Technical Report

TR-2007-025

Adapting Distributed Commit Protocol for Dynamic Metacomputing Frameworks

by

Pawel Jurczyk, Li Xiong

Mathematics and Computer Science

EMORY UNIVERSITY

Adapting Distributed Commit Protocol for Dynamic Metacomputing Frameworks

Pawel Jurczyk
Department of Math&CS
Emory University
pjurczy@emory.edu

Li Xiong
Department of Math&CS
Emory University
lxiong@emory.edu

Abstract

With increasing interest in applications which benefit from distributed computing paradigm and require at the same time to operate on data from heterogeneous data sources there is an increasing desire for commit protocols which can suit such systems. Well known two-phase commit protocol and three-phase commit protocol or its modifications do not provide flexibility which is desired for wide area of potential applications. In this paper we present an adaptation of three-phase commit protocol for dynamic distributed systems. The protocol is not only resistant to nodes/network failures, but also provides good scalability. Our protocol offers users a set of parameters which can be tuned and, depending on specific requirements of the system, guarantee consistency at desired level for the best price.

1 Introduction

An emerging class of distributed data intensive applications rely on large scale distributed and heterogeneous data sources; examples are enterprise end-system management, workflow management, and computer-supported collaborative work. Consider a nation-wide IT network provider that owns hundreds of thousands of network devices across the country. The devices produce hundreds of megabytes of data every hour (such as alarms and maintenance events). It is nearly impossible to store the data in one, nation-wide database server. Instead, hundreds of smaller servers with potentially heterogeneous database systems are used where each of them connects to local devices and stores information reported by them. In order to develop applications such as an enterprise-scale device management system or a report generation tool, data from multiple distributed and heterogeneous sources must be queried, integrated and possibly modified. The goal can be obtained using distributed metacomputing paradigm. As a consequence, the framework used to built such systems has to address multiple issues.

Scale. The scale of the applications we consider requires good performance and scalability without sacrificing data integrity. At one end of the spectrum, earlier distributed database systems [15], such as SDD-1, R* and Mariposa, share modest targets for network scalability (a handful of distributed sites). They focus on making distribution transparent to users and applications and encapsulating distribution with ACID guarantees for transaction. At the other end of the spectrum, Internet scale query systems, such as Astrolabe [28] and PIER [9], target a very large scale (thousands if not millions of nodes) but sacrifice on consistency semantics and data update functionalities. The scale of the applications we consider falls somewhere in the middle of the spectrum. This requires not only geographic scalability that allows systems scales to massively distributed geographic patterns but also load scalability that allow the system to expand or contract its resource pool to accommodate heavier or lighter query loads. Geographic scalability has been the focus of most Internet scale distributed systems that developed efficient distributed indexes and query routing schemes to address massively distributed data sources. The metacomputing paradigm originated from distributed systems community offers a cost effective way to provide load scalability where nodes can share resources and form a virtual supercomputer for both query processing infrastructure and application itself. The distributed commit protocol for applications we are considering requires unique characteristics. One of the main features from the point of scalability is a possibility of adjusting to the scale of the system which can vary from just few to hundreds of nodes.

Network and Resource Dynamics. Another characteristics of the large scale distributed applications in the wide area networks is the dynamic network conditions (such as node availability and network latency) and resource conditions (such as load of the nodes). In the area of data operating systems this has a few implications on the distribution of query processing or data operations. Conventional query processing strategies [15] will not work well because of their as-

sumption of constant throughput and homogeneous or constant network delay (e.g. they only consider data size to reduce data shipping). Also, conventional distributed commit protocol algorithms will fail if attempted to be used. In order to guarantee the usage of resources in the most efficient way and to achieve the desired scalability, techniques of operating on data have to adapt to network and resource dynamics. As far as commit protocol is concerned, it is required to handle scenarios of network or node failures. It is desired not to block whole protocol when one single node becomes unavailable for longer time but rather to unlock resources as soon as possible to increase system throughput for pending transactions.

Support for data operations. Most of applications do not only select data from database, but also requires to update, delete or create it. Applications using metacomputing paradigm are not an exception. Therefore, support for data operations is a must for any data operating middleware. As mentioned before, Internet scale systems generally sacrifice data update functionalities. On the other hand, classical distributed database systems offer ACID guarantees for transaction support. However, those systems do not scale well and distributed commit protocols used there are not suitable for larger scale dynamic environments with different characteristics.

Considering wide area of potential systems we are considering, we formulated three basic requirements which have to be satisfied by commit protocol that can be used. In our opinion, key features include (1) *non-blocking* approach, (2) resistance to *dynamic environment* and (3) possibility of *adjusting required consistency level*. The answer to variable systems scale and dynamics of nodes and interconnection network are both, non-blocking approach and resistance to dynamic environment feature. When some nodes become unavailable, especially in larger systems, the protocol should continue on available ones and finish as soon as possible. As the locks held by transactions on active nodes are released, further operations can be performed what increases the system utilization and availability. Of course, nodes which failed will have to perform recovery procedures which can guarantee data consistency; however, we strongly believe that the new approach to distributed data systems and possibly wide area of potential systems needs to leave decision about the consistency level on users site. Therefore, the users can specify whether the operation they want to execute requires full consistency, or if they are willing to take the risk of inconsistent state in case of some failures. One could of course argue that commit protocol has to offer full consistency at all times. In our opinion, however, the broad area of potential systems usage makes such a claim improper. Intuitively, guarantee of full consistency costs much more than guarantee of partial consistency. This fact has immediate result on the performance of the system and response time of operations and queries. In some cases, the consistency requirement is a must (e.g. implementation of distributed systems for a bank or financial institutions). On the other hand, implementation of the system for analysis purposes will not suffer from relaxed consistency and it is probably more beneficial to offer more efficient operations at the cost of possibly inconsistent state when some rare failures are met. Moreover, different deployment environments can lead to different conclusions. When an application is deployed on enterprise-scale highly reliable machines, the possibility of failures is very small. In such a scenario, it is again more beneficial to limit the cost of executing operations and use protocol which might not offer full consistency in case of extremely rare failures.

