Contents

1 RPC semantics

Message	Type	Direction	Remarks
REQ	Request	C $\rightarrow$S	Client wants service from server
REP	Reply	S $\rightarrow$C	Server replies to client request
ACK	Acknowledgement	(both)	Previous message arrived successfully
AYA	Are You Alive?	C $\rightarrow$S	Check server is functioning
IAA	I Am Alive	S $\rightarrow$C	Server confirms it is functioning
TA	Try Again	S $\rightarrow$C	Server is busy
AU	Address Unknown	S $\rightarrow$C	No server at this address

Table 1: Protocol Message Types

- REQ and REP are essential
- ACK helps ensure reliability by handling message loss
- AYA and IAA allow the client to determine if the server is available if it receives a timeout following a REQ or ACK

Remote Procedure Calls (RPC) encapsulates the message sending and receiving between the client and server such that the client code does not have to deal with it.

Assuming there is some client code C compiled to C_{obj} and server code S compiled to S_{obj}, in order for the code to be executed remotely both are linked to a language specific stub which contains the logic for the remote communication between client and server.

The stubs for the client and server are produced using an interface definition S_{if} which describes the interface to the server code S.

Basic call procedure:

1 Client calls client stub as it would the server code if it was compiled locally

2 Client stub builds the message (including parameter packing) to be sent to the server

3 Message is sent to the server

4 Server receives the message and passes it to the server stub

5 Server stub parses the message, unpacks the parameters and calls the actual routine

6 The code executes and the results are packed into a reply message

7 The reply message is sent to the client

8 The client stub unpacks the results and returns them to the client routine

1.1 Parameter Passing

1.1.1 Methods

Methods of parameter passing:

Pass by value

The parameter value is copied to the stack of the executing process.

Changes to the value of the variable in the called routine do not affect the original copy.

Pass by refernce
: A reference to the original variable is copied to the stack of the executing process.

  When the called routine changes the value of the parameter the original copy is also changed.

Pass by copy-restore (aka value-result)
: The parameter value is copied to the stack of the executing process, then copied back when the routine terminates.

  When the called routine changes the value of the parameter the original copy is also changed.

In most situations pass by reference and pass by copy-restore yield identical results.

In RPC pass by copy-restore is used in place of pass by reference.

1.1.2 Packing

Parameter marshalling and un-marshalling is converting a set of parameters to and from a message to b sent between a client and server.

Must have a standard representation for data types, e.g. integer types, due to different physical reorientations of different architectures, e.g. endianness.

1.2 Binding

Static Binding
: Address of server is hard coded in the client stub.

Dynamic Binding
: Address of server is located at run time.

1.2.1 Dynamic Binding

- The service specification is stored (registered) on a statically addressed binder (a.k.a. name server)
- The server uses the service specification to generate the server stub
- The developer of the client selects a suitable service specification which is used to compile the client stub
- The client stub contacts the binder at run time to obtain the address of the server

1.2.2 Service Specification

Stores following information on binder/name server:

- Name of service
- Service routine specification
- Address of server running the service

Require language independent way to define service routine specification (a.k.a function signature): Interface Definition Language (IDL).

e.g. `read(in char name[n], out char buf[n], in int length, in int pos)`

`in`, `out`, `inout` specify the direction of the variables.

`in` and `inout` parameters are sent from the client to the server.

`out` and `inout` parameters are sent from the server to the client.

An `inout` parameter is the equivalent of pass by reference.

1.3 Failure Handling

Two possible outcomes of an RPC call:

Normal Termination
Client received the reply from the server as expected.

Abnormal (Exceptional) Termination
Client does not receive a reply from the server as expected.

This can be for a number of reasons: communication issues, node crashes, etc.

Causes of failures:

1 Client cannot contact the server

2 Client's request message does not reach the server

3 Server's reply message does not reach the client

4 Server crashes during the call

5 Client crashes during the call

Actions in event of failures:

Failure 1
Client received address unknown message from the node it tried to contact.

Failures 2, 3 & 4
Client times out waiting for a reply from the server.

May makes several further attempts to contact the server before an abnormal termination or terminate right away.

