Assertional Verification of a Majority Consensus Algorithm for Concurrency Control in Multiple Copy Databases

Nicolien J. Drost and J. van Leeuwen

RUU-CS-88-13
March 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Assertional Verification of a Majority Consensus Algorithm for Concurrency Control in Multiple Copy Databases

Nicolien J. Drost and J. van Leeuwen

Technical Report RUU-CS-88-13
March 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

Abstract

The majority consensus algorithm of Thomas [Thom79] for concurrency control in multiple copy databases is proved correct, using system-wide invariants. The specification of the algorithm is extended to a more formal and more complete form. It is shown that the algorithm as given by Thomas does not guarantee internal consistency, but that a slightly modified form does. We also describe a modification in which votes need not be remembered.

Keywords: distributed databases, data replication, update synchronization, concurrency control, timestamps, assertional proof, system-wide invariant.

Contents

Chapter 1

Introduction

In this paper we explore the practical problems of a distributed database system in which replicated files are kept. An update mechanism for distributed databases in which multiple copies of the data are present should preserve the consistency of the data. Here two types of consistency are important:

1. mutual consistency: different copies of a data item in the database contain the same value at the same time.

2. internal consistency: for each copy certain predicates on the values of different items hold at all times.

To ensure these properties, an update mechanism should be atomic (the update is executed fully or not at all, but not partly), isolated (a transaction cannot use the results of another transaction that is not yet accepted), durable (an accepted update is never undone), and serializable [CePe85]. Two definitions of serializable are given in Chapter 4.

Update mechanisms for distributed databases may be centralized or distributed. In a centralized mechanism there is one node that controls the update. In a distributed mechanism there is no such central node.

The most widely known mechanism for concurrency control in distributed databases is two-phase locking [CePe85]. The simplest version of two-phase locking is centralized. The coordinating process, often the process that initiated the update, first acquires locks on all data items it needs, then performs its actions, and afterwards releases all the locks it holds. Two-phase locking is often used together with a two-phase commit protocol to ensure atomicity of the update. More extended versions of two-phase locking and two-phase commit choose a new coordinator if the old one crashes.

A well-known distributed update mechanism for distributed databases is described by Thomas [Thom79]. He suggested to assign timestamps to each data item in each copy, and to each update request. To create an update request, data are read from one copy of the database. The request then circulates through the network and tries to accumulate "OK" votes from the nodes. If a request acquires a majority of OK votes it is accepted, and the update is executed at all sites. Otherwise the request is rejected. The idea underlying Thomas' algorithm is a fundamental one, and has been (and still is) applied repeatedly in the design of distributed algorithms that involve some kind of distributed version control.

In [Thom79] only an informal argument for the correctness of this update mechanism is given. This paper gives a formal correctness proof of the update algorithm in [Thom79] using system-wide invariants. This technique has been successfully applied to the verification of many protocols, and gives considerable additional insight into the algorithm [Kro78], [Knu81], [SvL85], [SvL87], [Tel87], [DrSc88]. Independently, a (less complete) version of Thomas' update algorithm was verified by Jonsson [Jon87] as an application of a proof system for transition systems.

In Chapter 2 a more extensive description is given of the update mechanism of [Thom79], in Chapter 3 we refine the description to a form suitable for the proof, Chapter 4 contains the proof, and Chapter 5 the discussion.

Chapter 2

Description of the update mechanism

2.1 Assumptions

In [Thom79] a number of assumptions about the network and the distributed database are given. The network consists of sites, connected by communication channels. Each site has a local clock. There is no global clock. Message delay is arbitrary.

The most important assumption is full replication: at all sites of the network a full copy of the database is present. The database consists of a collection of named elements. The nature of these elements is not important here. Each named element has a value and a timestamp associated with it. The database copy at each site is accessible only through a database managing process (DBMP), which resides at that site. Query and update accesses to the database are initiated by application processes (APs). DBMP-DBMP and DBMP-AP communication is only possible via messages.

Thomas uses two models for communication between DBMPs. In model 1 messages are never lost. In model 2 a site may crash, and thus lose an unresolved update request that was present in this site. It is assumed that no other information is lost.

2.2 The Update Mechanism

Thomas [Thom79] describes his majority consensus update mechanism by giving five rules:

1. DBMP/DBMP Communication Rule,
2. Voting Rule,
3. Request Resolution Rule,
4. Update Application Rule,
5. Timestamp Generation Rule.

If an AP wants to execute an update, it first requests values of database items plus their timestamps from a DBMP. The set of data elements used by the AP is called the base set. From these values the AP computes new values for the data elements to be updated.

The set of data elements to be updated is called the update set. The update set must be a subset of the base set.

The AP now makes an update request. This request contains the timestamps of the items in the base set, and the new values of the items in the update set. The update request also gets a timestamp, using the Timestamp Generation Rule. This timestamp is derived from the timestamps of the base variables, and the value of the local clock. The AP sends this request to a DBMP.

A DBMP that receives an update request compares the timestamps of the base variables of the request to the timestamps of the items in its own copy of the database. It also checks if this update conflicts with other request on which it has voted, but about which it has not yet received a decision. Two update requests **conflict** if an item in an update set of one request is also present in the base set of the other. The DBMP then uses the Voting Rule to determine how to vote on the request, and adds its vote to a set of votes, attached to the request.

After voting, the DBMP checks whether the request has accumulated enough votes to decide if the request should be accepted or rejected. If there are enough votes, the Request Resolution Rule is used to decide whether the request is accepted or rejected. Request plus decision are then broadcast to all other DBMPs. If there are not enough votes, the request is sent to another DBMP using the DBMP/DBMP Communication Rule.

If a DBMP decides to accept a request, or receives the decision that an update request is accepted, it applies the update using the Update Application Rule. Timestamps in the database copy for the modified data elements are set to the timestamp assigned to the update.

2.3 More detailed description of the rules

2.3.1 DBMP/DBMP Communication Rule

In [Thom79] two ways of forwarding update requests are given:

- Daisy chaining,
- Daisy chaining with timeout.