Contributions. In this paper we present a distributed commit protocol that can be used in frameworks which address wide variety of applications. First, the protocol addresses both, small scale systems with couple of nodes and larger solutions which operate on hundreds of nodes. Second, it is resilient to network partitioning and multiple nodes failures. Finally, the protocol is flexible in the level of provided consistency. Such an approach helps to adjust the parameters to guarantee consistency guarantee for systems with different characteristics at the lowest cost. We would like to highlight that the solution we are providing is not specific to data operations or metacomputing systems. In general, our protocol is applicable to any distributed environment where requirement for atomic commit is present.

The protocol we offer has been implemented as a part of DObjects data query infrastructure for metacomputing frameworks. In the consequence important contribution of this paper is also that we provide along the lines easy to use distributed framework which provides not only a query infrastructure, but also data operations capabilities. The framework is complete, it is integrated with metacomputing infrastructure (i.e. it can benefit from resource sharing process) and can be easily used in applications benefiting from metacomputing paradigm.

Finally, we present deep analysis of our protocol. We analyze different settings of its parameters, describing consistency level provided by each of them. Moreover, our experimental analysis which uses both real protocol implementation and simulation environment gives good idea of cost that needs to be paid when different parameter settings are used.

Organization. Remaining part of this paper is organized as follows. Next section discusses related work to our problem. Section 3 presents our distributed commit protocol followed by its analysis (Section 4). Section 5 describes DObjects framework that can be used to build distributed systems based on metacomputing paradigm and requiring to operate on distributed data sources and that was extended with protocol presented here. In Section 6 we present exper-

imental evaluation of suggested protocol and we conclude in section 7.

2 Related work

Our work on adapting commit protocol for metacomputing frameworks was inspired and informed by a number of research areas.

Distributed system architectures Distributed systems have been a subject of research for quite a long time. Distributed computing paradigm offers great scalability at relatively low price by sharing distributed resources to solve large-scale computing problems. With many distributed computing architectures (e.g. client-server or P2P) and platforms (BOINC [3], Unicore [23], Globus Toolkit [7] just to name few) one can find framework which is the best suited for given problem. The variety of available solutions offers many approaches to the resource sharing process. Some systems are assumed to be run on a volunteer basis and can be treated as donating one's unused computational power to work on interesting computational problems. Such an example is BOINC [3] platform which is used to solve computationally intensive research problems. Other systems are more rigorous about the participants of computing network. For instance grid systems and such platforms as Globus Toolkit [7] and UNICORE [23], provide general frameworks for running software on grid architecture using different approaches such as component-based model [4] and distributed objects paradigm [29]. However, they require centralized services for authentication or job submission which limit these solutions.

The issue of centralized services was the main reason for emerging of a decentralized metacomputing platforms, such as H2O [17, 11]. This class of frameworks avoids the administrative burden related to using grid systems and makes resource sharing easier for providers, developers and resource users, in the spirit of the P2P model.

Distributed computing is also possible without using frameworks mentioned before. DCOM[1], CORBA[2] or RMI[3] offer the distributed object paradigms which can be used to built distributed systems. However, those solutions only allow objects to be distributed across a heterogeneous network whilst for instance H2O is a complete resource sharing platform which is built on top of distributed objects paradigm.

Distributed commit protocol Distributed commit protocol has also been research subject for quite a long time. Among a variety of distributed commit protocols the most popular seems to be two-phase commit protocol (2PC) presented in [18] or three-phase commit protocol (3PC) described in [25]. The phases in two-phase protocol are commit-request phase (in which the coordinator prepares all the cohorts to commit transaction) and commit phases (in which uniform decision is made by the coordinator and transaction is either committed or aborted). There are two most common variations of 2PC protocol which are presented in [21]. First of them named presumed abort assume that the transaction aborts if it is not explicitly committed; in such an approach abort messages do not need to be acknowledged by cohorts. The second variation is named presumed commit and it assumes the transaction commits if it is not explicitly aborted. In this approach commit messages do not need to be acknowledged, however, extra log information about prepared cohorts is required on coordinator site. As most transactions are assumed to commit, the second approach has an advantage. The problem with 2PC is that it is blocking solution, i.e. all nodes will keep their locks while waiting for the messages. Therefore, in case of node failures, active nodes will keep waiting and the locks will not be released. As can be noted, such an approach can be applied for rather static distributed databases as the protocol will not handle well cases when nodes join or leave the network. The solution to this problem was three-phase commit protocol (3PC). The protocol originally extended idea of the 2PC by adding third phase which is placed between commit-request and commit phases. The phase is called pre-commit which informs cohorts about the *early* decision of coordinator. The 3PC has additional transitions which are performed in case of timeouts. They guarantee that in case of failures the protocol would release locks even in case of node failure.

The basic three-phase commit protocol is not resilient to site failures, lost messages, and network partitioning. The solution to those issues was quorum-based commit protocol (Q3PC) that can maintain consistency even in the case of network partitioning and was proposed in [24]. The protocols do not require that a failure to be correctly identified or even detected, however, at the cost of protocol performance (and possibly blocking behavior). If network partitioning occurs, connected nodes construct quorum and decide whether they can solve the transaction. Later, further modifications were suggested to the original Q3PC. As there are some cases when Q3PC blocks quorum (e.g. in case of cascade failures), enhanced quorum-based commit protocol (E3PC) was presented in [13]. The main idea of the protocol is to maintain two additional counters which guarantee that connected quorum never blocks. The observation that E3PC tends to decide to abort transactions in many of those quorate failure sequences where Q3PC would block has led to improved version of E3PC presented in [13]. The new protocol makes progress towards committing a transaction during recovery in circumstances where E3PC would abort.