Failure 5
Orphan process is created on server.

See section 1.3.2.

Figure 1 shows which RPC semantic is used in each failure case.

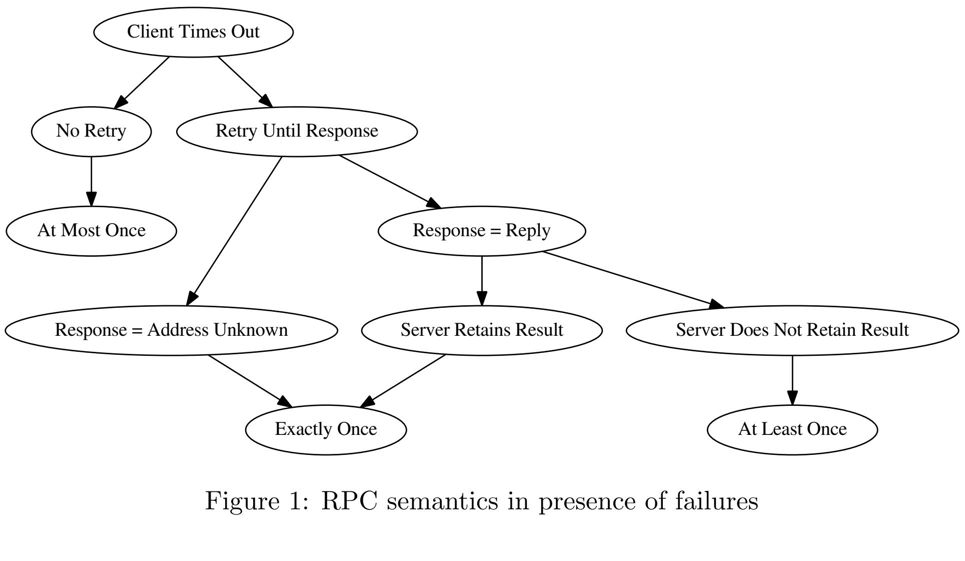


Figure 1: RPC semantics in presence of failures

1.3.1 Semantics

At Least Once

If client makes a call and it times out then it retries a finite number of times, if it continues the fail then the client terminates abnormally.

Server always executes the request from the client and sends the reply regardless of previous client requests (stateless server).

At Most Once

If a client request times out then retries are made as per At Least Once semantics, however each request also contains a sequence number such that all retries that are made have the same sequence number as the original request.

For each client, the server stores the results of the last executed call and the sequence number it corresponds to (stateful server).

If a request from a client is received and its sequence number already exists in the stored results then the stored results are returned to the client.

Need to ensure that the storage of previous results is persistent across server reboots and crashes.

Exactly Once

"All or nothing" behaviour requires that:

- Normal termination gives exactly one execution
- Abnormal termination gives exactly no executions

Difficult (sometimes impossible) to implement in the presence of server crashes, for instance if the server is in the middle of an unrecoverable operation.

Required client and server use a co-ordinated recovery strategy for handling abnormal termination.

See atomic transactions in section 3.

1.3.2 Orphans

Execution started on a server when a client request is received and the client crashes before it can receive the reply.

Orphans can consume resources and create consistency issues between orphans and normal executions (in he case where the client retries the request).

For instance, if an RPC locks a file to be written to and the client does not get a reply from the server they may retry by sending another RPC, which will be waiting for the orphaned RPC. In this case neither RPC will terminate.

Treatment of orphaned execution:

- When a server receives a request it could check for an identical request from the same client that is already being executed, if any are found they should be terminated before executing the current request.
- A server could periodically check that clients with requests executing on the server are alive, if the failure of a client is suspected then the server terminates any associated executions.

2 Clocks and Order

2.1 Happened Before Relation

The happens before relation ($A \rightarrow B$) defines the order of two events A and B such that event A happens logically before event B.