Daisy chaining is used if model 1 is assumed: no messages are lost. If a DBMP receives an update request it votes, and if a decision is not yet possible, it sends it to one other DBMP that has not yet voted. If model 2 is assumed (sites may crash and lose an update request) extra provisions are needed. A DBMP sets a timer when it forwards an unresolved request to another DBMP. If the timer expires before a decision about this request arrives, the DBMP again sends the request to a DBMP that has not yet voted. This need not be the same DBMP as before. After sending the DBMP again sets a timer.

2.3.2 DBMP Voting Rule

The voting rule is the basis for concurrency control. A DBMP receiving an update request checks whether the timestamps of the base variables of the request are current, i.e. equal to the timestamps of these items in its database copy. If this is the case, no conflicting updates were executed since the request was initiated. Also the DBMP checks whether the request conflicts with a **pending** request: a request on which the DBMP voted earlier but did not yet receive the decision. If all base variables are current and there are no conflicting pending requests, the DBMP votes OK. If a base variable in the request is "older" than the corresponding item in the database copy, the DBMP votes REJ. If most base variables of the request are current but some are "too new", then an accepted request updating these items must be on its way to the DBMP. So the DBMP defers voting until this other update is executed.

If a request conflicts with a pending request, the DBMP should not vote OK. If the pending request is executed, it makes the base variables of the received request obsolete. The DBMP could set the received request apart and defer voting until the decision about the pending request has arrived. But this could cause deadlock if DBMP i has request A pending and defers B, and DBMP j has request B pending and defers A. Therefore the timestamp of a request is also used as a priority. If the received request conflicts with a pending request of higher priority, the DBMP votes PASS, else it defers the request. This results in the following DBMP Voting Rule:

1. Compare the timestamps for the request base variables with the corresponding timestamps in the local database copy.
2. Vote REJ if any base variable is obsolete.
3. Vote OK and mark the request as pending if each base variable is current and the request does not conflict with any pending requests.
4. Vote PASS if each base variable is current but the request conflicts with a pending request of higher priority.
5. Otherwise, defer voting and remember the request for later consideration.

If the timeout and retransmit communication discipline is used (model 2), it is possible for a DBMP to be asked to vote on a request on which it has already voted. A DBMP is forbidden from changing its vote in such a case.

2.3.3 Request Resolution Rule

The basic idea is that the request should be accepted if a majority of the DBMPs has voted OK. The important property of majority consensus is that the intersection of any two majorities has at least one DBMP in common. This means that for any two requests that are accepted at least one DBMP voted OK for both. For model 1 the Request Resolution Rule is:

1. After voting on a request (R):

   a) if the vote was OK and a majority consensus exists:
      Accept R and notify all DBMPs and the AP that R was accepted.

   b) if the vote was REJ:
      Reject R and notify all DBMPs and the AP that R was rejected.

   c) if the vote was PASS and a majority consensus is no longer possible:
      Reject R and notify all DBMPs and the AP that R was rejected.

d) otherwise:
Forward R and the votes accumulated for it so far to a DBMP that has not yet voted on it.

2. After learning that a request (R) has been resolved:
   a) if R was accepted:
      i) Apply R to the local copy of the database using the Update Application Rule.
      ii) Reject conflicting requests that were deferred because of R.
   b) if R was rejected:
      i) Use the voting rule to reconsider conflicting requests that were deferred because of R.

If model 2 holds and daisy chaining with timeout is used, more copies of a request may be present in the network. To prevent that a request is both accepted and rejected by the DBMP set, the DBMP Request Resolution Rule is adapted as follows:

1. b) if the vote was REJ and a majority consensus is no longer possible:
   reject R and notify all DBMPs and the AP that R was rejected.

In model 1 one REJ vote suffices to reject a request. In model 2 a majority of REJ and PASS votes is needed to reject a request, which makes REJ and PASS votes in fact equivalent.

2.3.4 Update Application Rule

The Update Application Rule ensures that updates, executed on a data item in a copy of the database, are executed in timestamp order. In fact it ensures that an update that arrives too late, i.e., when another update on this item with higher timestamp is already executed, is not executed any more on this item.

The Update Application Rule is:

1. To apply an update (R) to a variable (v) in a database copy, compare the timestamp of R (T) with the timestamp of the variable in the database (Tv): If Tv<T, modify v and set Tv to T; otherwise, omit the update to v since it is obsolete.

For updates with more than one update variable it may happen that some of the assignments to data elements are performed and others are omitted as obsolete.

2.3.5 Timestamp Generation Rule

When a DBMP initiates a new request, it also generates a timestamp for this request. These timestamps should be unique, and timestamps generated by one DBMP should increase monotonically in time. A timestamp generated by a DBMP i is a pair (T,i), with T a time and i a node identification. The relations equality, greater than, and less than for timestamps are defined as follows:

Let T1=(C1,d1) and T2=(C2,d2) be timestamps.

Equality	: T1=T2 iff C1=C2 and d1=d2.
Greater than	: T1>T2 iff C1>C2, or C1=C2 and d1>d2.
Less than	: T1<T2 iff C1<C2, or C1=C2 and d1<d2.

The timestamp generation rule used by a DBMP i to assign a timestamp to a request R is:

1. Let T=1+max(time, max({Tb}))
   where "time" is the time obtained by DBMP i from its local clock and the {Tb} are the time-parts of the timestamps for R's base variables. The timestamp for R is (T,i). To ensure that its local clock increases and that it never regenerates the same timestamp, DBMP i also sets its local clock to T.

Chapter 3

A Description using State Variables and Events

To prove the update algorithm correct we need a description of everything relevant that may happen in the system while the algorithm is executing. For a proof using invariants this description must be in the form of state variables, and events that send and receive messages and change the values of the state variables. To prove an invariant correct it must be checked that all events preserve its correctness. Therefore events must be atomic. Either all actions of an event are executed, or none at all. This raises the problem of granularity of events. Events could be very small: they could consist of one action, like sending, receiving, or modifying the value of one variable. This is comparable with the situation in a real computer that does not use checkpoints. But a great number of small events leads to a lengthy proof [Lam82]. Events consisting of more actions correspond to a system with checkpoints. Big events have another disadvantage: it becomes difficult to see the results of an event, and this also complicates the proof. We choose to make events that start with the reception of a message, then optionally perform computations and change values of state variables, and end with the sending of zero or more messages. If sending of a message is not triggered by the reception of another message, then the corresponding event consists of performing computations and changing state variables if needed, and sending one or more messages.

