abort decision (see Figure 2(b) failure case). The participants non-force write the *abort* record and do not have to send an *acknowledge* message to the coordinator. In case of failures, the coordinator, not finding any information in the log regarding the transaction will deduce an abort decision. The commit case of 2PC-PA remains the same as in 2PC.

2.3 2PC Presumed Commit (2PC-PC)

2PC-PC, as opposed to 2PC-PA, reduces the cost of committed transactions. In 2PC-PC, the coordinator interprets missing information as a commit decision. To do so, the coordinator has to force write an *initiation* record for the transaction before sending *prepare* messages to participants (see Figure 3). When the coordinator decides to commit a transaction it force writes a *commit* record then it sends the commit decision. The participants non-force write the commit decision and release all the transaction resources without acknowledging the commit decision to the coordinator. Otherwise, when the coordinator decides to abort a

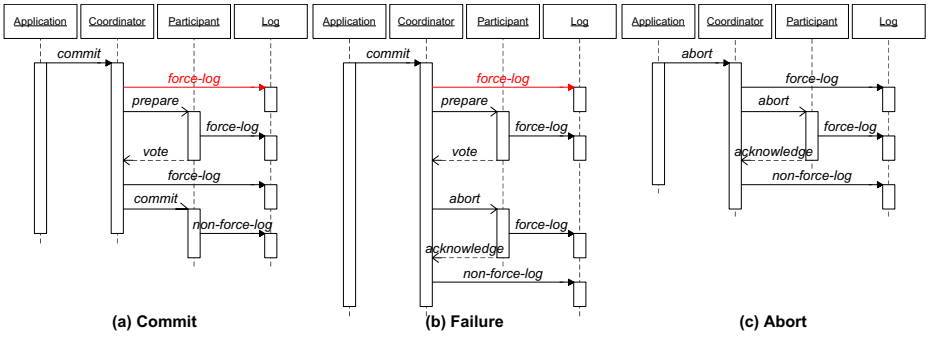

Fig. 3. The 2PC-PC protocol

transaction, it sends *abort* messages to all the participants that voted yes and waits for the acknowledges. The abort decision is not logged. When all the acknowledgements have been received, the coordinator writes a non-forced *end* record and discards all information related to the transaction. The participants force write the abort decision and send an acknowledgement to the coordinator.

3 Evaluation of Commit Protocols

Evaluation of commit protocols is often based on theoretic cost evaluation of message exchanges, and number and type of logs. This section aims at verifying the conformance of the theoretic costs to empirical measures obtained when implementing the protocols introduced in Section 2. The originality of this implementation lies in the definition of reusable components to implement various commit protocols. These components are reused in CATE to support self-adaptability of commit protocols as described in Section 4. The following sections analyse the theoretic costs of studied protocols, introduces details about the implementation of those protocols, and shows some empirical measures resulted from the execution of implemented protocols.

3.1 Theoretic Cost Measures

We show that these protocols differ in the number of messages sent and the number of forced log writes. Table 1 summarizes the three 2PC-based protocol costs. The differences between commit protocols lead to different completion time of the commit processing, communication and disk access costs. As in Section 2, the *Abort* use case considers transactions aborting unilaterally whereas the *Failure* use case depicts transaction aborting during the voting phase.

Table 1. The commit protocol theorical costs

Commit protocol	Messages			Forced log writes		
	Commit	**Failure**	**Abort**	**Commit**	**Failure**	**Abort**
2PC	$4p$		$2p$	$1+2p$		$1+p$
2PC-PA	$4p$	$3p$	p	$1+2p$	$1+p$	1
2PC-PC	$3p$	$4p$	$2p$	$2+p$	$1+2p$	$1+p$

Even though 2PC is widely implemented, it is considered as very expensive as shown in Table 1. It costs $4p$ message exchanges (p being the number of participants) and $1+2p$ forced log writes (the cost of non-forced log writes can be ignored). This highlights why several 2PC optimizations have been proposed.

Besides saving one force log write at the coordinator and at the participant's sites, 2PC-PA saves one *acknowledge* message from each participant in the abort case. Thus, when the commit process fails, 2PC-PA costs $3p$ messages and p forced log writes. If the transaction aborts unilaterally, 2PC-PA costs only 1

message and 1 forced log write, making it cheaper than 2PC and 2PC-PC. The cost to commit a transaction is the same as in 2PC.