Assume that:

- Processes communicate only via messages
- All events on a single process form a chain of happened before relations
- Sending or receiving a message is an event
- $A \not\rightarrow A$
- Gives partial order to events in a system

Defined by:

1 The relation $\rightarrow$ satisfies the following conditions:

   1 If A and B are events in the same process and A executes before B then $A \rightarrow B$

   2 If A is the sending of a message m to another process and B is the receiving of message m, then $A \rightarrow B$

   3 If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

2 Two distinct events A and B are concurrent ($A||B$) if $A \not\rightarrow B$ and $B \not\rightarrow A$

2.2 Logical Clocks

- Used to implement partial order in a system.
- Each event e in a process i has a timestamp $C_i(e)$

Conditions:

1 If A and B are events in process i and $C_i(A) < C_i(B)$ then $A \rightarrow B$

2 If A is the sending of message m from process i and B is the receiving of message m by process j then $C_i(A) < C_j(B)$

Note that the inverse of condition 1 cannot hold.

Implementation rules:

1 Each process i increments its logical clock C_i immediately after an event (meets condition 1)

2
   1 Each message m sent from a process i in event A has a timestamp $T_m = C_i(A)$

   2 When receiving message m at process j in event B, read T_m and increment $C_j(B)$ such that $C_j(B) > T_m$

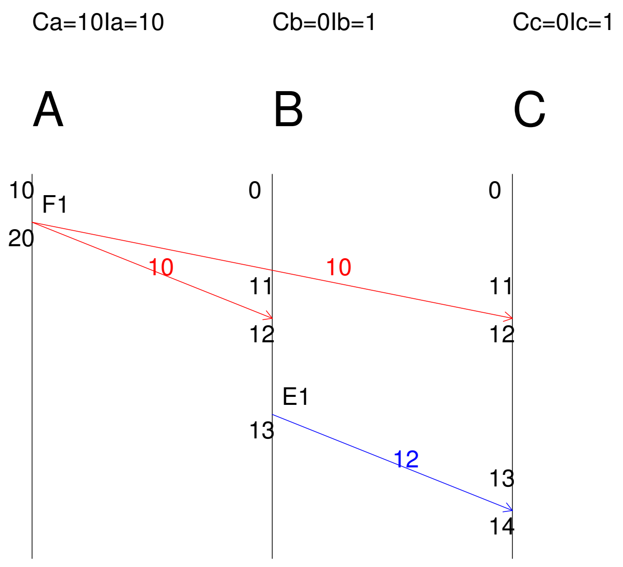


Figure 2: Logical Clock example

2.3 Total Order

Useful to put a total order on a set of events. Required a fixed tie braking rule for concurrent events, a common rule is using process numbers.

Define relation $\Rightarrow$ as $A \Rightarrow B$ if and only if:

- $C_i(A) < C_j(B)$
- or, $C_i(A) = C_j(B)$ and $P_i < P_j$

Note that:

- If $A \rightarrow B$ then $A \Rightarrow B$
- For a set of events there could be several valid $\Rightarrow$ relations for a single $\rightarrow$ relation, depending on the rule used to break ties.
- Total order based on logical clocks cannot be guaranteed to respect temporal ordering of events external to the system, this can lead to anomalous behaviour.

Figure 3 demonstrates a situation where anomalous behaviour can occur, in this case there is no way to guarantee that the logical clock timestamps of the computer messages will respect the actual order of events.

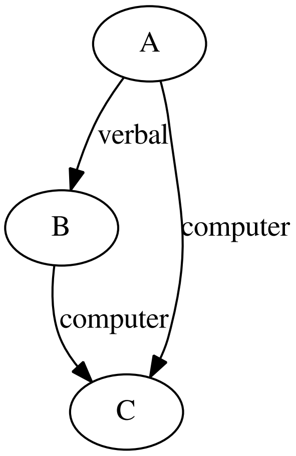


Figure 3: Anomalous Behaviour example

To avoid anomalous behaviour the strong clock condition must hold:

$$\forall\, a, b \in S : if\, a \rightsquigarrow b\, then\, C(a) < C(b)$$

This condition cannot be met using logical clocks as they only tick for events inside the system S.

2.3.1 Resource Management Example

Example usage: managing exclusive access to a single shared resource.