In [Thom79] the network consists of sites. Each site has a copy of the database and a DBMS process to perform actions on this copy. An update request is initiated by an Application Process (AP). We decided to make no difference between sites, DBMPs and APs, and to take nodes as entities. Nodes contain a database copy, initiate update requests, and read and write data items in this copy. So nodes have state variables and perform events. We shall first describe these for model 1.

3.1 Model 1

A description of variables and events of nodes is given in Figure 3.1 and Figure 3.2. Events and procedures are given in a Pascal-like notation. We assume that variables and operations for sets are present. a **into** A means: insert element a into set A. a **outof** A means: remove element a from set A. **for all** a $\in$ A **do** S0...Sn **od** means: execute statements S0...Sn for each element a in set A, the elements taken in arbitrary order. In boolean expressions **and** has a higher priority than **or** .

A request consists of timestamps of the base variables and new values for update variables. It also contains an update timestamp, and a set of votes. So the structure of a request A is:

- A.BaseSet : contains the names of the base variables,
- A.BaseTS(x) for each x ∈ A.BaseSet : the timestamp of base variable x. It consists of two parts: A.BaseTS(x).time and A.BaseTS(x).node,
- A.UpdateSet : contains the names of the update variables,
- A.UpValue(x) for each x ∈ A.UpdateSet : the new value for update variable x,
- A.UpTS : the update timestamp of A. Like A.BaseTS(x) it consists of two parts: A.UpTS(x).time and A.UpTS(x).node,
- A.VoteSet : contains (vote, node) pairs with nodes that voted already on this request.

There are two types of messages. A request message contains a request; a decision message contains a request and a decision: Accepted or Rejected.

As the only things that happen in a link are entering a message into it and removing a message from it, we did not model links as separate entities, but represented them by state variables of the nodes. Because nodes do not discriminate between messages from different links and do not receive messages in some prescribed order, we gave each node i a state variable Bag(i), which contains the messages sent to i but not yet received by i.

A node also contains a local copy of the database, and a set of pending requests: requests it has voted on, but about which it did not yet receive a decision. Each node has its own local clock. So the state variables of node i are:

- Bag(i) : messages sent to i but not yet received by i,
- Dbase_i(x) for each item x in the database : the local copy of item x. It consists of two parts:
  - Dbase_i(x).value : the value of this copy of item x,
  - Dbase_i(x).ts : the timestamp of this copy of item x.
- PendingSet(i) : the set of pending requests,
- Clock(i) : the local clock of i.

A node i also uses additional variables Vote and Decision. These are not considered as state variables.

Nodes may receive an update request, receive a decision about an update request, or spontaneously generate and send a new update request. So the events of a node are:

1. Receive request,
2. Receive decision,
3. Generate request.

```
Variables of node i:

Dbase_i(x).value for each data item x
Dbase_i(x).ts for each data item x
PendingSet(i)
Bag(i)
Clock(i)

Events of node i:

1) { A ∈ Bag(i) and not Deferred(A) }
   Receive(A);
   Vote := DetermineVote(A);
   Decision := Decide(A,Vote);
   Execute(Vote,Decision, A);

2) { (A,Decision) ∈ Bag(i) }
   Receive((A,Decision));
   if Decision=Accepted
   then ApplyUpdate(A)
   fi;
   Remove(A) from PendingSet(i);

3) A := MakeRequest;
   Send(A) to NextNode(A);

Procedures of node i:

procedure Deferred(A) : boolean;
   if ∃x ∈ A.BaseSet :
         A.BaseTS(x) > Dbase_i(x).ts
      or ∀x ∈ A.BaseSet :
         A.BaseTS(x) = Dbase_i(x).ts
         and ( ∀P ∈ PendingSet(i) :
         Conflict(A,P) ⟹
               P.UpTS > A.UpTS))
   then Deferred := false
   else Deferred := true
   fi;

procedure DetermineVote(A) :
         VoteType;
   if ∃x ∈ A.BaseSet:
         A.BaseTS(x) < Dbase_i(x).ts
   then DetermineVote := REJ
   elsif ∀x ∈ A.BaseSet:
               A.BaseTS(x)=Dbase_i(x).ts
         and ∀P ∈ PendingSet(i):
               not Conflict(A,P)
   then DetermineVote := OK
   elsif ∀x ∈ A.BaseSet:
         A.BaseTS(x)=Dbase_i(x).ts and
         ∀P ∈ PendingSet(i):
         Conflict(A,P) ⟹
               A.UpTS > P.UpTS
   then DetermineVote := PASS
   fi;

procedure Conflict(A,B) : boolean;
   if ∃x:
   x ∈ A.BaseSet and x ∈ B.UpdateSet
   or x ∈ A.UpdateSet and x ∈ B.BaseSet
   then Conflict := true
   else Conflict := false
   fi;

procedure Decide(A,Vote) : DecisionType;
   Decide := Undecided;
   Case Vote of
   OK : if Majority(A.VoteSet)=OK
         then Decide := Accepted
         fi;
   REJ : Decide := Rejected;
   PASS : if Majority(A.VoteSet)=PASS
         then Decide := Rejected;
   Esac ;

procedure Execute(Vote,Decision,A);
   Case Decision of
   Undecided : (Vote,i) into A.VoteSet;
         A into PendingSet(i);
         Send(A) to NextNode(A);
   Accepted : Broadcast((A,Decision));
         ApplyUpdate(A);
   Rejected : Broadcast((A,Decision));
   Esac ;

procedure MakeRequest : RequestType;
   var A: RequestType;
   Put base variable names
         into A.BaseSet;
   for all x ∈ A.BaseSet do
         A.BaseTS(x) := Dbase_i(x).ts od ;
   Put all update variable names
         into A.UpdateSet;
   Compute new values;
   for all x ∈ A.UpdateSet do
         A.UpValue(x):=
               new value of x;
   A.UpTS := MakeTimestamp(A.BaseSet);
   A.VoteSet := ∅;
   MakeRequest := A;
```

