A Formal Analysis of 2PC and related Optimisations

Tim Kempster, Colin Stirling, Peter Thanisch
Department of Computer Science, University of Edinburgh

Abstract

The Centralised Two Phase Commit Protocol (2PC) is the most widely used protocol in commercial distributed systems. Many variants of this protocol exist which try to reduce both the number of messages required to execution. In addition to this the extent to which the operational phase of a transaction and its commit phase overlap varies in practice. We provide a formal framework in which to describe these variations. We define serializability in this framework and show that it holds for many of the optimisations but not for others.

1 Introduction

Skip

2 The Model

We model a distributed transaction processing environment where transactions arrive at coordinator sites. In our simplified model a transaction consists of a set of actions that are supervised by a transaction coordinator (TC). The TC interacts with one or more resource managers (RMs) by means of message passing in order to perform these actions. Actions performed on resources may change the state of a resource.

We are interested in the behavior of the system when several RMs are involved in a transaction. Each process has an internal state and a set of rules which modify this state based on current state and the perceived state of remote processes. The rules and state variables for a particular type of process (in our system either RMs or TCs) are the same but a set of rules and a state exists for each process in the system. To express this we parameterise our rules. An instantiation of the parameterised rules gives rise to a set of rules for one process.

At various points during execution, the protocol being used might require that a process records its current state by writing it to non-volatile storage. After a crash, the process executes a recovery phase. The last recorded state is restored and the recovery begins.

2.1 Views and Message Passing

Let $\mathcal{T}$ be a set of transactions, and $\mathcal{V}$ be a set of resource states [1]. We use the notation $p.s$ to talk about the state variable s at process p. For example $t.final$ is the final state, either *commit* or *abort*, of the TC process t.

Processes in our distributed system have both their own local state and a *view* of the internal states of other processes. This view is constructed from information it receives from the other processes in the form of messages. We say,

$$q.s = \lhd x$$

holds at process p if the most up-to-date message p received from q informed p that q *was* in state x.

Not all local state is exchanged between processes. We distinguish between state variables that can be viewed at remote processes and call these variables *viewable*. Furthermore, in the centralised protocols we examine, local state is only exchanged between TCs and RMs.

We assume that processes in our system are connected by message passing channels. These channels are assumed to be reliable FIFO. Messages take some time to travel along these channels but messages are never lost and arrive in the order they were sent. Provided the sender and receiver of a message remain operational for long enough this level of reliability can be achieved in practice.

We also assume some mechanism to detect remote crashed processes in the system, sometimes called a *connection manager*, or *failure detector*. We

[1] this set contains the value $\perp$ which is the empty value.

say,

$$q.s = \lhd\, crashed$$

holds at p if p detects that q has crashed.

When a process q changes an externally–viewable local state variable, in our notation, $q.s := x$, a message is passed to a remote process p [2] enabling p to update its view of q. After the message is delivered at p, $q.s = \lhd\, x$ holds at p. We abstract all other communication issues (e.g. message routing, error detection, reliable delivery) in this way.

Sometimes the state stored at q will include more structured data than scalar values. In particular, in our model we wish to store sets of values. We say,

$$q.S \ni \lhd\, x$$

holds at p where S is a set in q's local state. This assertion will hold whenever the most up-to-date message p has received from q informs p that q's local state set S contained the element x. When q adds or deletes an element from its local set $q.S$ a message is propagated to other processes allowing them to update their views.

2.2 Rules, State and Stable Storage

Rules determine the behavior of processes by transforming the state of a process given that certain conditions are met. A rule is a pair which consists of a pre-condition and a postaction. The pre-condition is a logical statement which may contain assertions about views of remote process states. A postaction is a list of assignments to local state variables; assignments to viewable variables cause messages to be sent to remote processes.

Commit protocols are designed to ensure that after a process crash, a recovery process can continue with executing the protocol. Such recovery requires that at protocol-specific points in the execution, state is force written to stable storage. Force writing a record to stable storage is an expensive operation and, consequently, this is the focus for much of the research effort on protocol optimizations.

We add a new operator to our postactions which force writes *all* the process state to stable storage we call this operator *FORCE_WRITE*. After a crash the value of the process state will be restored to the value that was

[2] We don't specify to which processes messages are sent preferring to abstract this to those processes that are interested.

last force written. In practice not all state need be written but to simplify notation we will assume it is.

2.3 Rules for Centralised Two Phase Commit

Table 1 describes the state of a transaction coordinator t. A transaction coordinator carries out a set of actions from the set $\mathcal{A}$.

Associated with each TC is a set $Act_t = \{a_1, \ldots, a_k\}$ of actions which constitute the transaction t. These actions are assumed to have originated from some external application. $rm(a_i)$ is the resource manager that is to perform action a_i. The set of all RMs involved in carrying out these actions is denoted $RM(Act_t)$. For each transaction we assume each action is performed at a different resource manager. More formally, $rm(a_i) = rm(a_j) \Rightarrow i = j$.

In practice transactions may be interactive and have branching control structure. This means that actions are created as the transaction proceeds and several actions may be carried out at the same resource manager. We only model the case where all the actions are known at the start of the transaction, and each is carried out at a distinct resource manager. The rules we give do allow these actions to be executed in any concurrent manner.

To simplify the analysis, we assume that all actions yield results [3]. We use $ev(a)$ to denote the value that an action returns when executed at a RM. For example if the action was to read a data item at an RM the value would be the current value of the data item.