Compared to 2PC, 2PC-PC saves one forced log write and one *acknowledge* message from each participant for the commit case at the expense of one extra *initiation* forced log write at the coordinator. Thus the cost of committing a transaction is $2 + p$ forced log writes and $3p$ messages. For the abort case, 2PC-PC has one extra forced log write at the coordinator, the *initiation* record. Thus, aborting a transaction costs the same as in 2PC ($1 + 2p$ forced log writes and $4p$ messages).

Thus, it is cheaper to use 2PC-PA in a system where transactions are most likely going to abort, whereas, it is cheaper to use 2PC-PC if transactions are most probably going to commit. In a system where transactions have the same probability of abort and commit, it is cheaper to use 2PC-PA.

3.2 Implementation Issues

This section introduces the implementation of commit protocols presented in Section 2. These commit protocols are implemented using component-based software engineering. Before getting into the architecture details, we extend the definition of component proposed in [10] with the concepts defined in the Fractal component model [2]. A component architecture (or *configuration*) is mainly composed of *components* and *bindings*. A component is a software entity, which exports functions through *server interfaces* and imports its dependencies via *client interfaces*. A binding connects a client interface to a server interface to resolve a component dependency.

The proposed architecture generalizes the commit protocols to reuse common functionalities. The objective is (1) to make a component-based implementation of the three commit protocols presented in Section 2 and (2) to express principal differences only through bindings. Therefore, each commit protocol reuses exactly the same components but assembled with different bindings (see Figure 4).

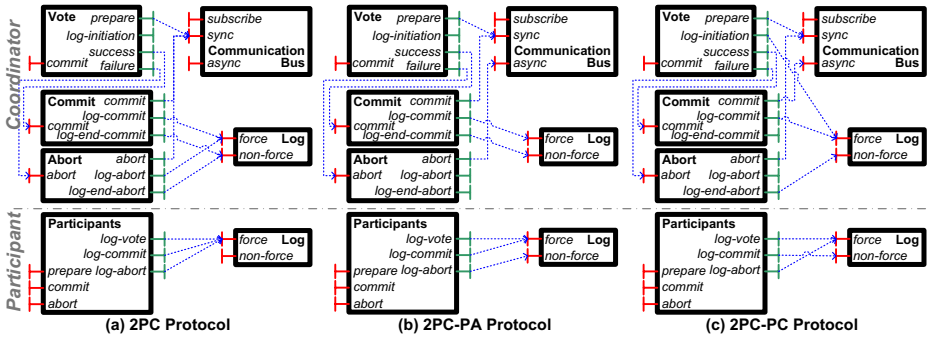

Fig. 4. Component-based architecture of 2PC-based protocols

To implement the commit protocols, we do not require to reify the Application object depicted in Figures 1 to 3. Thus, we define 3 components: Coordinator, Participants, and Log. To enforce the reuse of components, the Coordinator component is split into 3 components: Vote, Commit, and Abort. This separation means that a commit protocol encloses not only the 2PC protocol but also an abort protocol in case of failure. This abort protocol is reused to support aborting transactions unilaterally. Communication is supported by a Communication Bus component. The communication bus supports sending *synchronous* or *asynchronous* (using a callback approach) messages.

In Figures 1 to 3, the coordinator sends the prepare, commit and abort messages to all participants. Thus, the *prepare*, *commit* and *abort* client interfaces of the Coordinator are bound to the Communication Bus. In Figures 1 to 3, the coordinator and the participants require to journalize the steps of the commit protocol in a log. Thus, the Coordinator and the Participants component interfaces are bound to the *force* or *non-force* server interface of the Log component depending on the method call label declared in the sequence diagrams.

The coordinator part of this architecture is embedded in the transaction service whereas the participant part is implemented by resource managers (e.g., database managers) involved in the system. The implementation of the 3 commit protocols reuses these 6 components by changing only the bindings to provide the different semantics.

2PC. In 2PC (Figure 4(a)), the Coordinator sends a synchronous *prepare* message to all participants. Participants should attach their vote to the callback message returned to the coordinator. When a decision is taken, the Coordinator calls the *log-commit* (resp. *log-abort*) interface, which is bound to the *force* interface of the Log. The *commit* (resp. *abort*) message is sent synchronously to allow participants to *acknowledge* the decision. To terminate the protocol, the Coordinator non-force writes an *end* record calling the *log-end-commit* (resp. *log-end-abort*) interface. The Participants component receives (from the Communication Bus) the *prepare*, *commit*, and *abort* messages. It force writes its vote and the coordinator's decision.