Authors of [19] presented an interesting approach to efficient commit protocol for distributed main memory databases. The core idea is to limit number of sent mes-

[1] http://msdn2.microsoft.com/en-us/library/ms809340.aspx
[2] http://www.corba.org/
[3] http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

sages. In the proposed protocol instead of sending and receiving two sets of messages one after the other as in two phase commit, only one set of messages is sent after the coordinator completes committing a distributed transaction. An interesting enhancement to 2PC protocol was introduced in [30]. The paper presents approach of replicating the coordinator and running a Byzantine agreement algorithm among the coordinator replicas with the goal of dealing with faulty coordinators sending conflicting decisions to different participants.

Even though the distributed commit protocol seems to fit distributed databases area the best, the distributed database systems do not provide wide range of commit protocol solutions. In most cases, 2PC and its variations were implemented (R* system [21] and INGRES [27]). In many other projects, either the ACID requirement for transactions was sacrificed as a trade-off for a very large scale (PIER [10, 9]), or the system was mainly prepared for selecting data. Thus, such platforms as Seaweed [22], HyperQueries [12], Active XML[4] or Mariposa [26] provide a good architecture for processing queries, but do not provide features of data modification.

Fault tolerance in distributed systems. Another body of work related to our research is fault tolerance in distributed systems. With increasing popularity of grid computing, a problem of grid service reliability was brought up in [6]. The paper provides an analysis of reliability in the context various possible failures such as time-outs, network failures, program or resource failures. Such papers as [1, 5] were aimed at researching dependable storage systems relying on data replication. One of the most popular replication technique are quorum based protocols [8]. Such protocols assume that replication allows to perform reads and writes only on a subset of all available servers (quorum). Byzantine quorum systems [20] were developed to allow tolerance of Byzantine server failures. Research work presented in [14] attempts to increase efficiency of Byzantine quorum systems by using optimistic quorum collection and adaptive probabilistic cashing. Another idea for reducing overhead of Byzantine quorum protocols was presented in [16]. The problem of detecting faults in Byzantine quorum systems was investigated in [2]. The authors propose statistical approaches for estimating the risk posed by faulty servers.

3 Commit protocol

According to our earlier discussion, a key features for distributed commit protocol suitable for wide range of distributed applications are (1) non-blocking approach, (2) resistance to dynamic environment and (3) possibility of adjusting required consistency level. In this section we will provide description of protocol which satisfies all those requirements. We will start our discussion from analysis of classical *three-phase commit protocol*.

3PC and problems. Now, as we know what requirements need to be met by commit protocol suitable for dynamic metacomputing framework, let's take a look at existing solutions. 2PC cannot be used efficiently as it is blocking protocol. Our requirements clearly state non-blocking functionalities. 3PC protocol presented in Figure 1 is much better. It solves the blocking behavior by adding failure and timeout transitions along with extra phase in which coordinator informs cohorts about early decision of the transaction outcome. The protocol proceeds as follows. First coordinator sends commit request to all cohorts. If all cohort agree to transaction, coordinator moves to next phase. If any cohort does not agree, whole transaction is aborted. Next, after all the cohorts agreed to the transaction, coordinator sends prepare to commit message (pre-commit phase) to cohorts. After all the cohorts acknowledge receiving of the prepare message, coordinator sends commit messages and commits. The basic version of 3PC does not support, however, *network partitioning* or *multiple nodes failure* cases. If any of these is met, the protocol can end up in inconsistent state. For instance, when network partitioning occurs when coordinator starts sending pre-commit messages, nodes which received pre-commit would commit whilst others would abort. Moreover, it assumes atomic transitions from one state to another. As a consequence the message pre-commit can either be sent to all the cohorts or to none of them. Such a requirement is often hard to implement in various distributed computing paradigms without a dedicated hardware or operating system support. By extending the 3PC, such protocols as Q3PC or E3PC or X3PC, attempt to deal with those issues. The protocols are based on the idea of quorum creation which decides about unresolved transaction by communication among participating nodes at the cost of blocking approach. When quorums are created, site will hold unresolved transactions until the network or node failures are fixed. Even though those protocols can address some of requirements for our protocol, still the parametrized consistency level was not addressed.

3.1 Protocol proposal

The protocol which we are proposing is a modified version of basic 3PC. The modification attempts to address three key issues: (1) resistance to multiple nodes failure or network partitioning, (2) flexibility of consistency level requirement and (3) support for partial transitions from one state to another. We analyzed basic 3PC and observed that the efficiency of operations is reflected by multiple factors. First, the impact on efficiency has logging. As each log has to be forced to persistent storage before the protocol proceeds, such operations are expected to have impact on the response time of the commit protocol (especially for tightly

[4] http://activexml.net/

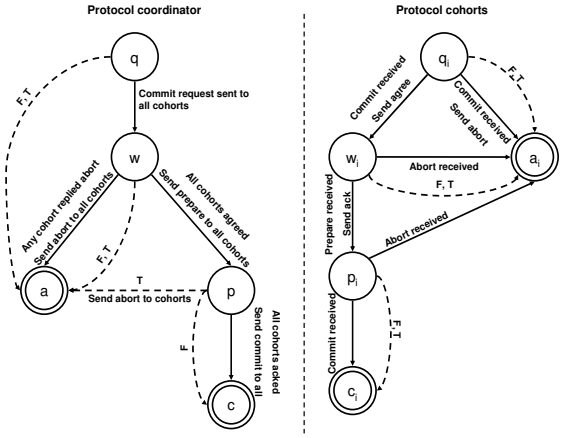

Figure 1: Basic 3PC protocol.