In the case that a TC reaches a final commit state $t.final = commit$ then $t.v(a)$ contains the result of each of the actions $a \in Act_t$, this can be thought of as the result of the distributed transaction.

For notational convenience we define the following sets for a TC action set Act_t.

- COMMIT $\stackrel{def}{=} \{commit_d \mid d \in RM(Act_t)\}$
- ABORT $\stackrel{def}{=} \{abort_d \mid d \in RM(Act_t)\}$
- PREPARE $\stackrel{def}{=} \{prepare_d \mid d \in RM(Act_t)\}$

[3] In reality some actions may just change a resources state without returning a value. In this case the result is the current value of the resource.

Name	Viewable	Description	Values
$t.s$	N	State	$\{\bot, prepare, commit, abort\}$
$t.A$	Y	Actions of the transaction	$\{a, done_a \mid a \in Act_t\}$
$t.Decn$	Y	Outcome of the transaction	ABORT $\cup$ COMMIT
$t.Prep$	Y	Prepare the transaction	PREPARE
$t.v$	N	Action return values	$Act_t \rightarrow \mathcal{V}$
$t.status$	N	Coordinator has crashed	$\{normal, crashed\}$
$t.final$	N	The final state	$\{commit, abort, zombie, \bot\}$

Table 1: The state of a transaction coordinator t. Initially $t.s = \bot$, $t.A = Act_t$, $t.Decn = t.Prep = \emptyset$, $\forall a \in Act_t$, $t.v(a) = \bot$, $t.status = normal$, $t.final = \bot$

We now give rules for TCs. These are divided into two sections. First we give rules for normal operation when $t.status = normal$, then we give the crash recovery rules. To ease notation, the clause $t.status = normal$ is omitted in the pre-conditions of the non recovery rules.

Before fully describing the rules and state for a RM we will introduce $d.R$ a viewable element of a resource manager's state. This variable is used to enable a RM to communicate with a TC. For example, in the next rule the clause $d.R \ni \lhd ack_t^{v'}$ becomes true when a TC receives a message from a RM acknowledging an action from its action set Act_t. In the rules we use the shorthand $t.A \cup := \{x\}$ for $t.A := t.A \cup \{x\}$.

CA Coordinator receives confirmation of an action
PRE: $a \in t.A \ \wedge \ rm(a).R \ni \lhd ack_t^{v'}$
POST: $t.A \cup := \{done_a\} - \{a\} \ \wedge \ t.v(a) = v'$

As soon as the TC gets an acknowledgement that a RM has performed an action it can start to prepare the transaction at that RM. There are two possible flavours of this rule. A TC might start to prepare a transaction at a RM as soon as the action at that RM has been acknowledged. We call this *overlapping prepare* where the operational and commit phases of the transaction overlap. Alternatively, the TC may require acknowledgements from all the RMs that the actions have taken place before starting to prepare a transaction at any RM. We call this *syncpoint prepare*. We will see that the choice of which rule to use may affect the serializability properties of protocols.

CP Overlapping Coordinator performs overlapping prepare
PRE: $done_a \in t.A$
POST: $t.Prep \cup := \{prepare_{rm(a)}\} \ \wedge \ t.s := prepare$

CP Syncpoint Coordinator performs syncpoint prepare
PRE: $t.A \cap Act_t = \emptyset$
POST: $t.Prep :=$ PREPARE $\wedge \ t.s := prepare$

Once the coordinator has started to prepare the transaction it starts to collect votes on the outcome of the transaction. We require two rules for this. The first collects a yes vote from a RM and the second collects a no vote or an outcome request.

CY Coordinator collects a yes vote
PRE: $d.R \ni \lhd \{yes_t\} \ \wedge \ t.s = prepare$
POST: $t.Prep := t.Prep - \{prepare_d\}$

CN Coordinator collects a no vote or an outcome request
PRE: $(d.R \ni \lhd no_t \ \vee \ (d.R \ni \lhd req_t \ \wedge \ t.s \neq commit))$
POST: $t.Decn :=$ ABORT $\wedge \ t.s = abort \ \wedge \ t.Prep := \emptyset \ \wedge \ FORCE\_WRITE$

If the coordinator receives yes votes from all the resource managers it can commit the transaction.

CC Coordinator commits the transaction
PRE: $t.Prep = \emptyset \ \wedge \ t.s = prepare \ \wedge \ t.A \cap Act_t = \emptyset$
POST: $t.Decn :=$ COMMIT $\wedge \ t.s := commit \ \wedge \ FORCE\_WRITE$

Finally, a RM may request the outcome of a transaction. We have seen in rule **CN** that this can cause a transaction to abort. In the case where the transaction has passed the point where it must commit, the request causes a commit outcome.

CCD Coordinator sends commit decision
PRE: $d.R \ni \lhd req_t \ \wedge \ t.s = commit$
POST: $t.Decn \cup := \{commit_d\}$ [4]

The coordinator now collects acks for the commit or abort decision made, and then finally commits or aborts the transaction. The abort rule **CFA** is not given but is similar.

[4] Although $commit_d \in t.Decn$, $commit_d$ is added to $t.Decn$ to cause a message to be resent to d to fulfill d's outcome request.