2PC-PA. In 2PC-PA (Figure 4(b)), as the coordinator does not log the abort decision, the *log-abort* interface is not bound to the Log. This leaves the Coordinator code unchanged. Next, the *abort* message is sent asynchronously because the abort decision does not need to be acknowledged. Finally, the *log-end-abort* interface is not bound to the Log because the end of an aborted transaction does not need to be logged. The commit case is the same as in 2PC. In the Participants component, the commit case remains the same as in 2PC. In the abort case, *log-abort* is bound to the *non-force* interface of the participant's Log component.

2PC-PC. In 2PC-PC (Figure 4(c)), before sending the *prepare* message, the Coordinator calls the *log-initiation* interface. Compared to the other protocols, such an interface is bound to the *force* Log interface. In 2PC-PC, the commit decision is sent asynchronously because it is not necessary to acknowledge the commit decision. Since the end of a committed transaction does not need to

be logged, the *log-end-commit* interface is not bound to the Log. The abort decision is not logged, nevertheless, the *abort* message is sent synchronously. The end of an aborted transaction is non-force written into the log. In the Participants component, the *log-commit* and *log-abort* interfaces are bound respectively to the *non-force* and *force* interfaces of the Log.

3.3 Empirical Evaluation

The objective of this section is to compare the theoretic cost evaluation with the empirical evaluation of the component-based implementation of 2PC, 2PC-PC, and 2PC-PA protocols. This comparison (1) validates our implementations of 2PC-based protocols regarding to their specification, and (2) confirms the theoritical cost evaluations of these 2PC-based protocols with an empirical evaluation. The scenario of Figures 5, 6 and 7 evaluates the average completion time of a number of transactions executed sequentially varying the number of participants (from 0 to 5). This scenario is applied to the 2PC, 2PC-PA and 2PC-PC protocols. Experiments have been done on a PC Pentium IV 2,4GHz with 1Gb of memory using the Ubuntu Linux distribution, the Sun J2SE Development Kit 5.0, and the AOKell implementation of the Fractal component model [11].

In Figure 5, all executed transactions are committed. In this case, 2PC-PC behaves better than 2PC and 2PC-PA. This is because 2PC-PC saves 1 forced log write and 1 acknowledge message from each participant. The initial overhead of 2PC-PC is due to the initiation record that is automatically force written. 2PC and 2PC-PA have similar performance because their commit case follows the same process.

In Figure 6, all executed transactions fail during the commit process. This shows that 2PC-PA, whose completion time is closed to 0, performs much better than 2PC and 2PC-PC. This is because 2PC-PA saves 1 acknowledge message from each participant in the abort case (see Section 2.2). 2PC and 2PC-PC have similar performance because they have similar costs even if 2PC-PC makes an extra force log write (see Section 2.3).

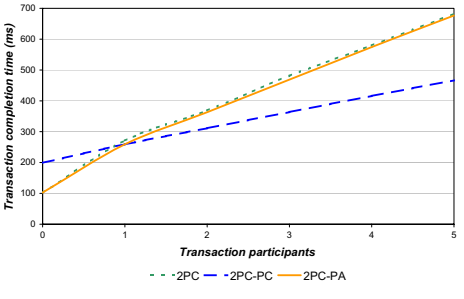

Fig. 5. High commit rate

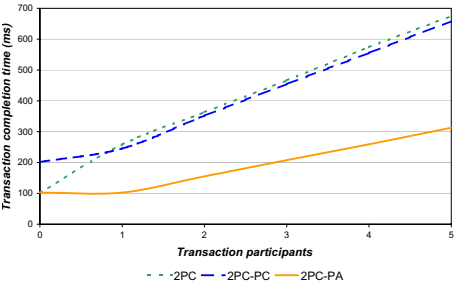

Fig. 6. High failure rate

In Figure 7, all executed transactions are aborted unilaterally. In this case, 2PC-PA performs much better than 2PC and 2PC-PC. This is because 2PC-PA uses only 1 asynchronous message and 1 forced log write to abort the transaction. 2PC and 2PC-PC have similar performance because their abort case follows the same process. The abort protocol applies only one issue. This predictable issue is applied by several 2PC optimizations [12, 13, 14] to exploit the efficiency of transaction aborted unilaterally.