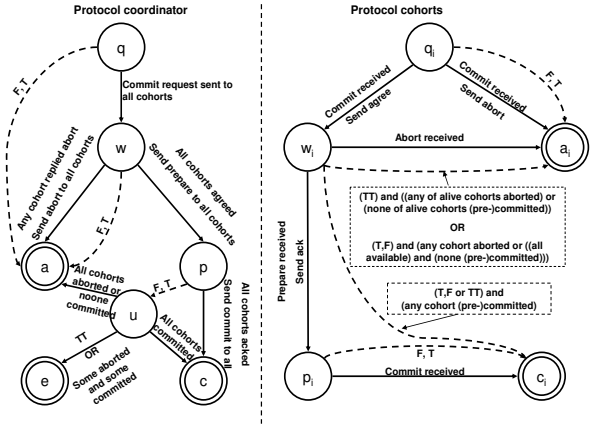

Figure 2: Commit protocol. T is timeout waiting for next message, TT is *transaction timeout*.

coupled distributed systems connected by very fast connection backbone). The amount of logged information also impacts the response time. Second, the timeout parameter in 3PC (i.e. the time after which timeout transition is followed) impacts possibly other transactions or queries which are trying to access given data items due to locks held by active transactions. The longer the timeout is, the longer many data items can be locked by transactions which harms especially wider systems (e.g. systems offering statistical analysis) where consistency is not the matter of such an importance. Finally, as we mentioned before, the requirement of atomic transitions is hard to implement in many distributed metacomputing systems.

Assumptions. We assume that our implementation of commit protocol uses nodes which provide interface being able to perform operations in the following way. Operations are executed and later committed or rolled back. When our algorithm successfully executes operation on given interface, it guarantees that the operation can be locally committed. Later, the algorithm can either commit or roll back the operations. However, when node failure occurs after operations were executed, but before committed or rolled back, the state of operations is unidentified (e.g. interface does not have to provide any guarantee here).

Protocol overview. To solve problems described above we are proposing an adaptation of three-phase commit protocol for highly distributed and dynamic environments. Our main idea is to make the algorithm configurable so that, depending on specific deployment characteristics of the system, users can tune it to achieve the best efficiency at the lowest cost in terms of system resources. As we pointed above, amount of log data and timeout value both have impact on the efficiency. Therefore, our commit protocol has two parameters, log level and transaction timeout.

Log level. The log level switch allows to switch users between the amount of logged information. Three options are considered, namely no logging, optimistic logging and full logging. In no logging case, each node does not log any information to persistent storage. When option of optimistic log level is enabled, algorithm saves in persistent storage only information about crucial steps of protocol (such as commit request, prepare or commit). However, no information about operations being performed is saved. In this case recovery options are limited. When node fails after operations were executed on operations interface but before they were committed, it can obtain the state of transaction it was participating in. However, it does not have information about operations which should be executed to undo/redo the transaction (please remember that our assumptions do not guarantee anything for executed operations but not committed/rolled back). Option of full logging saves to persistent storage both, steps of protocol and undo/redo information. In this case, each node has full information that enables it to undo or redo any operations even if it fails before committing or rolling back operations on operations interface. We should mention that it is assumed for each party to use write ahead log (WAL), e.g. information is logged on persistent storage before any actions on local operations interface are performed.

Transaction timeout. The parameter that defines max time unresolved transaction can persist in the system. When timeout is reached, decision is made even if some participants of the commit protocol are not available. Of course such an approach can lead to inconsistent decisions which have to be solved by users (for instance by resubmitting the

transaction after the system was repaired). When timeout is set to infinity, the transaction will not be finished before the full agreement is reached.

Figure 2 presents modified 3PC protocol we are suggesting. Our assumption is that, when transition is performed, operations associated with this transition are executed in new state. However, the operations do not have to be executed atomically. For instance, in transition between states *w* and *p* of coordinator, sending prepare messages starts *after* the transition to *p* is performed and coordinator can fail after sending any number of messages (including none). Reader should be also aware of the fact that two types of timeouts are used. First, timeout *T*, means timeout when waiting for message. The second, *transaction timeout* (abbreviated as *TT*) which represents parameter of our transaction protocol. Coordinator site of the protocol introduces a new state called *u* representing *unknown outcome* of the transaction. The state solves following issue. When coordinator is in state *p*, it can start sending pre-commit to cohorts. When failure of coordinator occurs just after sending few messages, but not all of them, exact outcome of transaction is unknown (it could not happen in original 3PC because of transaction atomicity assumption). In this case, cohorts can achieve full or partial agreement which depends on *transaction timeout* and *log level* parameters. Therefore, when all cohorts committed, transition to state *c* is followed and transaction commits on coordinator site. On the other hand, when all cohorts aborted, transaction aborts. Finally, when some cohorts committed, some aborted or when some cohorts are not available and the transaction timeouts, then user has to be notified about the error while executing the transaction (state *e*). In this case, additional steps have to be performed by user (e.g. transaction has to be resubmitted).
Cohort's site of the protocol is also quite similar to the solution presented in the original 3PC protocol. The only modification was required when cohort is in state *w*. In the original protocol, both failure and timeout transitions lead to the *a* state. However, as we allowed non-atomic transitions in coordinator, such an approach is impossible. In our approach only some cohorts could be notified with pre-commit or abort message (e.g. in case of coordinator failure during sending those messages) and some cohorts have to make final decision with respect to possible decisions of others. Thus, when *T* or *F* occurs at state w_i, and when all cohorts are present, transaction on cohort can be committed or aborted - depending on state of other cohorts. The approach here is somehow similar as in quorum-based protocols. If *TT* occurs, max allowable time for transaction has expired and decision has to be made even when not all the cohorts are available.
A word of explaining is required for the idea of timeout *T* in cohort's state w_i. *T* can be treated as a timer - when it expires, protocol checks the conditions. If conditions of some transition in the graph are satisfied, it is followed. If no conditional transition is satisfied, protocol stays in the same state and resets the counter.

4 Protocol analysis

In this section we will analyze the algorithm we presented above. We will start from proving that for certain parameters configuration one can achieve full consistency requirement. Next, we will discuss different configurations and consistency levels they offer.