CCD Coordinator collects decision acks
PRE: $d.R \ni \lhd ackd_t$ [5]
POST: $t.Decn := t.Decn - \{commit_d,\ abort_d\}$

CFC Coordinator finally decides commit
PRE: $t.s = commit \ \wedge\ t.Decn = \emptyset$
POST: $t.final := commit$

We must now give rules to allow the coordinator to recover from a crash. There are two cases. If the coordinator has passed the point of committing the transaction it must continue to commit. This is why the commit record must be force written when the commit message is sent to the resource managers. The second case is when the transaction has not yet committed or if it has aborted, in this case the TC aborts the transaction.

CRA Coordinator recovers from crash and aborts
PRE: $t.s = abort \ \wedge\ t.status = crashed$
POST: $t.Decn := \mathrm{ABORT} \ \wedge\ t.status := normal$

CRC Coordinator recovers from crash and commits
PRE: $t.s = commit \ \wedge\ t.status = crashed$
POST: $t.Decn := \mathrm{COMMIT} \ \wedge\ t.status := normal$

It might be the case that a coordinator recovers from a crash and all record of the transaction it was coordinating has been lost. In fact the coordinator does not really exist as a separate entity. We model this situation by saying a coordinator enters a zombie state. It never recovers and all outcome requests result in abort replies.

CRZ Coordinator in a zombie state
PRE: $(t.s = \perp \ \vee\ t.final = zombie) \ \wedge\ d.R \ni \lhd req_t \ \wedge\ t.status = crashed$
POST: $t.Decn \cup := \{abort_d\} \ \wedge\ t.final := zombie$

2.4 The Resource Manager

We now turn our attention to the resource manager. Table 2 describes the internal state of a resource manager.

Resource managers carry out the actions of the Transaction Coordinators. Actions on these resources may conflict. Locking is used to detect

[5] In fact abort acknowledgements will not be received from RMs that voted “no”, however “no” votes can be collected instead.

these conflicts. Some locking information, $\mathcal{L}$, must be kept at the resource managers. For example, a data item resource might have shared locks and exclusive locks. Shared locks do not conflict with each other but they do with exclusive locks whereas exclusive locks always conflict. The following three operations are assumed to exist.

- $SET(d.L, a, t)$ updates the locking information at a resource manger d for transaction t, and action, a.
- $CLEAR(d.L, t)$ removes the locking information for transaction t.
- $TEST(d.L, a, t)$ is used to see if an action, a, would cause a conflict, by examining the types of locks held.

Name	Viewable	Description	Domain of Values
$d.Q$	N	Set of in doubt transactions	$2^{\mathcal{T}}$
$d.R$	Y	Responses	$2^{\{ack_t^v,\, yes_t, no_t, ackd_t, req_t, ro_t\}}$
$d.L$	N	Locking information	$\mathcal{L}$
$d.v$	N	Resource value	$\mathcal{V}$
$d.sv$	N	Saved value	$\mathcal{V}$
$d.status$	N	Recovering from crash	$\{normal, crashed\}$

Table 2: The state of a resource manager. Initially there is no locking information $d.R = d.Q = \emptyset$, $d.v = d.sv = \bot$ and $d.status = normal$.

We now give the rules for the RM. The first rule describes how a resource manager performs an action. Again we do not include the clause $d.status = normal$ in all the pre-conditions.

RA Resource manager performs an action
PRE: $t.A \ni \lhd a \,\wedge\, d = rm(a) \,\wedge\, TEST(d.L, a, t)$
POST: $d.R \cup := \{ack_t^{ev(a)}\} \,\wedge\, d.v := ev(a) \,\wedge\, UPDATE(d.L, a, t)$

We now give rules to allow the RM to vote. A RM may decide unilaterally to vote no and abort [6] or to vote yes and await the global outcome of the transaction. We also add a rule that allows a RM to vote no if it sees a

[6] A RM may vote no and abort for a number of reasons. One such reason might be to resolve a deadlock.

prepare message for a transaction that it has performed no action for. This is to ensure that if the RM crashes and loses all knowledge of the transactions and subsequently recovers and receives a prepare message, it can abort the transaction by voting no.

RUP Resource manager receives an unexpected prepare
PRE: $t.Prep \ni \lhd prepare_d \wedge \{ack^v_t | v \in \mathcal{V}\} \cap d.R = \emptyset$
POST: $d.R \cup := \{no_t\}$

RVY Resource manager votes yes
PRE: $t.Prep \ni \lhd prepare_d \wedge \{ack^v_t | v \in \mathcal{V}\} \cap d.R \neq \emptyset$
POST: $d.R \cup := \{yes_t\} \wedge d.Q \cup := \{t\} \wedge FORCE\_WRITE$

Whenever a RM votes "yes" to a transaction it adds the transaction to its local set $d.Q$ and awaits the outcome of that transaction.

RVN Resource manager votes no and unilaterally aborts
PRE: $\{ack^v_t | v \in \mathcal{V}\} \cap d.R \neq \emptyset \wedge t \notin d.Q$
POST: $d.R \cup := \{no_t\} \wedge d.v := d.sv \wedge CLEAR(d.L, t)$

If a RM votes has performed an action but has not yet voted "yes" to that transaction it is able to vote "no" and unilaterally abort the transaction.