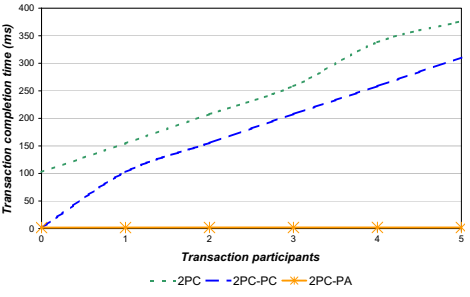

Fig. 7. High abort rate

4 CATE: A Context-Aware Transaction sErvice

In this section, we introduce the second part of our proposal, a Context-Aware Transaction sErvice (CATE), which supports application context variations. The objective of CATE is to apply the best fitting 2PC protocol in presence of unpredictable commit rates.

Section 4.1 shows the context aware mechanism used in our approach. Section 4.2 introduces the reconfiguration process. Section 4.3 presents the policy used to enable commit protocol reconfiguration. Section 4.4 presents some performance measures. Finally, Section 4.5 discusses several issues concerning CATE.

4.1 Context Awareness

In this paper, we consider the transaction abort and commit rate as the application context. Thus, to be able of changing at the right moment, the commit protocol, it is necessary to monitor the abort and commit rates of transactions. The commit rate represents the occurrence of the commit use case of a transaction service whereas the abort rate represents the occurrence of the failure and the abort use cases. This logic is named *adaptation policy*. An adaptation policy is defined by a kind of ECA rules (Event, Condition, Action). The Event is the commit/abort rate, the Condition specifies when it is necessary to change the active protocol and the Action is the protocol change.

Figure 8 shows a transaction manager (Tx Manager) and its relationship with some transaction components (Tx(2PC-PX)). The Context Awareness component implements the adaptation policy. It monitors the number of committed and aborted transactions. Besides, it decides when the active protocol should be changed. This is possible thanks to the predefined ECA rules. For instance a ECA rule may say: if (abort-rate $<$ 10%) then use 2PC.

To count the number of committed and aborted transactions, the Context Awareness component uses the *subscribe* interface provided by the Communication

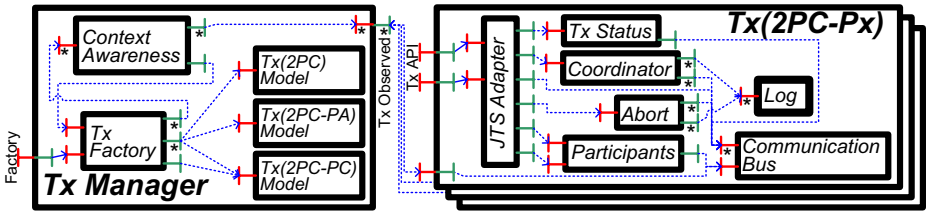

Fig. 8. CATE component-based architecture

Bus component of each transaction (see Figure 4). The *subscribe* interface allows subscribing to different kinds of events. Thus, the Communication Bus of active transactions notifies the Context Awareness component when each Coordinator sends *commit* or *abort* messages to the participants.

Figure 8 shows the general architecture of the CATE implementation, which supports JTS transactions [15]. The Tx(2PC-Px) component represents a JTS transaction implementing the 2PC-Px protocol. The JTS Adapter component is bound to components grouping the core functions provided by CATE. These core components include those presented in Figure 4 plus other general components such as the Tx Status. The Coordinator component reifies the commit protocol applied by the transaction. The Abort component reifies the abort protocol applied by the transaction when aborting unilaterally. This component reuses the Abort component defined in Figure 4 and it is configured with the 2PC-PA protocol. Thus, CATE provides the best completion time in case of predictable abort decisions. The Tx Manager component is in charge of managing the instance of active transactions. Figure 8 shows the Tx Manager component and its relation with the Tx(2PC-Px)'s Communication Bus component.

The commit protocol reconfiguration is done through a dedicated *configuration* attribute. This attribute is read by the Tx Factory component when new transactions are created. Depending on the value of this attribute, the Coordinator component implements the 2PC, 2PC-PA, or 2PC-PC protocol. Thus, the reconfiguration process consists of changing the value of this attribute depending on the predefined Conditions.