4.1 Informal proof of algorithm correctness when *full logging* and *infinite transaction timeout* are used

Claim 1 When *full logging* and *infinite transaction timeout* are used, all cohorts commit or abort transaction.

We will prove *Claim 1* by contradiction. Let's assume that, the protocol can end with inconsistent decision among cohorts. In case of no-failure operation, the inconsistent decision would require that some nodes obtained commit and some obtained abort messages. However, such a situation is not possible, as either all cohorts are notified with pre-commit and commit or with abort message and once the coordinator makes decision to commit the transaction, this decision is not changed.
Coordinator failure. Now, let's assume that coordinator failed before sending commit messages to cohorts or after sending only few pre-commit messages and in consequence no cohort received commit message and some received pre-commits. For cohort, which have to be either in state w_i (if pre-commit was not received) or p_i (if pre-commit was received), the only way of reaching commit state without commit message is to follow timeout or failure transition from those states to *commit* state. If current state is q_i, we know that cohort wanted to commit and transaction is committed. Now let's take a look at cohorts in state w_i. As TT is set to infinity, the TT part of condition in transition to *abort* state will never be satisfied. The second part of this condition can only be satisfied if any cohort received abort (not possible as coordinator failed before sending commit message, but after sending pre-commits) or non of cohorts involved in transaction received pre-commit or commit. Clearly, such condition guarantees that if any cohort received commit or pre-commit, others will not abort but will eventually follow failure or timeout transition from state w_i to c_i and commit the transaction.
When coordinator decides to abort transaction, it starts sending abort messages to cohorts. When such a decision is made, there is no possibility of changing it (when coordinator is in state *p* meaning it has decided to start sending pre-commits, it can follow only to *c* or *u*; abort message is not sent to cohorts). If abort message is delivered to any cohort, or if none receives pre-commit or commit, the protocol guarantees that all other cohorts would follow to *abort* states.

Cohort failures. Let's now take a look at failures on cohort sites. If cohort fails in state q_i, it follows to *abort* state immediately. Moreover, when any of cohorts is in the state above, coordinator is in state *w* or in *q* and in case of failure or timeout it follows to *abort* state. Therefore, both coordinator and failed cohorts abort transaction. Cohorts which did not fail, can now be in either of states, q_i or w_i. As those cohorts would not receive further messages, they timeout and can only follow transaction to *abort* state as no pre-commit or commit was sent by coordinator. A bit different situation is encountered when cohort being in state w_i fails. In this case coordinator can be either in *w* (in this case no pre-commit was sent) or in *p* (some pre-commit messages might have been sent). If coordinator is still in w and fails, it moves to *abort*. Other cohorts will also eventually reach the abort state (including the failed cohort). If coordinator is in state p and it fails, it moves to u. However, some cohorts might have received pre-commit message and move to state p_i and eventually to $commit$ state. The failed cohort at state w_i after recovery contacts other cohorts what guarantees the consistency. Please observe that infinite *transaction timeout* guarantees that either cohort which knows the outcome of transaction has to be contacted or protocol waits for all cohorts to verify that outcome is unknown and aborts transaction. If pre-commit message was delivered to any cohort, whole transaction will be *committed*. If no one received pre-commit, the transaction will be *aborted*. Finally, as coordinator being in state *u* sees that all the cohorts either committed or aborted, he makes the final decision as well.
When cohort being in state p_i fails, it commits the transaction after recovery. When cohort is in state p_i, coordinator can be in state *p* or *c* and other cohorts can be either in state w_i or q_i. If coordinator being in state p fails, it failed either after sending at least some pre-commit messages (as at least failed cohort received this message). When other cohorts which have not received pre-commit message timeout from state w_i, they will verify state of other cohorts and eventually *commit* as *transaction timeout* being set to infinity ensures that at least one cohort aware of outcome or all cohorts are required for making final decision. If cohort being in state q_i fails and coordinator being in state c fails, all cohorts must have received the pre-commit message and are in states q_i. Therefore, all the cohorts follow failure or timeout transactions and *commit*. One aspect needs explaining. When cohort being in state w_i fails, it can either commit or abort the transaction. However, as full logging option was used, cohort have all the information to perform correct actions.
By the discussion above we showed that, it is not possible to reach inconsistent decision among cohorts. Therefore, the assumption of the hypothesis was wrong and algorithm does guarantee the uniform decision. What is not surprising, the protocol with described parameters is blocking. If failure or network partitioning occurs, the protocol waits until the problem is fixed. However, such a behavior was expected as Skeen proved in [25] that there does not exist a non-blocking commit protocol resilient to multiple nodes failure or network partitioning.

4.2 Analysis of the protocol for various parameters configuration

We will now discuss different parameters configuration and take a closer look at consistency level they offer. Previous section proved that when *full logging* and *infinite timeout* are used, the protocol guarantees full consistency. Let's analyze what happens when user attempts to change a log level.
No log information. In case of no logging situation is relatively simple. In case of node failure and restart, no information about the state of transaction is available. In fact, such a node even does not know the transaction existed. We assume that when no-logging was used, no failure transition can be followed. Even in such a case there are some scenarios which lead to consistent state. For instance, when coordinator fails before sending commit messages but after sending all the *pre-commits*, all cohorts would time out from states p_i and the transaction would be committed among all other cohorts. There are many cases, however, which can lead to inconsistency. One of examples is as follows. Let's assume coordinator starts sending pre-commit messages and it fails after the first one is sent (when coordinator fails and no logging option is used, it does not follow any failure state and simply forgets about the transaction). Next, cohort which received pre-commit waits, times out to *commit* and fails forgetting about the transaction. As other cohorts also timeout, they start to ask others about the outcome of transaction. When failed cohort recovers, however, it does not have any information about the past transaction so it cannot inform cohorts about the fact it committed transaction. In consequence, other cohorts abort and the state is inconsistent.
Optimistic log. When optimistic log is used, it is enough to solve the inconsistency issue described above. In this case, each node participating in transaction leaves transaction marks. Failed cohort above would know after the restart that the transaction above was committed and he would notify other nodes about it. Therefore, other cohorts pending in state w_i would follow transition to commit state and consistent state is maintained. Even though the optimistic logging solves some problems, it still can lead to inconsistency. It is enough to continue discussion above and add failure of cohort which waits in state w_i. Even though the node has the knowledge of transaction and its state, according to our assumptions it has to undo or redo actions depending on the transaction outcome. However, as such a knowledge is not available, transaction would become inconsistent. Such a

problem is solved by *full logging*. We will not continue the discussion here as this scenario has already been covered by subsection 4.1.