RC Resource manager commits a transaction
PRE: $t.Decn \ni \lhd commit_d$
POST: $d.R \cup := \{ackd_t\} \wedge d.Q := d.Q - \{t\} \wedge d.sv := d.v \wedge CLEAR(d.L, t) \wedge FORCE\_WRITE$

RAB Resource manager receives abort decision
PRE: $t.Decn \ni \lhd abort_d \wedge \{no_t\} \notin d.R$
POST: $d.R \cup := \{ackd_t\} \wedge d.Q := d.Q - \{t\} \wedge d.v := d.sv \wedge CLEAR(d.L, t) \wedge FORCE\_WRITE$

RRQ Resource manager requests outcome
PRE: $t \in d.Q$
POST: $d.R \cup := \{req_t\}$

We give rules for crash recovery of a RM. Firstly, the RM resolves all transactions in the set $d.Q$ by requesting the outcome from the TCs using the rule **RRQ** above. When all the requests have been made the RM can continue processing.

RR Resource manager recovers
PRE: $d.status = crashed \wedge \forall t \in d.Q, req_t \in d.R$
POST: $d.v := d.sv \wedge d.status := normal$

3 Presumed Abort (PrA)

Presumed abort optimises the basic two phase commit by reducing the number of messages and forced writes required in the case that the transaction aborts. Because the coordinator always force writes a commit decision it need not force write an abort decision. The absence of a commit decision is enough to infer an abort decision. A RM need not now acknowledge abort decisions, because the final abort state is reached at the TC as soon as it decides abort. We can change our rules to reflect this optimisation as follows.

CN(PrA) Coordinator collects a no vote or an outcome request
PRE: $(d.R \ni \lhd no_t \vee (d.R \ni \lhd req_t \wedge t.s \neq commit))$
POST: $t.Decn :=$ ABORT $\wedge\ t.s = abort \wedge t.Prep := \emptyset \wedge t.final = abort$

RAB(PrA) Resource manager receives abort decision
PRE: $t.Decn \ni \lhd abort_d \wedge \{no_t\} \notin d.R$
POST: $d.Q := d.Q - \{t\} \wedge d.v := d.sv \wedge CLEAR(d.L, t)$

Once an abort decision is made at the coordinator the transaction can be forgotten. If a RM requests the outcome of a decision once the transaction is forgotten then the decision will be to abort. This is expressed in the rule **CN** by allowing an decision request in the pre-condition.

Typically, only a tiny proportion of transactions abort, so PrA would not, by itself, be a significant advantage. However, the big difference is that PrA can be combined with the read only optimisation described in the next section. When combined it can treat read only transactions as if they were aborted, providing a significant optimization.

4 Read Only Optimizations (RO)

Often an action only examines and does not change the state of a resource. If this is the case the resource manager does not need to know the overall decision of the transaction because it would take the same actions in the case of abort or commit. To optimise our protocols a resource manager can vote read only, (ro) to a prepare message and release any locks held for that transaction. We can change our rules to reflect this read only optimisation as follows.

CY(RO) Coordinator collects a yes vote (or a read only vote)
PRE: $(d.R \ni \lhd yes_t \vee d.R \ni \lhd ro_t) \wedge t.s = prepare$

POST: $t.Prep := t.Prep - \{prepare_d\}$

RVR Resource manager votes read only
PRE: $t.Prep \ni \lhd prepare_d$
POST: $d.R \cup := \{ro_t\} \wedge CLEAR(d.L, t)$

RC Resource manager commits a transaction
PRE: $t.Decn \ni \lhd commit_d \wedge \{ro_t\} \notin d.R$
POST: $d.R \cup := \{ackd_t\} \wedge d.Q := d.Q - \{t\} \wedge d.sv := d.v \wedge CLEAR(d.L, t) \wedge FORCE\_WRITE$

RAB Resource manager receives abort decision
PRE: $t.Decn \ni \lhd abort_d \wedge \{no_t, ro_t\} \notin d.R$
POST: $d.R \cup := \{ackd_t\} \wedge d.Q := d.Q - \{t\} \wedge d.v := d.sv \wedge CLEAR(d.L, t) \wedge FORCE\_WRITE$

The rule **CC** changes slightly. If a resource manager responds ro_t to a prepare message then it need not participate in the second phase of the transaction. In fact if all resource managers respond ro_t (i.e.. when the whole transaction is read only which is often the case in practice) and the PrA optimisation is being used in conjunction with the RO optimisation, then no commit decision need be made at all. The transaction can be handled in the same way as if it had aborted and the coordinator can move directly to a final *commit* state, forget the transaction and need not force write any records. We do not explicitly model the accounting information which needs to be maintained at each TC but it would be straight forward to do this.

5 Grouping rules

In order to define serializability, we will first group our rules into categories. These categories can be further divided by the transaction or resource with which the rule interacts. For example, a RM d may end a transaction i by receiving an abort decision from a TC we write this rule as RAB_i^d, and we say it is a member of the end rules ϵ_i^d. The following table classifies the rules.

Start rules σ	Middle rules μ	End rules ϵ	Post-end rules ϕ
RA	CA, CRA, CP, CRC, CY, CRZ, RRQ,RUP, CN, CC, CCD	RVN RC RAB $Crash_d$, $t \notin d.Q$ RVR	CCA CFC

All start rules, σ, are of the form RA where a RM d performs an action for a transaction i which we sometimes write as RA_i^d. This rule updates locking information and performs the action as long as there is no conflict with other outstanding actions. Middle rules, μ, define the interaction between a TC and its RMs. These include collecting action acknowledgement, performing the steps of two phase commit, and any required crash and recovery steps. None of the rules in this category change the locking information held at a RM. End rules, ϵ, end the interaction of a RM with a TC. We can think of the case that a RM crashes before a transaction is recorded as being in doubt (i.e. in the set $d.Q$), as one of these rules. We write, ϵ_i^d to represent a rule which ends the involvement of RM d with transaction i. After an end rule a TC may collect acknowledgments of a decision we call these rules ϕ rules.