4.2 Reconfiguration Rules

To dynamically switch over another protocol, the Context Awareness component needs to *stop* the transaction factory, *unbind* the current protocol, *bind* the new one, and *restart* the transaction factory. In that way, newly created transactions use the appropriate commit protocol.

When the Context Awareness component decides to change a protocol (based on defined Conditions) it calls the *change-config* interface bound to the *configuration* interface of the Tx Factory component. Then the Tx Factory component connects the *active-config* interface to the appropriate configuration, which is

listed by the *available-config* interface. Thus, future transactions will be created using this new active configuration. In Figure 8, we show the transaction manager implementation containing the three commit protocols (Tx(2PC), Tx(2PC-PC) and Tx(2PC-PA)) and the active configuration is Tx(2PC-PC).

When the Tx Factory component creates a new transaction, it subscribes the *listener* interfaces (retrieved via the *probe* interface) of the Context Awareness component to make possible the commit/abort event monitoring.

4.3 Adaptation Policy

Knowing the current commit/abort rate allows predicting the future transaction context. That is, if the abort rate is about 30%, we consider that this tendency will remain the same in a near future. This is why the abort/commit rate motivates the reconfiguration.

The Condition that specifies when to change of commit protocol in CATE (see Section 4.1) is based on the following equation:

$$\begin{cases} x + y = 100 \\ x \times C_{2PC-PC} + y \times A_{2PC-PC} < x \times C_{2PC-PA} + y \times A_{2PC-PA} \end{cases}$$

Where x (resp. y) represents the number of transaction committed (resp. aborted) and C_{2PC-PX} (resp. A_{2PC-PX}) represents the commit (resp. abort) cost of the 2PC-PX protocol (here 2PC-PA and 2PC-PC protocols).

The solution of this equation is:

$$\begin{cases} y = 100 - x \\ x > \frac{100 \times (A_{2PC-PA} - A_{2PC-PC})}{(C_{2PC-PC} - A_{2PC-PC} - C_{2PC-PA} + A_{2PC-PA})} \end{cases}$$

Figure 9 applies this solution to the measures of Figure 5 and 6. It appears that the limit between 2PC-PC and 2PC-PA depends on the number of transaction participants. For example, in the case of transactions involving 20 participants, 2PC-PC becomes more interesting than 2PC-PA when the commit rate is above 54%. This limit is used by CATE to switch between the 2PC-PC and the 2PC-PA configurations.

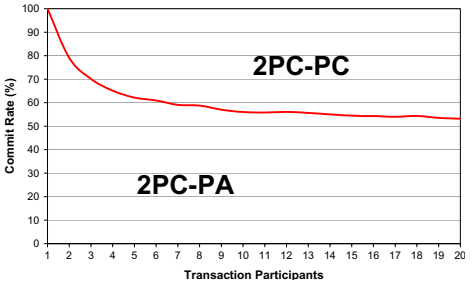

Fig. 9. 2PC-PA/2PC-PC border rate

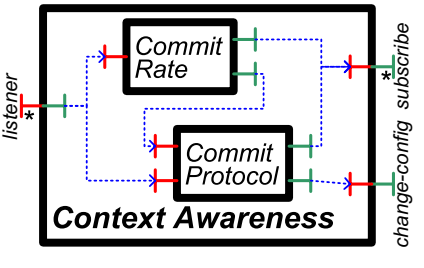

Fig. 10. Adaptation policy architecture

Figure 10 introduces the component-based architecture of the adaptation policy. This policy is composed of two parts. The Commit Rate component computes the appropriate commit rate depending on the average number of transaction participants. Thus, the component reconfigures the Commit Protocol component depending on the execution context variations. The Commit Protocol component reconfigures the transaction factory depending on the current commit rate. The computation of the commit rate is based on a configurable weighted moving average. It ensures that the commit protocol is adapted to important fluctuations in the commit rate without reconfiguring too often.