Conditions

1 A process that has been granted a resource must release it before it can be granted to another process

2 Separate requests for the same resource must be granted in the order they were made

3 If every process is granted a resource eventually releases it, then every request is eventually granted

Assumptions

- Events can be ordered by the total order relation $\Rightarrow$
- For any processes P_i and P_j, messages that are sent between them are received in the sent order
- A process can send a message to every other process
- Each process maintains a local message queue
- Initial state:
  - P_0 has the resource
  - All message queues contain the message: $T_0 : P_0$ `requests resource`
  - Where T_0 is less than the clock value of any process

Rules

1 To request a resource P_i sends the message $T_m : P_i$ `requests resource` to every other process and keeps a copy in its local message queue

2 When the request message is received by a process, assuming the process has not already replied to another request a timestamped acknowledgement is sent to P_i

3 To release a resource the process P_i sends message $T_m : P_i$ `releases resource` to all processes and removes the request message from its local message queue

4 When a process receives a release message it removes any associated request messages from its local queue

5 A process P_i is granted access to the resource when the following conditions are met:

   1 There is a request message $T_m : P_i$ `requests resource` in its local queue that is ordered before every other request message in the queue

   2 P_i has received an acknowledgement message with a timestamp greater than T_m from every other process in the system

2.4 Physical Clocks

The correctness of a physical clock is given by the error coefficient k (where $k \to 0$) and the synchronisation error ϵ.

Conditions:

1 For every clock i and every time t: $1 - k \leq \frac{dC_i(t)}{dt} \leq 1 + k$

2 For every clock i and j and very time t: $|C_i(t) - C_j(t)| \leq \epsilon$

μ is the minimum message transmission time in system S.

Maximum value of ϵ must be such that the synchronisation error is lower than the minimum message transmission time after taking into account the clock error coefficient:

$$\epsilon < (1 - k)\mu$$

Clock adjustments are performed periodically (usually in software) to keep physical clocks in synch.

Large "jumps" or rollbacks in time of a physical clock should be avoided.

Once example is the "Central Spray" approach, in which a central "master" clock transmits its time to all other systems which update their clocks to the received time.

The time transmitted to each node is corrected for the estimated transmission time between the master and the node such that the transmitted time will be correct by the time the message is received by the node.

3 Transactions

For controlling operations on persistent storage resources.

"ACID" properties

Atomicity
: "All or nothing".
In the event of failure the state must be restored to its condition prior to execution.

Consistency
: Ensures executions are scheduled to run such that one will not interfere with another.

Independence
: The effects of an execution (A) must not become visible to another execution (B) until after A has successfully completed.

Durability
: Results produced by a successful execution are not damaged by a subsequent failed execution.

Transaction primitives

Begin
: Command indicating start of a transaction

End
: Command indicating end of a transaction in which changes should be preserved

  Normal termination
  : Normal commit happens
  All changes are made to non-volatile storage

  Aborted termination
  : Transaction terminates without producing any results.

Abort
: Explicitly abort the transaction and discard changes

Transaction operations are often built into stubs to hide implementation details from the calling code.

3.1 Approaches used to support transactions

1 Concurrency control

  a Operations (such as read and write locking) are used to access shared objects *(consistency)*

  b While a transaction is running it holds locks it acquires to prevent other transactions accessing or modifying the objects *(independence)*

2 End of transaction (commit)

  a Locks are released during commit protocol, therefore changes only become visible after the transaction has completed *(independence)*

  b All updates are forced to disk so that node crashes do not destroy the results of a transaction *(durability)*

3 A client or server crash during a transaction leads to an abort *(atomicity)*

3.2 Serializability

- Concurrent modification of objects can cause unexpected behaviour
- Transaction executions are required to be a serializable schedule

A schedule S consisting of a set of transactions $\{T_1, T_2, \ldots, T_n\}$ is serializable if there is no situation where a transaction needs to obtain a lock on an object that is already held by another transaction that has not yet completed.

3.2.1 Locking

Concurrent access to shared objects is controlled by locking objects.

Locks are acquired before using an object and released sometime after the process has finished with it.