6 Executions

Let $\tau_1, \ldots, \tau_n$ be the local states of the TCs and $\delta_1, \ldots, \delta_m$, be the local states of the RMs in our system, at a point in time.

The global system state is simply the $m+n$ tuple

$$S = \langle \tau_1, \ldots, \tau_n, \delta_1, \ldots, \delta_m \rangle.$$

The system state evolves by taking steps. These steps are defined by the rules. A rule may happen at a process when the precondition to that rule becomes true. Exactly one step is taken at a time. A step is assumed to be an atomic instantaneous event. In some of our rules an atomic event may comprise one or more local state changes. However, an atomic event only ever affects one process and all changes refer to a particular transaction.

Suppose rule R fires for t_i, when it is in state τ_i. We write this as $\tau_i \xrightarrow{R} \tau_i'$. This causes the global system state to take a step

$$\langle \tau_1, \ldots, \tau_i, \ldots, \tau_n, \delta_1, \ldots, \delta_m \rangle \xrightarrow{R} \langle \tau_1, \ldots, \tau_i', \ldots, \tau_n, \delta_1, \ldots, \delta_m \rangle$$

Similarly, a resource manager d_i may take a step, $\delta_i \xrightarrow{R} \delta_i'$ causing the global state to change in a similar way.

Definition 1 An *execution*, ρ, of our system, from an initial global state S_1, is a sequence of rules $R_1, R_2, \ldots, R_r, \ldots$ such that $S_1 \xrightarrow{R_1} S_2 \xrightarrow{R_2} \ldots \xrightarrow{R_{r-1}} S_r \xrightarrow{R_r} \ldots$.

Definition 2 A transaction is *before* another with respect to a RM d, $t_i \overset{d}{\prec} t_j$, in an execution iff the rule [7] ϵ_i^d is before the rule σ_j^d in that execution. If neither $t_i \overset{d}{\prec} t_j$ nor $t_j \overset{d}{\prec} t_i$ we say there exists an *interleaving* of t_i and t_j on d.

Definition 3 A transaction is *before* another in an execution, $t_i \prec t_j$ iff $\forall d \in RM(Act_{t_i}) \cap RM(Act_{t_j})$, $t_i \overset{d}{\prec} t_j$. If neither $t_i \prec t_j$, nor $t_j \prec t_i$ in an execution we say t_i and t_j are *interleaved* in that execution.

Definition 4 A *serial execution* is one in which no two transactions are interleaved. More formally a total order π exists in which $t_{\pi_1} \prec t_{\pi_2} \prec \ldots \prec t_{\pi_n}$.

Perhaps we should mention Hebrand Semantics as an alternative classification of the semantics and equivalence of schedules

Definition 5 Let τ', τ'' be two states of a TC t, after it has taken two distinct sequences of steps from a common initial state τ. We say τ' and τ'' are equivalent states, $\tau' \equiv \tau''$, iff $(\tau'.final = \tau''.final) \wedge (\tau'.final = commit \Rightarrow \forall a \in Act_t, \tau'.v(a) = \tau''.v(a))$ Similarly, we say δ', δ'' are equivalent states of a RM, $\delta' \equiv \delta''$, iff $\delta.sv = \delta'.sv$.

Definition 6 A TC has *terminated* when it has reached a decision, or it has become a zombie, $t.final \neq \bot$. We say a RM is *idle* when there are no in doubt transaction at this RM, $d.Q = \emptyset$. We say an execution, ρ, has *terminated* when all TCs have terminated and all RMs are idle in ρ.

[7] We note that each for each transaction at most one start and one end rule per resource manager is allowed by the rules.

Definition 7 Let $S' = \langle \tau'_1, \ldots, \tau'_n, \delta'_1, \ldots, \delta'_m \rangle$, $S'' = \langle \tau''_1, \ldots, \tau''_n, \delta''_1, \ldots, \delta''_m \rangle$ be the global states after two distinct terminated executions ρ', and ρ'' starting from an initial state global state S. We say these terminated executions are *equivalent*, $\rho' \equiv \rho''$ iff $\forall i \in \{1, \ldots, n\}$, $\tau'_i \equiv \tau''_i$ and $\forall j \in \{1, \ldots, m\}$, $\delta'_j \equiv \delta''_j$.

Definition 8 A terminated execution is *serializable* iff there exists an equivalent terminated serial execution.

Definition 9 A protocol is *serializable* iff given an initial set of TCs and RMs every possible terminated execution generated by the rules of the protocol is serializable.

Lemma 1 *Let t_i be a committing transaction in an execution, ρ then the subsequence of start and end rules pertaining to t_i in ρ will have the form.*

$$\ldots \sigma_i^{d_1} \ldots \sigma_i^{d_n} \ldots \epsilon_i^{d_{\pi 1}} \ldots \epsilon_i^{d_{\pi n}} \ldots$$

where $\{d_1, \ldots, d_n\} = RM(Act_{t_i})$ and π is a permutation of $1 \ldots n$.