4.4 Empirical Performance Measures

CATE does not switch to 2PC because taking as context only the commit/abort rate, 2PC is more expensive than the other considered protocols. The scenario of Figures 11 and 12 evaluates the average completion time of 50 transactions executed sequentially with constant commit/abort rate variations (10 transactions commit, then 10 transactions abort, then 10 transactions commit, etc.). Transaction services using static configuration of 2PC, 2PC-PA and 2PC-PC protocols are executed and compared to CATE. Figures 12 depicts the average completion time since the transaction service has been started.

The measures of Figure 11 show the average completion time that varies depending on the transaction commit/abort rates. Performance of CATE is the best thanks to its capacity of self-adaptation. 2PC-PA and 2PC-PC suffer from the context variations. In CATE, when the commit rate is high, the active protocol is 2PC-PC. Otherwise, CATE uses 2PC-PA. Thus, CATE benefits of the best performance of 2PC-PC and 2PC-PA. In this experiment, 2PC is used as the initial protocol. Performances of Figure 12 show that the CATE reconfiguration does not introduce important overheads compared to the static configuration of the use cases protocols while providing better completion time. The cost of switching between commit protocols appears only when a new transaction is created. CATE computes the commit rate of terminated transactions to create

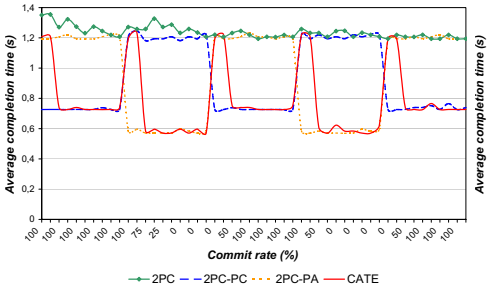

Fig. 11. Empirical performance measure

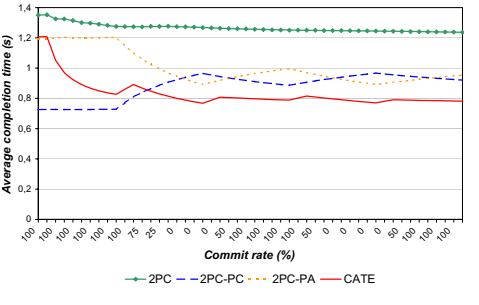

Fig. 12. History performance measure

a new transaction providing the best performing commit protocol. This mechanism coupled to a caching mechanism reduces the overhead of switching between the commit protocols.

4.5 Discussion

This section discusses various general aspects concerning CATE.

Reconfiguration of active transactions. Changing the protocol of active transactions compromises the recovery process in case of failures. That is why in CATE it is not possible to change the commit protocol once a transaction has begun. Different active transactions can use different commit protocols but each transaction begins and ends with the same commit protocol.

Using CATE. To be able of using CATE, for instance, in an application server, the following hypotheses should be guaranted. 1) The participant part is implemented by resource managers that are free to choose the way this implementation is done (Figure 8 suggests one implementation solution); 2) All considered protocols in CATE must be implemented by resource managers; finally, 3) Resource managers must be able to change the active protocol.

CATE extension. CATE may support other commit protocols that can be different to those used in this paper. We choose 2PC-based protocols as an experiment to show components reusability. Nevertheless, reusability is not necessary to the CATE operation. Thus, with CATE it is possible to switch to different commit protocol implementations, which makes it extensible. For example, 1PC and 3PC protocols can be considered because CATE monitors the commit rate of active transactions to find the best commit protocol. Besides, switching between 1PC or 3PC protocols requires the application to support 1PC or 3PC validation processes.

Predictable issues. Some commit protocols draw benefits from predicable issues of transactions [12, 13, 14]. Using piggybacking or callback mechanisms, they determine if the transaction is marked for read-only or abort before starting the commit protocol. Thus, depending on the known issue of the transaction, they can optimize the completion time of the transaction. This approach is complementary to our approach because CATE aims at optimizing transactions with unpredictable issues. However, CATE can also define transactions supporting several commit or abort protocols to improve the completion time of transaction with predictable issues.

Preserving the global semantics of the system. In software reconfiguration, it is necessary to preserve the semantics of the system. In our case, the transaction properties must be preserved. If an atomic commit protocol is replaced by another, which does not enforce the atomicity property (for instance, the semantic atomicity [16]), the transaction correctness is compromised. This is why in this paper, used protocols ensure the atomicity property. Thus, programmers must be careful about the choices they made when defining adaptive middleware systems.