Read Lock

- Reading can be shared between multiple processes
- Read lock is granted assuming there is no write lock on the object

Write Lock

- Writing is exclusive to a single process
- Write lock is granted only if there are no other locks on the object

Lock Conflicts

When a process attempts top acquire a lock on an object that is already locked by another process.

Handled by deadlock prevention rules (see 3.3.2).

Conflicts ([want to acquire] - [already acquired]):

- read-write
- write-read
- write-write

A read-read conflict is not possible.

Deadlock

When two processes attempt to lock two objects that are currently locked by the other process.

Can either by prevented (see 3.3.2) or permitted on the assumption that they can be recovered from.

Can be detected by monitoring number of "wait for" cycles in acquiring locks, not a simple task in a distributed system.

3.3 Two Phase Locking

Two phase locking ensures the consistency (C) and independence (I) ACID properties.

The schedule S being processed must be serializable.

Assume that:

- Deadlocks are already dealt with by some means
- Client C knows at the start:
  - The set of all objects accessed during the transaction
  - The number of times each object is accessed

Release an acquired lock only if:

- No new locks are going to be acquired
- The lock to be released will never need to be re-acquired

3.3.1 Rules

- A transaction must obtain a lock on an object before using it
- In the growing phase locks are acquired as they are needed and not released
- In the shrinking phase acquired locks are released and no new locks acquired
- At some point the transaction will have acquired all the locks it needs to acquire, this is called the "lock point"

Theorem is that two phase locking only permits serializable schedules.

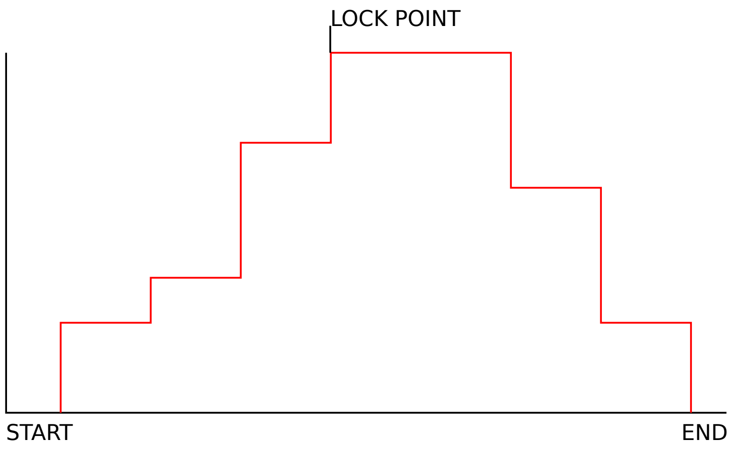


Figure 4: Two Phase Locking

When releasing locks sequentially (as per figure 4) it is possible that a transaction T_1 may release the lock on an object o_1 which is then locked by transaction T_2.

If T_1 aborts before completion then T_2 will have read dirty data from o_1 and will also need to abort (cascade abort).

This can be avoided using strict two phase locking where all the locks are released simultaneously at the end of the transaction.

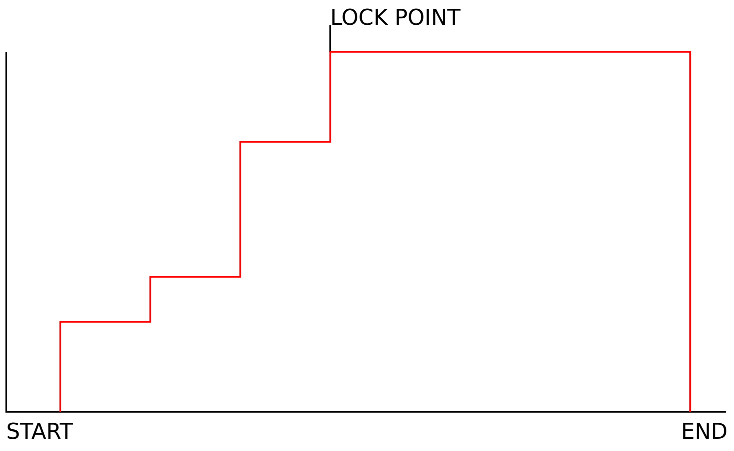


Figure 5: Strict Two Phase Locking

Strict two phase locking ensured the consistency (C) and independence (I) ACID properties.