Proof To establish the lemma we show that no end rule for transaction t_i may happen until all start rules have happened. The rule CP(Syncpoint) requires all start actions to have happened before the TC t_i starts to prepare the transaction. No end rules may happen before CP(Syncpoint) thus no end rules happen until all start rules have happened.

There is one exception to this. A RM d, could crash and release locks after performing an action but before that transaction becomes in doubt at d. After this crash a different RM, d', may perform an action. This would cause ϵ_i^d to be before $\sigma_i^{d'}$ in the execution. If this happens we can see that on recovery d will receive and unexpected prepare from TC, and rule RUP will cause d to vote no causing t_i to abort.

If t_i commits then then each RM in $RM(Act_{t_i})$ must either vote read only with rule RVR or acknowledge the commit decision with rule RC. Furthermore all RMs must perform an action with rules RA. We conclude that each RM in $RM(Act_{t_i})$ perform a start rule and an end rule and all end rules happen after all the start rules.

□

Lemma 2 *Let t_i be an aborting transaction in an execution, ρ then t_i makes no persistent changes at any $d \in RM(Act_{t_i})$ and furthermore no other transaction t_j can see temporary changes made by t_i.*

Proof

If an action executed on behalf of t_i for a resource manager d, changes the value $d.v$ of d, rule RA_i^d must acquire an exclusive lock. This means rule RA_k^d may not happen for any other transaction t_k until d performs an end rule, ϵ_i^d. There are four possible end rules:-

- The RM d commits RC. This is not possible since t_i aborts.
- The RM d votes read-only, RVR. This is not possible since t_i changes the value $d.v$ at d.
- The RM d aborts RAB. The postaction of this rule releases locks but also restores the value of $d.v$ before it was changed by RA.
- The RM crahes releasing locks before the transaction was in doubt. In this case the locks are lost but on recovery the d restores the previous value $d.sv$ into $d.v$ before any other RA rules may happen. The case when the RM crashes when t_i is in doubt is not considered as locks are not lost since they were written to stable storage.

In each case if locks are released we see that the value before the action, $d.sv$, is restored to $d.v$. We can conclude that no persistent changes are made to $d \in RM(Act_{t_i})$ and furthermore no other transaction can see temporary changes.

□

Theorem 1 *The presumed abort protocol with read only optimisation and syncpoint prepare is serializable.*

Proof

Consider two transactions, t_i and t_j, in an execution, ρ suppose neither $\tau_i \prec t_j$ nor $t_j \prec t_i$ and also without loss of generality t_i aborts in ρ. We can construct an execution ρ' from ρ by moving all rules pertaining to t_i to the very start of the execution maintaining their original order in ρ. We now claim $\rho \equiv \rho'$. By lemma 2 the values of $d.v$ and $d.sv$ for all $d \in RM(Act_{t_i})$

is unchanged by t_i so for all RMs rho' will be equivalent to those in ρ. Futhermore, by lemma 2 no other transaction t_j may see any temporary changes made by t_i so the values of $t_j.v$ remains the same in ρ'. Since t_i aborts in ρ and ρ' we can conclude $\rho \equiv \rho'$.

Now consider two commiting transactions t_i, t_j and suppose neither $\tau_i \prec t_j$ nor $t_j \prec t_i$ in ρ. By lemma 1 each transaction's start rules are before any of their end rules therefore there exists a common resource manager that they are interleaved on; more formally $\exists d \in RM(Act_{t_i}) \cap RM(Act_{t_j})$ such that

$$\ldots \sigma_i^d \alpha \sigma_j^d \beta \epsilon_i^d \gamma \epsilon_j^d \ldots$$

is part of the execution ρ where α, β, γ represent arbitrary, possibly empty, sequences of rules.

To show any execution is serializable we will use double induction. The outer induction will be on the number of *interleavings* of the form above. The inductive step of this outer induction will require a further sub-induction. This sub-induction will show that if two transactions t_i, t_j, are interleaved on a RM $d \in RM(Act_{t_i}) \cap RM(Act_{t_j})$ in an execution ρ then an equivalent execution ρ' can be constructed in which these transactions are not interleaved on d.

The sub-induction will be on the distance of the end rule of t_i at d, denoted ϵ_i^d, from the start rule of t_j at d, denoted σ_j^d, assuming without loss of generality t_i starts at d before t_j starts at d in the execution.

We will start with the base case of the sub-induction where the distance rule ϵ_i^d from σ_j^d in ρ is zero (i.e. β is the empty string). We can write this as

$$\ldots \sigma_i^d \alpha \sigma_j^d \epsilon_i^d \gamma \epsilon_j^d \ldots$$

We can commute σ_j^d and ϵ_i^d because t_i and t_j don't conflict (otherwise the original execution would not be possible).

To prove the inductive step of the sub-induction we are required to reduce the length of β by one. Let us assume we have chosen t_i, t_j and d in such a way that there are no other t_i, t_j or d we could have chosen which give a smaller β.

Before providing transformations to reduce the size of β in our sub-induction we prove the following fact and use it to transform β to a form which contains no rules of the form σ_k^d or $epsilon_k^d$, for $k \notin \{i, j\}$.