Figure 3.1: Variables and events of nodes and procedures for model 1

```
procedure ApplyUpdate(A);
  for all x ∈ A.UpdateSet
  do  if A.UpTS > Dbase_i(x).ts
      then Dbase_i(x).value := A.UpValue(x);
           Dbase_i(x).ts := A.UpTS
      fi
  od ;

procedure MakeTimestamp(A):
      TimestampType;
  var TS: TimestampType;
  TS.time := 1 + max(Clock(i),
      max{A.BaseTS(x) | x ∈ A.BaseSet});
  TS.node := i;
  Clock(i) := TS.time;
  MakeTimestamp := TS;

procedure Send(A) to i;
  A into Bag(i);

procedure Remove(A) from PendingSet;
  if ∃P ∈ PendingSet :
       P.UpTS = A.UpTS
  then A outof PendingSet
  fi;
procedure Receive(A);
  A outof Bag(i);

procedure NextNode(A): NodeType;
  provides the id. of a node that is
       not yet present in A.VoteSet;

procedure Majority(VoteSet):
            MajorityType;
  gives OK if a majority of OK votes
            is present in VoteSet,
       PASS if a majority of OK votes
            is no longer possible,
       Undecided otherwise.
```

Figure 3.2: Rest of procedures for model 1

If a node receives a request, it votes, and decides, or sends the request to another node for voting. After making a decision, a node broadcasts the request plus the decision to all other nodes, and applies the update if the decision was Accepted. A node receiving a decision message removes the request from the PendingSet (if present), and applies the update if the decision was Accepted. Generation of an update request consists of reading the base variables and their timestamps, computing a timestamp for the request, and sending the request to a node for voting.

The rules in [Thom79], see also section 2.2, are translated into procedures. We will describe these for each rule. In figure 3.1 and 3.2 the detailed "code" is given.

3.1.1 Communication rule

If a node has voted and not taken a decision, procedure NextNode chooses in some unspecified way a node from the nodes that did not vote yet on this request. This results in daisy chaining.

3.1.2 Voting Rule

In [Thom79] a DBMP that received a request may vote OK, REJ, or PASS, or defer voting and remember the request for later consideration. We modeled this by postulating that a node can only receive undeferred requests. Procedure Deferred tests whether a request is deferred with respect to the local database copy. A request is deferred if it contains "too recent" base variables (and no obsolete ones), or if all its base variables are current and it conflicts with a pending request with a lower timestamp. If a node receives an undeferred request, procédure DetermineVote determines the vote according to the Voting Rule in [Thom79]. Procedure Conflict is used to test whether the request conflicts with a pending request.

3.1.3 Request Resolution Rule

When a node has voted on a request, it calls procedure Decide. This procedure decides whether there are enough votes to take a decision, and makes a decision according to the Decision Rule in [Thom79]. An unspecified procedure Majority decides whether there is a majority of OK votes, a majority of OK votes is no longer possible, or there are as yet too few votes. Majority is a function from sets of (vote,DBMP) pairs to {OK, PASS, Undecided}. The essential properties that are required of it are:

- If Majority(V1)=OK and Majority(V2)=OK, then V1 and V2 have at least one DBMP in common that voted OK in both cases.
- If Majority(V1)=OK and Majority(V2)=PASS, there is at least one DBMP that voted OK in V1 and PASS or REJ in V2.
- If for all V2 that contain V1 Majority(V2) $\neq$ OK, then Majority(V1) = PASS.

Procedure Execute takes action upon this decision. If the decision was Undecided, the vote plus an identification of the node are added to the VoteSet of the request, a copy of the request is placed into the PendingSet, and the request is sent to another node. Otherwise the request plus the decision are broadcast to all other nodes. If the decision was Accepted, the update is applied to the local database copy (see Update Application Rule).

The second part of the Decision Rule in [Thom79] gives the rules for a node receiving a decision (event 2 in figure 3.1). Our description of this algorithm does not follow these rules fully:

- If a request A was accepted, conflicting requests that were deferred because of A are not rejected. Such requests are now no longer deferred, because A is no longer pending, and may be received in a later event, and will then be rejected.
- If A was rejected, we do not in the same event use the voting rule to reconsider conflicting requests that were deferred because of A. The requests are no longer deferred, and may be received by the node.

These changes make our version a more general form of the algorithm in [Thom79], and thus do not invalidate our proof of the algorithm in [Thom79].

Also, it is not specified in [Thom79] what happens to requests that are deferred because some of their base variables are too recent. In our version of the algorithm they may be received by the node as soon as the timestamps in the local database copy have increased enough to make these base variables current or obsolete.

3.1.4 Update Application Rule

Procedure ApplyUpdate applies an update request according to the Update Application Rule in [Thom79]: an update A to data item x in node i is only applied if A.UpTS > Dbase_i(x).ts.

3.1.5 Timestamp Generation Rule

Procedure MakeTimestamp generates a timestamp for a new request A according to the Timestamp Generation Rule in [Thom79]. A timestamp is a pair (T,i) with i the identification of the node and
T = 1 + max(Clock(i), max{ A.BaseTS(x).time | x ∈ A.BaseSet }).

3.2 Model 2

In model 2 a node may vote more than once on a request, and must vote the same every time on a same request. Therefore the nodes have an extra state variable: OldVoteSet, which contains (Request.UpTS, Vote) pairs.

To handle it, event 1 is changed in such a way that a node looks first whether there is already a vote for this request, and only calls DetermineVote if this is not the case.

Two new events are added:

- Event 4 models the timeout. At some unspecified time a node decides to send a copy of a pending request to some arbitrary node that has not yet voted on this request.
- Event 5 models the loss of a request by a node. The node receives a request and does nothing with it. No other information is lost.