3.3.2 Deadlock Prevention

Prevent wait for cycles caused by two transactions attempting to acquire locks on objects held by the other transaction.

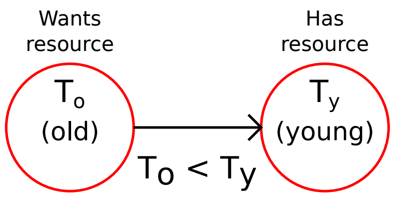
Simple approach

- Transactions never wait
- If lock on object is held by another transaction T_2 then T_1 aborts
- Simple and easy to implement
- Effective if conflicts are rare

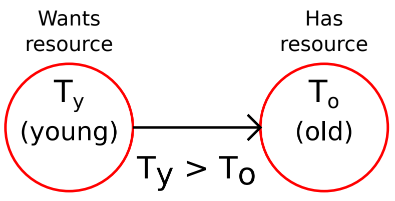
Timestamp Based

Another approach is to permit waiting only if there is no danger of deadlock.

To do this transactions are assigned timestamps such that they are totally ordered, these timestamps are then used in conflict resolution.



(a) Old - Young



(b) Young - Old

Figure 6: Deadlock prevention scenarios

Wait-Die

- Rule: If $T_1 < T_2$ then T_1 waits, otherwise T_1 aborts

- Old transactions (figure 6a) wait, newer transactions (figure 6b) die

Wound-Wait

- Rule: If $T_1 < T_2$ then T_1 forces T_2 to abort, otherwise T_1 waits
- Old transactions (figure 6a) abort/wound younger transactions, newer transactions (figure 6b) wait

3.4 Commit Protocol

The commit protocol ensures the atomicity (A) and durability (D) ACID properties.

- Changes to objects are made on copies of the object
- If transaction is aborted these copies are discarded
- If transaction is committed then changes are saved back to non-volatile storage
- This required a multi-pass commit protocol as changes must be made on either every node or no nodes

3.4.1 Two Phase Commit

- Termination of the transaction is carried out by the client (coordinator).
- The first phase determines the outcome of the transaction
- The second phase is used to enforce the outcome
- Every node maintains a transaction log:
  - Maintains information about transactions the node is taking part in
  - Writing to log file is ensured to be atomic
  - Stored on non-volatile storage
- Every node has a recovery manager that executes when the node recovers from a crash
  - Scans transaction log and determines actions required
  - Tries to terminate all transactions that were active at the time of crash

Phase 1

Coordinator

1 Send *get ready* message to all servers

2 Wait with timeout for reply from all servers

3 If all servers replay with `yes` then verdict is commit, if any one server replies with *no* then verdict is abort

4 If verdict is commit then write transaction ID and server node addresses to the transaction log

Server

1 Wait with timeout for *get ready* command from coordinator

2 If timeout elapses then abort transaction

3.1 If server does not want to commit then reply *no*

3.2 Otherwise write transaction ID and coordinator node address to the transaction log and reply *yes*

Phase 2

Coordinator

1 Send verdict to all servers

Server

1 Wait without timeout for verdict from coordinator

2.1 If verdict is commit then store the copies of the objects on the stable data store

2.2 Otherwise if verdict is abort then discard the copies

Remarks

- In phase 1 any server can replay *no*, this acts as a veto for the entire transaction, only one server needs to reply `no` for the transaction to be aborted
- In phase 1 if a server replies *yes* then it is in a position to either commit or abort as it holds both original and modified versions of the objects
- Once a server replies *yes* it must wait for the verdict from the coordinator
- If information about a transaction is missing in a transaction log then the transaction will eventually abort (this can happen if a node crashes just before the log is written)
- If a server crashes after writing to the transaction log then the recovery manager reads this log to decide (after communication with the coordinator) if it should commit or abort
- This protocol is a blocking protocol
- If the coordinator crashes in phase 2 and is not restarted then some servers may wait indefinitely

3.5 Lock Free Transactions

- Alternative to Two Phase Locking
- Still use Two Phase Commit to ensure atomicity and durability

3.5.1 Sequential Ordering

Good performance when two transactions accessing the same object is unlikely.