Whenever another transaction t_k starts (ends) at d in β then it will also end (start) in β. More formally, $\forall k \notin \{i, j\}$ $\sigma_k^d \in \beta$ *iff* $\epsilon_k^d \in \beta$. Suppose not then,

$$\ldots\sigma_i^d\alpha\sigma_j^d\beta'\sigma_k^d\beta''\epsilon_i^d\gamma'\epsilon_k^d\ldots \; \mathit{or} \; \ldots\sigma_k^d\alpha'\sigma_j^d\beta'\epsilon_k^d\gamma'\epsilon_j^d\ldots$$

provide smaller candidates for β, which contradicts our assumption that β is the smallest. So we can write β as

$$\ldots\sigma_{k_1}^d,\ldots,\sigma_{k_2}^d\ldots,\sigma_{k_n}^d\ldots\epsilon_{k_{\pi_1}}^d\ldots\epsilon_{k_{\pi_n}}^d\ldots$$

where π is a permutation of $1,\ldots,n$.

We now make the following observation. A start rule σ^d, can be moved *before* any middle rule μ and also before any start rules of the form $\sigma^{d'}$, $d' \neq d$ in an execution to provide an equivalent execution. Using this observation and the fact that σ_j^d does not conflict with any $\sigma_{k_l}^d$, $l = 1\ldots n$ (as otherwise the original execution ρ would not be possible) we can move all start rules of the form σ_k^d, $k \notin \{i,j\}$ before σ_j^d. Similarly we can move all end rules of the form ϵ_k^d, $k \notin \{i,j\}$ that are in β after ϵ_i^d to give the equivalent execution below.

$$\ldots\sigma_i^d\alpha\sigma_{k_1}^d\ldots\sigma_{k_n}^d\sigma_j^d\beta'\epsilon_i^d\epsilon_{k_{\pi_1}}^d\ldots\epsilon_{k_{\pi_n}}^d\gamma\epsilon_j^d$$

Using this initial transformation we can assume β does not contain any rules of the form σ_k^d or ϵ_k^d for $k \notin \{i,j\}$.

The following transformations reduce the size of β and form the inductive step of our sub-induction. Transformation T1 covers the cases where the first and last rules of β are start and end rules respectively, on a RM $d' \neq d$. In the sequal, $\sigma_{\{i,j\}}^d$ is short for $\sigma_i^d \; \mathit{or} \; \sigma_j^d$.

$\underline{T1}$

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\alpha & {} & \sigma_j^d & \sigma^{d'}_{\{i,j,k\}} \arrow[ll, bend left] & \mid & \beta & \epsilon^{d'}_{\{i,j,k\}} \arrow[rr, bend right] & \epsilon_i^d & {} & \gamma\epsilon_j^d\ldots \end{tikzcd}$$

In T1 we move a start (end) rule for any transaction on a RM $d' \neq d$ before (after) a start (end) rule on a RM d. The rules commute because they don't conflict since $d \neq d'$. Furthermore, start (end) rules can always be moved earlier (later) in an execution. Clearly T1 results in an equivalent execution, reducing the size of β.

$\underline{T2}$

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\alpha & \sigma_j^d & \epsilon^{d'}_{\{i,k\}} \arrow[l, bend left] & \Big| & \beta & \sigma^{d'}_{\{j,k\}} \arrow[rr, bend right] & \epsilon_i^d & \gamma\epsilon_j^d\ldots \end{tikzcd}$$

In T2 we move an end (start) rule on a RM $d' \neq d$ before (after) a start (end) rule on a RM d. In general an end (start) rule cannot be moved before (after) a start (end) rule to yield an equivalent execution. However, in the case that rules are for different transactions as well as on different RMs we can see the resulting execution will indeed be equivalent. Again the size of β is reduced.

$\underline{T3}$

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\alpha\sigma_j^d\beta & r & \sigma_i^{d'} \arrow[l, bend left] & \epsilon_i^d\gamma\epsilon_j^d\ldots \qquad r \neq \epsilon^{d'}_{\{i,j,k\}} \end{tikzcd}$$

T3 does not reduce the size of β but it may need to be performed before one of the β reducing transformations.

If $\sigma_i^{d'}$ is immediately to the before of ϵ_i^d in ρ it may be move earlier in the execution. Unfortunately, it cannot be moved before any end rules involving d' as this may not result in an equivalent execution. The transformation therefore moves $\sigma_i^{d'}$ earlier as long as it is not moved before an end rule on d'. This results in an equivalent execution. Although this does not reduce the size of β it may allow other transformations to do so.

$\underline{T4}$

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\alpha\sigma_j^d & \epsilon_j^{d'} \arrow[r, bend right] & r & \beta\epsilon_i^d\gamma\epsilon_j^d\ldots \qquad r \neq \sigma^{d'}_{\{i,j,k\}} \end{tikzcd}$$

T4 is very similar to the T3. In the case that an end rule $\epsilon_j^{d'}$ is immediately after of $\sigma_j d$ we may move $\epsilon_j^{d'}$ after any rule that is not a start rule on d'. This will result in an equivalent execution. Again it does not reduce the size of β but it may allow other transformations to do so. We now consider the cases when the first or last rules of β are μ rules.

$\underline{T5}$

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\alpha & \sigma_j^d & \mu_{i,k}^{d,d'} \arrow[l, bend left] & \Big| & \beta' & \mu_j^{d,d'} \arrow[rr, bend right] & \epsilon_i^d & \gamma\epsilon_j^d\ldots \end{tikzcd}$$

The previous transformation allows us to remove μ rules from β. Clearly, μ rules commute with start rules, provided they are on different transactions, to give an equivalent execution. We use this fact to move $\mu_{i,k}$ rules earlier in the execution before σ_j^d. Also a μ rule can be moved after a neighbouring end rule, again provided that it is on a different transaction. This allows us to move μ_j after ϵ_i^d to giving an equivalent execution.