5 Related Work

[5] proposes to dynamically adapt applications by composing at runtime (by weaving) functional (application-related) and non-functional concerns. Authors are interested in making the weaving process adaptive to runtime execution conditions. Their objective is to choose at runtime the appropriate non functional code. Thus, they propose to change the weaving of non-functional code according to context aware adaptation policies.

[6] proposes runtime application adaptability by assembling appropriate non-functional services thanks to service repositories. Repositories contain component-based non-functional services and meta-information describing such services. This approach requires the applications to be developed using the component-based approach. Our approach does not make any assumption about the application design and we choose to adapt the non-functional service itself rather than the instance of the used service.

Compared to our proposal, [5] and [6] consider non-functional services as the adaptation grain. Our approach proposes self-adaptability of non-functional services using components as adaptation granularity. Unlike [5] and [6], we made several experiments that underline the advantages of our proposal. Our proposal is validated with performance measures that show the self-adaptability advantages.

This paper improves [17] by providing a description of legacy 2PC-based protocols using UML Sequence diagrams. Descriptions of commit and abort protocols are supported and can be implemented as various configurations built with reusable components. This paper improves the adaptation policy presented in [17] to consider the commit rate variation depending on the number of transaction participants. The support of commit protocols for transactions with predictable issues has been introduced. The completion time of transactions with predictable issues, such as transaction aborting unilaterally, is improved compared to traditional commit protocols.

[12] proposes a new commit protocol for self-adaptive Web services, which supports both 2PC-PA and 2PC-PC participants. Such a protocol allows participants with different presumptions to be dynamically combined in one transaction. Compared to the work presented in this paper, [12] does not address evolution concerns. In our work, we use 2PC, 2PC-PA and 2PC-PC as use cases. Our approach can easily support new commit protocols to extend the application adaptive ability.

In general, works presented in [18, 19, 9] are simulation-based. Performance results focus on the semantics of transactions (e.g. read-only transactions, update transactions, transaction‘s length) and the presence of failures. Whereas, in this paper, besides addressing performance of each protocol based on commit and abort rates, we address the performance of changing the software configuration to migrate from a protocol to another. Our performance results, based on a prototype implementation, shows that the reconfiguration cost is negligible compared to gains obtained from the use of appropriate protocols depending on the application context.

6 Conclusions and Future Work

Self-adaptation is a current challenge in component-based software engineering. Several works have been devoted to adaptive applications, nevertheless, there has been little work on adaptability of non-functional services. This paper focused on transaction services, and more specifically on the commit process. On the one hand, it proposed a component-based architecture of standard 2PC-based protocols. Each protocol contains exactly the same components but assembled according to different configurations. On the other hand, it proposed a Context-Aware Transaction sErvice (CATE). CATE selects the most appropriate commit protocol with respect to the execution context. Performance measures show that changing the commit protocol depending on the context performs better that using only one commit protocol on a variable transactional system.

Our future work includes to study the component-based configuration of other 2PC-based protocols (e.g., [12]) but also 1PC and 3PC protocols. The idea is to extend CATE to support more commit protocols. The evaluation of runtime performances of these additional commit protocols will be useful to refine the CATE adaptation policies, e.g., adding new conditions and reconfiguration actions to switch between protocols.

Besides, we consider to investigate a model-driven approach to design commit protocols using UML sequence diagrams (see Figures 1 to 3) and to automatically generate the implementation of the `Coordinator` components and their bindings to the `Communication Bus` and the `Log` components. This model-driven approach, complementary to that we defined into [20, 21], will provide a dedicated high level language to define, study, compare commit protocols, and also an efficient way to implement them.

Availability. CATE is freely available under an LGPL licence at the following URL: http://gotm.objectweb.org.

Acknowledgments. This work is partially funded by INRIA, and the Region Nord - Pas-de-Calais.

References

1. Preuveneers, D., Berbers, Y.: Adaptive Context Management Using a Component-based Approach. In: 5th Int. Conf. on Distributed Applications and Interoperable Systems (DAIS). Volume 3543 of LNCS. Athens, Greece, Springer (2005) 14–26
2. Bruneton, E., Coupaye, T., Leclercq, M. *et al.*: An Open Component Model and its Support in Java. In: Int. Symp. on Component-Based Software Engineering (CBSE). Volume 3054 of LNCS. Edinburgh, UK, Springer (2004) 7–22 http://fractal.objectweb.org.