Transaction timeout. The discussion above assumed that *transaction timeout* parameter was not altered (e.g. infinite value was used). At this point we will take a closer look at what happens when user decides to change it and specifies fixed value. The transaction timeout limits amount of time that nodes can wait for being able to contact others in case of network partitioning or node failure. Let's first consider scenarios when no node failure occurs. When network partitions and some cohorts which are in state w_i become unreachable from coordinator, those cohorts will simply wait for messages. When none is received, they face time out and attempt to contact others to find the outcome of transaction. However, if network failure is not fixed before *transaction timeout* is reached, decision can be made without respect to some of the cohorts. This can lead to inconsistency. For instance, when coordinator sends pre-commit messages and network partitioning occurs, the cluster of connected nodes which includes the cohorts which received pre-commit would commit whilst clusters which did not receive any pre-commit would abort.

Let's now consider a case of node failures. For each of discussed log levels we showed some scenarios which lead to consistent states. Those scenarios remain valid when *transaction timeout* is used. This parameter simply limits the waiting time for nodes to be restarted. If failed node is restarted before *transaction timeout* occurs, the scenario will be followed and consistency will be maintained to the extent provided by given log level. If the node is not restarted before the *transaction timeout* occurred, some cohorts can make decision without respect to the state of transaction on failed cohorts and inconsistent state can be reached.

4.3 Summary of the parameters configuration

Table 1 contains a summary of available parameters for our commit protocol. We included all possible configurations and described consistency level that can be reached by using them. Reader can notice that when *full logging* and *infinite transaction timeout* are used, full consistency is guaranteed at the cost of blocking approach. However, in case of altering the parameters, protocol is resistant to particular type of failures, but some scenarios can lead to inconsistency.

5 DOBjects - a metacomputing framework for distributed data systems

Our initial implementation of the protocol presented above was provided for DObjects system. In this section we will provide readers with more details about this framework. DObjects is general purpose framework that exploits distributed metacomputing paradigm. We will discuss architecture of the framework and data operations which are available for users.

	No log	**Optimistic log**	**Full log**
Finite timeout	Consistency if protocol finishes before timeout (nodes failure cause inconsistency). Possibly no information for users about the outcome of transaction	Consistency if protocol finishes before timeout (nodes failure can cause inconsistency)	Consistency if protocol finishes before timeout
Infinite timeout	Resistant to net. failures, nodes failures lead to inconsistency. Possibly no information for users about the outcome of transaction	Resistant to net. failures, nodes failures can lead to inconsistency	Full consistency, prone to net. and node failures, blocking protocol

Table 1: Summary of available consistency level depending on transaction parameters

5.1 System Architecture

Figure 3 presents a vision of deployed DObjects framework. The system has no centralized services and thus allows system administrators to avoid all the burden in this area. It uses the *metacomputing* paradigm as a resource sharing substrate. Each node in the system provides its *computational power* that can be used by others during query execution. In addition, nodes can run *data adapters* which pull data from external data sources and transform it to a uniform format that is expected while building query responses. Front-end users can connect to any system node; however, while the physical connection is established between a client and one of the system nodes, the logical connection is established between a client node and a virtual database system consisting of all the participating nodes.

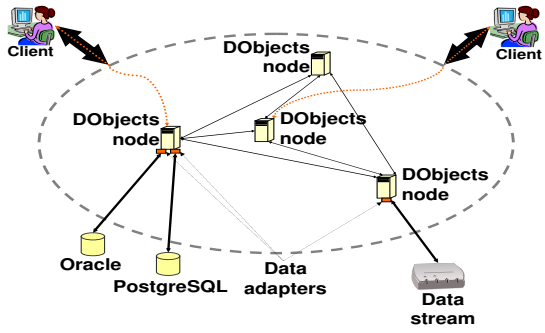

Figure 3: System Architecture

Current implementation builds on top of a Java metacomputing platform, H2O[11, 17], that provides lightweight, decentralized and peer-to-peer resource sharing and communication. Data adapters are implemented

using a Java object/relational mapping API, Hibernate[5], to pull data from heterogeneous relational data sources such as MySQL, PostgreSQL, Oracle etc. Adapter interface can be also implemented for any abstract sources providing data (for example, data file, network device or any system device driver).

5.2 Data Operations

To support various applications, DObjects supports *ad-hoc queries*, *batch queries* and possibly in the future *continuous queries*. In case of an *ad-hoc query*, the system provides response immediately after its completion and execution of user's code is blocked until the query is finished. As the opposite, batch queries are executed in background. When a *batch query* is executed, user's code is not blocked and notifications about results are provided on a given listener. What is important, user can get results *incrementally* and operate on partial results while the query is executed. The idea of *continuous queries* that remain in the system until explicit termination will be based on the following idea. Whenever new data comes from adapter, system can determine whether it is potential answer to the standing queries. If so, the response is prepared and sent to appropriate users.

The system offers data in the form of data objects which form a hierarchy. Therefore, the query language for the system could be implemented using any language that allows one to specify populated attributes or conditions for given attribute in objects hierarchy. XPath or XQuery as well as SQL-like (or OQL-like) language are all valid approached. In its final form DObjects will provide SQL-like (or OQL-like) query interface as this family of languages is probably the most common. Current stage of DObjects does not provide, however, parser for any of mentioned solutions. The system provides its internal query language which builds on top of Java implementation.