<u>T6</u>

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\alpha\;\sigma_j^d & \mu_j^{d,d'} \arrow[r, bend right] & r & \beta'\epsilon_i^d\gamma\epsilon_j^d\ldots \qquad r\neq\mu_j,\epsilon_j \end{tikzcd}$$

Transformation T6 does not reduce the size of β but it may need to be performed before one of the β reducing transformations. It can only be applied when $r \neq \mu_j, \epsilon_j$. The transformation yeilds an equivalent execution.

<u>T7</u>

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\alpha\sigma_j^d\beta' & \mu_j^{d,d'} \arrow[r, bend right] & \epsilon_i^d & \gamma\epsilon_j^d\ldots \end{tikzcd}$$

T7 allows μ_j rules to be moved after ϵ_j^d. This last final transformation allows us to remove all the μ from β.

Applying transformations T1, T2, T5 or T7 (possibly prefixed with transformations T6, T4 or T3) reduce the size of β by one, however we need to be sure that one transformation is always applicable. There is one case which will result in a situation where no transformation can be applied. That case is when $\beta = \epsilon_j^{d'}\sigma_i^{d'}$. We can write such an execution

$$\begin{tikzcd}[column sep=tiny] \ldots\sigma_i^d\ldots\sigma_j^d & \epsilon_j^{d'} \arrow[r, bend right, leftrightarrow] & \sigma_i^{d'} & \epsilon_i^d\ldots\epsilon_j^d\ldots \end{tikzcd}$$

In general, commuting $\epsilon_j^{d'}$ with $\sigma_i^{d'}$ does not give an equivalent execution. However, we will argue that if the original execution ρ was possible then these rules do commute to give a possible equivalent execution. Notice t_i, t_j at d do not conflict (i.e. they hold a read-only lock on d) because σ_i^d is followed by σ_j^d before ϵ_i^d. By examining the rules RC and RVR we see that exclusive locks are never released before shared locks at the RMs involved in a committing transaction. This means $\epsilon_j^{d'} = RVR_j^{d'}$, that is RM d' votes read-only. Thus the state of RM d' is not changed by t_j. So commuting $\epsilon_j^{d'}$ and $\sigma_i^{d'}$, yields an equivalent execution.

Our sub-induction reduces the number of interleavings by one, furthermore it does not create any other interleavings in the process. This then forms the inductive step of the outer induction. The base case of the outer induction is when the number of interleavings is 0, in which case the execution is trivially serialized. We conclude any execution can be serialized.

□

Theorem 2 *The presumed abort protocol with read only optimisation and overlapped prepare is not serializable.*

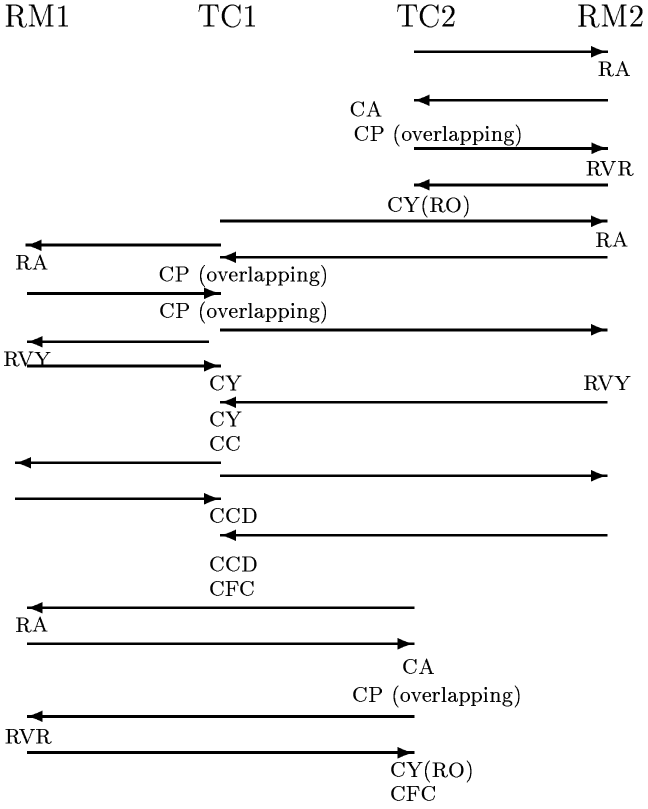

Figure 1: Strong Isolation is not maintained

Proof The counter example in figure 1 shows an execution that is not serializable. TC1 and TC2 both access RM1 and RM2. TC1 performs a write action at both these RMs between the two read actions of TC2 at RM1 and RM2. Neither of the possible serial executions of these two transactions will result in the same values being read at TC2 as the interleaved execution presented in figure 1. Interestingly, in the previous proof we made use of the fact that no end rule, for a committing transaction, proceeds a start rule for that transaction in an execution. We can no longer make this assumption.

7 Conclusions

skip

References

[1] S. Levitan Y.J. Al-Houmaily, P.K. Chrysanthis. An argument in favour of the presumed commit protocol. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, pages 255–265. IEEE, April 1997.