3. David, P., Ledoux, T.: Towards a Framework for Self-Adaptive Component-Based Applications. In: Int. Conf. on Distributed Applications and Interoperable Systems (DAIS). Volume 2893 of LNCS. Paris, France, Springer (2003) 1–14
4. Layaida, O., Hagimont, D.: Designing Self-Adaptive Multimedia Applications through Hierarchical Reconfiguration. In: 5th Int. Conf. on on Distributed Applications and Interoperable Systems (DAIS). Volume 3543 of LNCS. Athens, Greece, Springer (2005) 95–107
5. David, P., Ledoux, T.: Dynamic Adaptation of Non-Functional Concerns. In: ECOOP Workshop on Unanticipated Software Engineering. Malaga, Spain (2002)
6. Arntsen, A.B., Karlsen, R.: ReflecTS: a flexible transaction service framework. In: 4th Middleware Workshop on Adaptive and Reflective Middleware (ARM). Volume 116 of AICPS. Grenoble, France, ACM Press (2005) 1–6
7. Coulson, G., Blair, G., Grace, P. *et al.*: OpenCOM v2: A Component Model for Building Systems Software. In: IASTED Int. Conf. on Software Engineering and Applications (SEA). Cambridge, MA, ESA (2004) 1–6
8. Gray, J.: Notes on Database Operating Systems. In: Advanced Course: Operating Systems. Number 60 in LNCS, Springer (1978)
9. Mohan, C., Lindsay, B., Obermarck, R.: Transaction Management in the R* Distributed Database Management System. ACM Trans. on Database Systems (TODS) **11**(4) (1986)
10. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc. (2002)
11. Seinturier, L., Pessemier, N., Duchien, L. *et al.*: A Component Model Engineered with Components and Aspects. In: 9th Int. SIGSOFT Symp. on Component-Based Software Engineering (CBSE). LNCS Stockholm, Sweden, Springer (2006) To appear.
12. Yu, W., Wang, Y., Pu, C.: A Dynamic Two-Phase Commit Protocol for Self-Adapting Services. In: Int. Conf. on Services Computing (SCC). Shanghai, China, IEEE (2004) 7–15
13. Al-Houmaily, Y.J., Chrysanthis, P.K., Levitan, S.P.: Enhancing the performance of presumed commit protocol. In: Proc. of ACM Symp. on Applied computing (SAC). New York, NY, USA, ACM Press (1997) 131–133
14. Attaluri, G.K., Salem, K.: The Presumed-Either Two-Phase Commit Protocol. IEEE Transactions on Knowledge and Data Engineering **14**(5) (2002) 1190–1196
15. Cheung, S.: Java Transaction Service Specification. Sun Microsystems Inc., San Antonio Road, Palo Alto, CA. Version 1.0 edn. (1999)
16. Garcia-Molina, H.: Using Semantic Knowledge for Transaction Processing in a Distributed Database. ACM Trans. on Database Systems (TODS) **8**(2) (1983)
17. Serrano-Alvarado, P., Rouvoy, R., Merle, P.: Self-Adaptive Component-Based Transaction Commit Management. In: 4th Middleware Workshop on Adaptive and Reflective Middleware (ARM). Volume 116 of AICPS. Grenoble, France, ACM Press (2005) 1–6
18. Chrysanthis, P.C., Samaras, G., Al-Houmail, Y.: Recovery and Performance of Atomic Commit Protocols in Distributed and Database Systems. In: Recovery Mechanisms in Database Systems. Prentice Hall (1998)
19. Liu, M.L., Agrawal, D., Abbadi, A.E.: The Performance of Two Phase Commit Protocols in the Presence of Site Failures. Kluwer Academic Publishers Distributed and Parallel Databases (DAPD) **6**(2) (1998)

20. Rouvoy, R., Merle, P.: Towards a Model Driven Approach to Build Component-Based Adaptable Middleware. In: 3rd Middleware Workshop on Reflective and Adaptive Middleware (RAM). Volume 80 of AICPS. Toronto, Canada, ACM Press (2004) 195–200
21. Rouvoy, R., Merle, P.: Using Microcomponents and Design Patterns to Build Evolutionary Transaction Services. In: Int. ERCIM Workshop on Software Evolution. Lille, France (2006) To appear.