Through implementation of the commit protocol presented in this paper, the system was extended with capabilities of creating, deleting and updating data in data sources with transaction semantics. Moreover, for each executed operation user has a chance to decide through transaction parameters what consistency level is desired in case of nodes or network failure. The section below presents an evaluation of the algorithm.

6 Experimental evaluation

In this section we present experimental evaluation of distributed commit protocol described in previous sections. The evaluation is divided into two parts. The first part presents results obtained from evaluating implementation of the commit protocol for DObjects framework. The second part of evaluation presents simulation results.

The experimental implementation of the algorithm was performed for DObjects system described in Section 5. As DObjects is framework that supports data operations, transaction management extended the system with operations of creating, deleting and updating data. This part of experimental evaluation is mainly focused on performance analysis and presentation of log level impact on commit protocol efficiency.

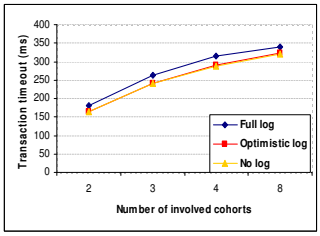

Figure 4: Transaction response time (create operations)

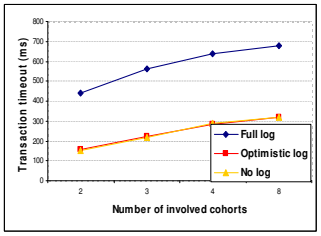

Figure 5: Transaction response time (delete operations)

Our first experiment was focused on the analysis of transaction response time. We show results for two data operations, namely create and delete operations. The experiment involved 8 system nodes and was deployed on modern general purpose PCs (Intel DualCore CPUs, 1GB RAM). The interconnection network was FastEthernet. Figure 4 presents response time for create and Figure 5 response time for delete operations. What noticeable, our assumptions were proven, reduction of log level from full log option to optimistic log gives better performance of the transactions. Comparison between optimistic log and no log options gives conclusion that there is very small difference in transaction duration. Reader can observe huge disproportion between comparison of full log and optimistic log options between create and delete operations. The phenomenon needs explaining. When full log for create operation is generated, UNDO operation requires simply create operation and REDO log needs delete operation. Importantly, delete operation requires only the knowledge of object id, which, in current implementation of the protocol for DObjects system, is known from create operation directly. Therefore, no extra information is needed. A bit different situation is met when full log is created for delete operation. In this case implementation of the protocol offers only id of deleted object. Such an information is enough to create UNDO log. REDO log, however, requires create operation of possibly deleted object. In this case information about whole object is needed and it can be obtained only after querying the database what explains the longer operation duration.

The next experiment involved transaction throughput. We measured average transaction throughput for different number of independent clients. Operations were divided between create and delete operations. The results are pre-

[5]http://www.hibernate.org/

Parameter name	Description	Default value
failProb	Probability of coordinator failure	0.01
lockProb	Probability of that given transaction will have to wait for another transaction due to locks owned by that transaction	0.005
tTimeout	Transaction timeout (parameter of the commit protocol)	Variable
failD	Average duration of cohort failure	100000 epochs

Table 2: Simulation parameters

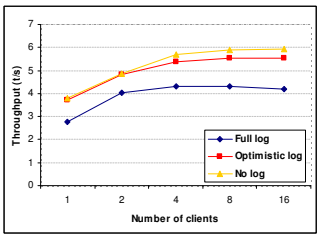

Figure 6: Throughput of transactions when three cohort nodes are involved

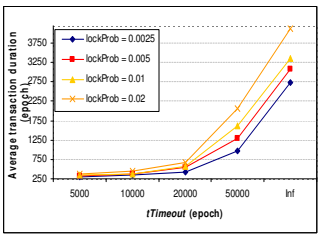

Figure 7: Transaction durations for simulation results (variable *tTimeout* and *lockProb*).

sented in Figure 6. As expected, limiting amount of logged information led to better throughput. Moreover, now one can observe difference between optimistic logging and no logging options. Especially for larger number of clients, when the system becomes overloaded, no logging shows better performance. Such a phenomenon is not surprising, as writing even very small information to persistent storage (e.g. hard drive) is considerable expensive operations, especially in case of high hardware utilization.

Transaction timeout parameter has large impact when nodes fail or leave network. To verify the impact of this parameter we have prepared a simulation of the protocol on discrete event simulator. Our simulation provided few configuration parameters which are described in Table 2. The results we obtained for 40 cohorts and 5 coordinators are presented in Figures 7, 8 and 9. In our scenario each coordinator submits transactions one by one, and each transaction involves random subset of 50% of all the cohorts. Please note that in our simulation only coordinator nodes can fail - this is more interesting scenario for evaluation of *transaction timeout* parameter. The first Figure presents transaction duration as a function of *tTimeout* and *lockProb* parameters for default value of *failProb* and *failD*. Reader can observe that the higher *lockProb* is, the higher transaction duration. Intuitively, such a behavior is expected as when more transactions have to wait for other transactions, the longer they will last in the system. What is noticeable, when *tTimeout* increases, transaction response time increases rapidly. Especially, for infinite value of *tTimeout*, which in practice means that transactions will have to wait until the failed

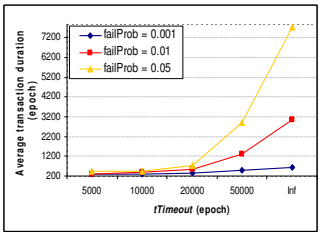

Figure 8: Transaction durations for simulation results (variable *tTimeout* and *failProb*).

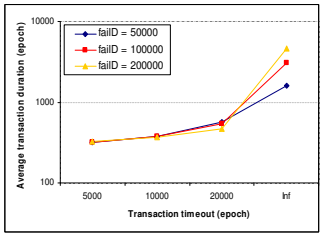

Figure 9: Transaction durations for simulation results (variable *tTimeout* and *failD*).

node was repaired, transactions take much longer.