Procedure Decide is changed. A node no longer decides Rejected upon one REJ vote, but only if a majority of OK votes is no longer possible. This is needed because more request messages for one request may be generated, and so more than one node may make a decision about the same request. If a node could decide Rejected upon one REJ vote, decisions of different nodes about the same request could be different.

In procedure Execute a request is removed from the PendingSet (if present), if a decision is made by the node about this request. All other variables, events and procedures are the same as in model 1. The changes are given in Figure 3.3.

3.3 Global Observer

An invariant is a predicate on state variables of the system. It gives no information about the sequence of events in the system. As some of the claims in [Thom79] are about the sequencing of updates, we need a mechanism to describe some aspects of the history of an execution of the algorithm. Therefore we add a Global Observer. This Global Observer records facts about the events that happen in the system, but does not influence these events. Therefore the Global Observer has its own state variables, to which values are assigned if certain events happen in the system. The Global Observer has its own monotone clock, called GlobalClock.

The Global Observer records all requests A that are initiated. The variables used in model 1 are:

- A.BaseSet
- A.BaseTS(x) for each x ∈ A.BaseSet
- A.UpdateSet

New variable of node i:
OldVoteSet(i)

New and changed events of node i, respectively:

```
1) { A ∈ Bag(i) and not Deferred(A)}
   Receive(A);
   if (A.UpTS,OldVote) ∈ OldVoteSet(i)
   then Vote := OldVote
   else Vote := DetermineVote(A);
         (A.UpTS,Vote) into OldVoteSet
   fi;
   Decision := Decide(A,Vote);
   Execute(Vote,Decision, A);

4) { A ∈ PendingSet(i) } /*timer expires*/
   Send(A) to NextNode(A);

5) { A ∈ Bag(i) } /*A is lost in node i*/
   Receive(A);
```

Changed procedures:

```
procedure Decide(A,Vote) : DecisionType;
  Decide := Undecided;
  Case Vote of
      OK : if Majority(A.VoteSet)=OK
            then Decide := Accepted
            fi;
      REJ,PASS :
            if Majority(A.VoteSet)=PASS
            then Decide := Rejected;
            fi;
  Esac ;

procedure Execute(Vote,Decision,A);
  Case Decision of
      Undecided : (Vote,i) into A.VoteSet;
            A into PendingSet(i);
            Send(A) to NextNode(A);
      Accepted : Broadcast((A,Decision));
            ApplyUpdate(A);
            Remove(A) from PendingSet(i);
      Rejected : Broadcast((A,Decision));
            Remove(A) from PendingSet(i);
  Esac ;
```

Figure 3.3: Changes in variables, events, and procedures for model 2

- A.UpTS
- A.VoteSet

The Global Observer also records:

- A.InitNode : the node that initiated A
- A.Decision : the decision taken about A.
- A.DecisionTime : the global time when the decision was made about this request.
- A.DecisionKnown(i) : registers whether node i received a decision about request A.

In model 2 more messages containing the same request may be present, so more sets of votes and more decisions of one request are recorded by the Global Observer. The new variables are: A.VoteSet(n) and A.Decision(n), one for each copy. Only the time of the first decision about a request is recorded.

Values are assigned to the variables of the Global Observer when a request is initiated, a decision is taken about a request, or a decision is received by a node. GlobalClock is increased at the end of each event in which state variables of the Global Observer are changed.

3.4 Initialization

When execution of the algorithm starts, each copy of a data item must have the same timestamp. Also all copies must have the same value. No request messages, decision messages, or pending messages may be present. So the initialization may be given as:

$\text{Dbase}_i(x).\text{value} := \text{initvalue}(x)$ for data items x,
$\text{Dbase}_i(x).\text{ts} := \text{InitTS}(x)$ for data items x,
$\text{Clock}(i) := \text{InitClock}(i)$,
$\text{Bag}(i) := \emptyset$,
$\text{PendingSet}(i) := \emptyset$.

The initial clockvalue for each node is arbitrary, but for practical reasons these clockvalues and the timestamps of the data items in the database should not vary greatly, for then many of the first requests will be rejected or deferred.

Before a request is initiated, the corresponding variables of the Global Observer have values $\emptyset$ (set variables), ∞ (timestamps), false (boolean variables), or undefined (the rest). The GlobalClock is initialized to an arbitrary value.

Chapter 4

The Proof of Thomas' Update Algorithm

A correctness proof for an update algorithm must prove three things:

1. The algorithm "does what it should do",
2. The algorithm does not cause deadlock,
3. The algorithm eventually terminates if no more new tasks (requests) are generated.

Necessary properties for correct execution of an update algorithm for distributed databases are atomicity, isolation, durability, and serializability [CePe85]. Atomicity of update transactions is guaranteed in the majority consensus algorithm because an update is not applied before the request is accepted. Isolation is guaranteed by the same reason. Durability is guaranteed because the algorithm never undoes an update, and it is assumed that database values are not affected by crashes.

The notion of serializability in the theory of centralized databases is well-understood. In [CePe85] serializability for centralized databases is defined as follows:

Definition A **schedule** is a sequence of operations performed by transactions.

Definition A schedule is **serializable** iff it is computationally equivalent to a serial schedule.

In [CePe85] two conditions are given that together are sufficient to ensure that two schedules are computationally equivalent:

1. Each read operation reads data which are produced by the same write operations in both schedules,
2. The final write operation on each data item is the same in both schedules.

More definitions exist for serializability in distributed databases. A commonly accepted definition is the following [CePe85] :

Definition [CePe85] The distributed execution of a set of transactions $T_1...T_n$ is **serializable** iff:

1. all local schedules are serializable,
2. there exists a total ordering of $T_1 ... T_n$ such that each local schedule is equivalent to a serial schedule that has an order of transactions consistent with this total order.

This form of serializability guarantees that a data item is not modified between the time a transaction reads a data item and writes it.