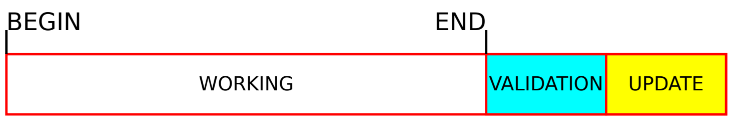


Figure 7: Sequential Ordering Phases

Each object has a write timestamp which records the last time the value of an object was modified.

Three phases:

1 Working Phase
   - On read: return most recently committed (MRC) value from server
   - On write: client retains all writes

2 Validation Phase

    - Conflict resolution
    - Result of conflict resolution defines if transaction will commit or abort

3 Update Phase

    - Only performed if validation phase was successful
    - Perform two phase commit if any writes took place

Working Phase

For every read operation:

1 Read the MRC value v of object o from server s

2 Store $< o, v, s >$ in read set RS

For every write operation:

1 Store $< o, v, s >$ in write set WS

    - Where v is the new value to be written to object o on server s

The state of the server is not changed during the working phase.

Validation Phase

- After transaction ends it receives a sequence number n (becomes T_n).
- T_n cannot enter the validation phase until T_{n-1} has completed both validation and update phases

When T_n enters the validation phase iterate through every value in RS and check that the value on the server matches the value read by T_n.

This check can also be performed by reading the write timestamp of each object as it is read (in working phase) and comparing this with the write timestamp read in the validation phase. For validation to succeed the timestamps must match for all objects.

If all values match (i.e. no values has been changed on a server during the transaction) then validation is successful, otherwise validations has failed and the transaction must be aborted (a dirty read has taken place).

Update Phase

- Skip if WS is empty, transaction is committed/successful
- Execute two phase commit with every server in WS
- *get ready* message will include WS
- Write timestamp of each modified object is set to n

3.5.2 Timestamp Ordering

- Each transaction has a timestamp $T_n.ts$, originating from a synchronised physical clock
- Timestamps impose total order
- Cannot know how many transactions precede it based on timestamp (as could be done using sequence numbers in sequential ordering)

Read/Write Issues

Dirty Read

A transaction (T_r) reads a value that is stored on the server but yet to be committed (modified by T_w), T_w later aborts.

Eliminated by:

- $T_r > T_w$: force T_r to wait until T_w commits
- $T_r < T_w$: consider allowing T_r to precede

Stale Read

Between a transaction T_r reading a value and committing its results another transaction T_w modified the value and commits.

Eliminated by:

- Stopping T_w writing until after T_r has read

Server side data

For each object:

Read timestamp $RTS_{max}(X)$

Maximum timestamp of transactions that were permitted to read the value of X

Write timestamp $WTS(X)$

Timestamp of last committed write to X

Server also maintains tentative versions of objects that have been updated prior to them being committed.

Write rules

A write attempt is rejected if it is deemed too late with respect to other transactions.

T_w is allowed to tentatively write to X if and only if:

1 $T_w > WTS(X)$
  A later transaction has not already committed a new value of X

2 $T_w \geq RTS_{max}(X)$
  A later transaction has not read the value of X

Read rules

Reject a read attempt if it is too late with respect to $WTS(X)$.

Force the transaction to wait if it is too early with respect to pending tentative writes.

"all versions" includes the most recently committed and every tentative version.

End Transaction

Execute two phase commit:

- Transactions are not required to commit in timestamp order
- A server replies with *no* for transaction $T_c$in phase 1 if $T_c < WTS(o)$ of an object o that T_c wants to persist

```
if Tr > WTS(X)
  Tmx = latest X such that Tmx < Tr
  if Tmx is committed
    read MRC(X)
  else
    wait until Tmx commits or aborts
    read MRC(X)
else
  abort
```

Listing 1: Read rule pseudocode

3.6 Comparison of commit protocols

- Two phase commit and timestamp ordering are not equivalent
- Each operate on different subsets of all serializable schedules