The second simulation experiment presented in Fig. 8 uses default value of *lockProb* and *failD* and varies the coordinator failure rate and transaction timeout. As we might expect, with increase of *failProb* and *tTimeout*, the transaction response time increases. Increased *failProb* causes more coordinator failures which causes longer locks for transactions and has direct impact on the transaction duration. With smaller value of *tTimeout* the transactions timeout faster and release kept resources. This allows other transactions to proceed sooner what has an effect in smaller response time.

The last simulation experiment presented in Figure 9 involved variable *tTimeout* and *failD* for default value of *lockProb* and *failProb*. The results seems to be quite surprising. When value of *tTimeout* increases, the transaction duration increases as well. When transaction timeout is finite, increase in average failure duration of coordinators causes transaction duration to reduce. For infinite value of *tTimeout*, however, the situation is completely different. In this case, the longer average duration of coordinator failure, the higher transaction response time. The phenomenon can be explained as follows. When *transaction timeout* is short enough, resources held by transaction submitted from coordinator which fails are released quickly. In this case, other coordinators can keep submitting the transactions and, as one of coordinators is down, the probability of transaction which have to wait for other transactions is reduced. Simply, the longer average duration of coordinator failure, the better for other coordinators. On the other hand when infinite timeout is used and failure of coordinator which holds locks occurs, the locks will not be released until the coordinator is repaired. During this time, other transactions which require locks on items being locked by failed coordinator cannot proceed. Intuitively, the longer average duration of failure is, the longer other transactions will have to wait.

7 Conclusion

In this paper we have presented a new approach to commit protocol. We have introduces a new protocol which is

based on 3PC and provides configuration parameters which can be used for consistency level adjustment. Our research was motivated by the fact that depending on specific system deployment characteristics, users are willing to alter the protocol in such a way it provides good consistency guarantee at the lowest cost.

Besides the protocol itself, we have presented a data operations metacomputing framework which implements it. The framework itself is worth of attention as it fully integrates with metacomputing system. Not only it uses distributed computing paradigm when queries or operations are executed, but users also can develop distributed applications which use shared resources. In this case, the data operations system and user's application both can benefit from distributed metacomputing paradigm in transparent way. Finally, our analysis of real implementation of the protocol and simulation results prove that our assumptions were correct. When users change parameters of the protocol, and allow inconsistency in some scenarios of failures, the system throughput changes. In this case, when parameters are chosen correctly to given system specific deployment characteristics, one can achieve the best performance and consistency level at the lowest price.

In the future work the fault tolerance area of the protocol will be explored. We are planning to introduce data replications and extend the system with tolerance of Byzantine nodes failures.

References

[1] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.

[2] L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):996–1007, 2001.

[3] D. Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, 2003.

[4] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren. The cca component model for high-performance scientific computing. *Concurrency and Computation: Practice & Experience*, 18(2), 2006.

[5] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 19–19, Berkeley, CA, USA, 2000. USENIX Association.

[6] M.-Y.-S. Dai, S. M.-Y. Pan, and M.-X. Zou. A hierarchical modeling and analysis for grid service reliability. *IEEE Trans. Comput.*, 56(5):681–691, 2007.

[7] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Lecture Notes in Computer Science*, 3779, 2006.

[8] D. K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM.

[9] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of pier: an internet-scale query processor. In *CIDR*, 2005.

[10] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, 2003.

[11] P. Hwang, D. Kurzyniec, and V. Sunderam. Heterogeneous parallel computing across multidomain clusters. In *Proceedings of 11th European PVM/MPI Users' Group Meeting*, LNCS. Springer-Verlag, 2004.

[12] A. Kemper and C. Wiesner. Hyperqueries: Dynamic distributed query processing on the internet. In *The VLDB Journal*, 2001.

[13] T. Kempster, C. Stirling, and P. Thanisch. A more committed quorum-based three phase commit protocol. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, pages 246–257, London, UK, 1998. Springer-Verlag.

[14] L. Kong, D. J. Manohar, M. Ahamad, A. Subbiah, M. Sun, and D. M. Blough. Agile store: Experience with quorum-based data replication techniques for adaptive byzantine fault tolerance. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 143–154, Washington, DC, USA, 2005. IEEE Computer Society.

[15] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32, 2000.

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2007. ACM.

[17] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam. Towards self-organizing distributed computing frameworks: The H2O approach. *Parallel Processing Letters*, 13(2), 2003.

[18] B. W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, London, UK, 1981. Springer-Verlag.

[19] I. Lee and H. Y. Yeom. A fast commit protocol for distributed main memory database systems. In *ICOIN '02: Revised Papers from the International Conference on Information Networking, Wireless Communications Technologies and Network Applications-Part II*, pages 691–702, London, UK, 2002. Springer-Verlag.

[20] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578, New York, NY, USA, 1997. ACM.

[21] C. Mohan, B. G. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[22] R. Mortier, D. Narayanan, A. Donnelly, and A. Rowstron. Seaweed: Distributed scalable ad hoc querying. In *ICDE Workshops*, 2006.

[23] M. Riedel and D. Mallmann. Standardization processes of the unicore grid system. In *Proceedings of 1st Austrian Grid Symposium 2005*, 2006.

[24] D. Skeen. A quorum-based commit protocol. Technical report, Cornell University, Ithaca, NY, USA, 1982.

[25] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *Concurrency control and reliability in distributed systems*, pages 295–317, 1987.

[26] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. A. Olson. Mariposa: A new architecture for distributed data. In *ICDE*, 1994.

[27] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[28] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2), 2003.

[29] Y.-J. Woo and C.-S. Jeong. Distributed object-oriented parallel programming environment on grid. *Lecture Notes in Computer Science*, 2668, 2003.

[30] W. Zhao. A byzantine fault tolerant distributed commit protocol. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, 2007.