Thomas [Thom79] uses a slightly different definition of serializability:

Definition Let $A_1 ... A_n$ be the set of *accepted* updates of an execution E of an update algorithm. The execution E is **serializable** according to [Thom79] iff there exists an ordering of $A_1 ... A_n$ which, if executed on a one copy database with the same initial values for the items, produces the same update values for the items as in execution E.

To prove that executions of the majority consensus algorithm are serializable according to this second definition, we need an extra definition [Thom79]:

Definition Let A and B be update requests.
A **must precede** B, notation A<B, iff $A \neq B$ and
(A reads a variable x that is also in B's UpdateSet, and the timestamp of A's reading of x is smaller than the timestamp of B's update,
or
A updates a variable x that is also in B's BaseSet, and the timestamp of A's update is smaller than or equal to the timestamp of B's reading of x.)

Hence, for $A \neq B$: $A<B \iff$
$\exists x(x \in A.\text{BaseSet}$ and $x \in B.\text{UpdateSet}$ and $A.\text{BaseTS}(x)<B.\text{UpTS})$
or $\exists x(x \in A.\text{UpdateSet}$ and $x \in B.\text{BaseSet}$ and $A.\text{UpTS} \leq B.\text{BaseTS}(x))$.

We will prove that the relation "must precede" for a set of accepted updates of an execution of the majority consensus algorithm can be transformed to a total order that preserves this relation.

Full mutual consistency would be reached if all copies of the database are the same per data item at each moment. But this is not possible in the majority consensus algorithm, because the decision to execute an update request is taken at one node, and there is no bound on message delay. Therefore the decision about a request will reach the nodes at different times. Two accepted updates may even arrive at two nodes in reverse order. If nodes i and j decide at about the same time to accept requests A and B, respectively, node i first executes the update of A and then of B, and node j first executes B and then A.

A weaker form of mutual consistency is: all copies of a database item have the same sequence of values in time, but changes do not necessarily occur at the same moment. But even this is too strong for the majority consensus algorithm, because two updates need not arrive at different nodes in the same order. However, the following can be proved: the updates, executed on a copy of a data item, are a co-infinite subset of all updates of this data item, and are executed in a fixed order.

We give a proof of the consistency property for both model 1 and model 2. For model 1 we prove:

1. Only one node decides about a request.

2. The accepted updates of an execution of the majority consensus algorithm are serializable according to the definition in [Thom79].
3. Updates, executed on a data item in one copy, are a co-infinite subset of all updates, executed on this item, and are executed in a fixed order.
4. The algorithm does not cause deadlock.
5. The algorithm eventually terminates if no more new requests are generated.
6. If no more new requests are initiated, all database copies converge to the same value.
7. The algorithm takes correct decisions.
8. Every request is decided.

For model 2 we prove the same statements, with one exception. Statement 1 becomes:

1. All decisions made about a request are the same.

These statements are proved using invariants. An invariant is a predicate on state variables of the system. An invariant should hold at any moment during an execution of the algorithm that is being analyzed. An invariant is proved correct by verifying that it holds at initialization and that no event can make it false.

Many of the invariants we prove are of the form X $\Longrightarrow$ Y, e.g.
A $\in$ PendingSet(i) $\Longrightarrow$ $\exists$j : A $\in$ Bag(j) or (A,Decision) $\in$ Bag(i).
Proving such invariants consists of:

1. Checking that the invariant holds after initialization. Often this is trivial because the premise does not hold.
2. Verifying that an event cannot make the premise A true while the conclusion B is false.
3. Verifying that an event cannot make the conclusion B false while the premise A is true.

For step 2 and 3 we list all actions that could make the invariant false (the "dangerous" actions), and check each event in which one or more of these actions could occur. In the proof we repeatedly refer to the specification of Thomas' algorithm given in figure 3.1 and 3.2, by using the indices and names introduced there.

4.1 Proof for model 1

4.1.1 A decision about a request is only taken once

Lemma 1.1 There is at most one request message present of request A.

Proof:
The dangerous actions are:

- A request message is sent.
  1) i receives request message A, votes, and sends a request message. Then i sends no more than one message.
  3) i initiates a new request. Then there was no message of this request in the system, and i sends only one request message during the event. □

Theorem 1 Only one decision is taken about a request A.

Proof:
If i decides on request A, i receives a request message with A, and sends no request messages during the event. There is at most one request message present (lemma 1.1), and a request message is only generated once, hence no second decision can be taken. □

4.1.2 Serializability

We prove that the set of accepted updates of an execution of the algorithm is serializable as defined in [Thom79]. We first prove that timestamps of requests are unique. We use A,B,... to denote requests:

Lemma 2.1 CreatingNode(A)=i $\Longrightarrow$ A.UpTS.time $\leq$Clock(i).

Proof:
The dangerous actions are:

- CreatingNode(A):=i.
  3) i creates request A. Then A.UpTS.time gets a value, and Clock(i):= A.UpTS.time. So the conclusion holds after the event.

- A.UpTS changes its value.
  3) i creates request A. This case is already treated.

- Clock(i) changes its value.
  3) i creates another request B. Then Clock(i) increases its value, so the conclusion still holds after the event. □

Lemma 2.2 A $\neq$ B $\Longrightarrow$ A.UpTS $\neq$ B.UpTS.

Proof:
The dangerous actions are:

- A.UpTS changes its value.
  3) Node i initiates request A.
  If B is not yet initiated, then B.UpTS = ∞.
  If B is initiated by another node than i, B.UpTS.node $\neq$ A.UpTS.node.
  If B is initiated by i, then B.UpTS.time $\leq$Clock(i) before the event (lemma 2.1). And A.UpTS.Time := 1 + max(Clock(i), max{A.BaseTS(x).time | $x \in$ A.BaseSet}). So A.UpTS.time > B.UpTS.time, and A.UpTS $\neq$ B.UpTS.

- B.UpTS changes its value.
  Analogous. □

If node i has already voted on request A, then A is in PendingSet(i), or i knows the decision about A:

Lemma 2.3 (Vote,i) $\in$ A.VoteSet $\Longrightarrow$ A$\in$ PendingSet(i) or A.DecisionKnown(i).

Proof:
The dangerous actions are:

- (Vote,i) is put into A.VoteSet.
  1) i receives A and votes. If i votes OK or PASS and takes no decision, then A is put into PendingSet(i). If i takes a decision A.DecisionKnown(i) becomes true.

- A is removed from PendingSet(i).
  2) i receives (A,Decision). Then A.DecisionKnown(i) becomes true. □

If the update of A is executed in node i and x is an item in A's UpdateSet, then the timestamp of x in the database copy is greater than or equal to the timestamp of A:

Lemma 2.4 A.Decision=Accepted and A.DecisionKnown(i) and
x $\in$ A.UpdateSet $\Longrightarrow$:
Dbase$_i$(x).ts $\geq$A.UpTS.

Proof:
The dangerous actions are:

- A.Decision := Accepted.
  1) i receives A, x $\in$ A.UpdateSet, i votes OK, decides Accepted, and A.DecisionKnown := true. Then the update of A is executed in i and if Dbase$_i$(x).ts < A.UpTS then Dbase$_i$(x).ts := A.UpTS.

- A.DecisionKnown(i) := true.
  1) i receives A and decides Accepted. This case is already treated.
  2) i receives (A,Accepted). Then i executes the update of A and if Dbase$_i$(x).ts < A.UpTS then Dbase$_i$(x).ts := A.UpTS.

- Dbase$_i$(x).ts changes its value.
  2) i receives (B,Accepted). Then Dbase$_i$(x).ts increases. □

If i voted OK on request A and item x is in A's BaseSet, then the timestamp of x in the local database copy is greater than or equal to the timestamp of x in A.

Lemma 2.5 (OK,i) $\in$ A.VoteSet and x $\in$ A.BaseSet $\Longrightarrow$ Dbase$_i$(x).ts $\geq$A.BaseTS(x).

Proof:
The dangerous actions are:

- (OK,i) is put into A.VoteSet.
  1) i receives request A and votes OK on A. This is only possible if all base variables of A are current w.r.t. i's database copy, so Dbase$_i$(x).ts = A.BaseTS(x).

- Dbase$_i$(x).ts changes its value.
  2) i receives (B,Accepted) and x $\in$ B.UpdateSet. Then Dbase$_i$(x).ts may only increase (see procedure ApplyUpdate). □

If a decision message for request A is on its way to a node, or a node has received the decision, the (global) time when A was decided is smaller than the current global clock value:

Lemma 2.6 $\exists$i:(A,Decision) $\in$ Bag(i) $\Longrightarrow$ A.DecisionTime < GlobalClock.

Proof:
The dangerous actions are:

- (A,Decision) is put into Bag(i).
  1) some node receives A, decides, and places (A,Decision) into Bag(i).
  Then A.DecisionTime := GlobalClock, and GlobalClock is increased.

- GlobalClock changes its value.
  1)2)3) GlobalClock only increases. □

Lemma 2.7 $\exists$i : A.DecisionKnown(i) $\Longrightarrow$
A.DecisionTime < GlobalClock.

Proof:
The dangerous actions are:

- A.DecisionKnown(i) := true.
  1) i receives request A, votes, and decides. Then A.DecisionTime := GlobalClock. and GlobalClock is increased.
  2) i receives (A,Decision). Then A.DecisionTime < GlobalClock before the event, and GlobalClock only increases during the event.

- GlobalClock changes its value.
  1)2)3) GlobalClock only increases. □

Lemma 2.8 A.DecisionTime $\neq \infty \Longrightarrow$ A.Decisiontime < GlobalClock.

Proof:
The dangerous actions are:

- A.DecisionTime := value ($\neq \infty$).
  1) i receives A, votes, and decides. Then A.DecisionTime := GlobalClock, and GlobalClock is increased.

- GlobalClock increases its value.
  1)2)3) GlobalClock only increases. □

The next two lemmas say that if request A must precede request B and there is a node that voted OK for both, the decision about A was taken before the decision about B (in global time), or B was rejected.

Lemma 2.9 x $\in$ A.UpdateSet and x $\in$ B BaseSet and A.UpTS $\leq$B.BaseTS(x)
and (OK,i) $\in$ A.VoteSet and (OK,i) $\in$ B.VoteSet $\Longrightarrow$
A.DecisionTime < B.DecisionTime.

Proof:
The dangerous actions are:

- (OK,i) is put into A.VoteSet.
  1) i receives A and votes OK. If i has already voted OK on B, then $Dbase_i(x).ts \geq B.BaseTS(x)$ (Lemma 2.5). And $A.BaseTS(x) < A.UpTS \leq B.BaseTS(x)$ (for the first inequality see procedure MakeTimestamp). So i does not vote OK on A (procedure DetermineVote). Contradiction.

- (OK,i) is put into B.VoteSet.
  1) i receives B and votes OK. If i has already voted OK on A, then A $\in$ PendingSet(i) or A.DecisionKnown(i) (lemma 2.3).
  Suppose A $\in$ PendingSet(i). A and B conflict, so i does not vote OK on B (see DetermineVote). Contradiction.
  Suppose A.DecisionKnown(i). Then A.DecisionTime < GlobalClock (lemma 2.7). If B is decided in this event then B.DecisionTime := GlobalClock. If B is not yet decided then B.DecisionTime = ∞.

- A.DecisionTime changes its value.
  This happens only once, from ∞ to a finite value.

- B.DecisionTime changes its value.
  1) i receives B and decides.
  If A.DecisionTime < B.DecisionTime, then A.DecisionTime has a finite value < GlobalClock (lemma 2.8). And B.DecisionTime:= GlobalClock. □

Lemma 2.10 x $\in$ A.BaseSet and x $\in$ B.UpdateSet and A.BaseTS(x) < B.UpTS
and (OK,i) $\in$ A.VoteSet and (OK,i) $\in$ B.VoteSet $\Longrightarrow$
A.DecisionTime < B.DecisionTime
or B.Decision = Rejected.

Proof:
The dangerous actions are:

- (OK,i) is put into A.VoteSet.
  1) i receives A and votes OK. If i voted already on B, then B $\in$ PendingSet(i) or B.DecisionKnown(i) (lemma 2.3).
  Suppose B $\in$ PendingSet(i). A and B conflict, so i does not vote OK on A. Contradiction.
  Suppose B.DecisionKnown(i) and B was accepted. Then $Dbase_i(x).ts \geq B.UpTS$ (lemma 2.4). If A.BaseTS(x) < B.UpTS, then $A.BaseTS(x) < Dbase_i(x)$. Then i does not vote OK on A. Contradiction.
  Suppose B.DecisionKnown and B was rejected. Then the conclusion holds.

- (OK,i) is put into B.VoteSet.
  From this point the proof is identical to the proof of lemma 2.9. □

Now we are able to prove that, if A and B are accepted and A must precede B, then A was decided before B.

Lemma 2.11 A<B and A.Decision=Accepted and B.Decision=Accepted $\Longrightarrow$
DecisionTime(A) < DecisionTime(B).

Proof:
If A and B were both accepted, there is a node j with (OK,j) $\in$ A.VoteSet and (OK,j) $\in$ B.VoteSet. If A<B, then $\exists$x : x $\in$ A.UpdateSet and x $\in$ B.BaseSet and A.UpTS $\leq$B.BaseTS(x), or $\exists$x : x $\in$ A.BaseSet and x $\in$ B.UpdateSet and A.BaseTS(x) < B.UpTS. In the first case DecisionTime(A) < DecisionTime(B) because of lemma 2.9. In the second case this holds because of Lemma 2.10. □

We now prove: if A must precede B, then A reads its base variables before B updates them, and A writes its update variables before B reads them.

Lemma 2.12 Dbase_i(x).ts = t ($\neq$ initTS(x)) $\Longrightarrow$
$\exists$A : x $\in$ A.UpdateSet and A.UpTS=t and A.Decision=Accepted
and A.DecisionKnown(i).

Proof:
The dangerous actions are:

- Dbase_i(x).ts := t.
  1) i receives request X, x $\in$ X.UpdateSet, X.UpTS=t, and Dbase_i(x).ts < X.UpTS. Then i updates Dbase_i(x), and DecisionKnown(i) := true. So the conclusion holds after the event.
  2) i receives (X,Accepted), x $\in$ X.UpdateSet, X.UpTS=T, and Dbase_i(x).ts < t. Then i updates Dbase_i(x), and again the conclusion holds after the event. □

Lemma 2.13 B.CreateNode=i and x $\in$ A.UpdateSet and x $\in$ B.BaseSet
and A.UpTS $\leq$B.BaseTS(x) $\Longrightarrow$
A was already executed at i before B was created at i,
or $\exists$C: A<C and C<B and C was already executed at i.

Proof:
When i creates B, B.BaseTS(x) := Dbase_i(x).ts. If Dbase_i(x).ts $\neq$ initTS(x), then a request C exists with x $\in$ C.UpdateSet and C.UpTS=Dbase_i(x).ts and C is executed at i (lemma 2.12).
Suppose A.UpTS=B.BaseTS(x). Then A.UpTS=C.UpTS, so A=C (Lemma 2.2).
Suppose A.UpTS < B.BaseTS(x). Then A<C, for x $\in$ A.BaseSet and x $\in$ C.UpdateSet and A.BaseTS(x) < A.UpTS < B.BaseTS(x) = C.UpTS. And C<B, for x $\in$ C.UpdateSet, and x $\in$ B.BaseSet and C.UpTS $\leq$B.BaseTS(x) (C.UpTS = Dbase_i(x).ts = B.BaseTS(X)).
□

Lemma 2.14 A.CreateNode=i and x $\in$ A.BaseSet and x $\in$ B.UpdateSet
and A.BaseTS < B.UpTS $\Longrightarrow$
B has not been executed at i when A is created.

Proof:
When i creates A, A.BaseTS(x) := Dbase_i(x).ts. Suppose B was already executed in i. This means that B.Decision = Accepted and B.DecisionKnown(i). So Dbase_i(x).ts

$\geq$B.UpTS (lemma 2.4). But B.UpTS > A.BaseTS(x) = Dbase$_i$(x).ts. Contradiction. So B is not executed at i when A is created. □

Now we are able to prove serializability according to the definition in [Thom79]:

Theorem 2 For each execution of the majority consensus algorithm there exists a sequence of the accepted updates A$_1$...A$_n$ which, if executed on a one copy database with the same update values for the items produces the same update values for the items as in the original execution.

Proof:
Define a total order ($\sqsupset$) on A$_1$...A$_n$ as follows:
A$_i \sqsupset$A$_j$ iff A$_i$.Decisiontime < A$_j$.DecisionTime. If A$_i$< A$_j$, then A$_i \sqsupset$A$_j$ (lemma 2.11).
If A$_1$...A$_n$ are executed on a one copy database in the sequence defined by this total order, A$_i$ is executed before A$_j$ if A$_i \sqsupset$A$_j$. To ensure that the same update values are computed in the multi copy execution, only the order of conflicting updates is of importance. If A<B, there is no C such that A<C and C<B, and A writes an item that B reads, then A must write this item before B reads it. This indeed happens because of lemma 2.13. If A reads an item that B writes, then A must read this item before B writes it. This is the case because of Lemma 2.14. □

4.1.3 Update sequence on one copy of an item

Updates on one copy of a data item may be a subset of all updates on this item, but they must be executed in a fixed order. Let A and B be accepted updates from an execution of the majority consensus algorithm, and let $\sqsupset$ be the total order defined in the proof of Theorem 2.

Theorem 3 If A<B and x $\in$ A.UpdateSet and x $\in$ B.UpdateSet and A and B are both executed on item x in node i, then A is executed before B in node i.

Proof:
Suppose B is executed before A. When the decision about A arrives at i, Dbase$_i$(x).ts $\geq$B.UpTS (lemma 2.4). If A<B and x $\in$ A.UpdateSet, then A.UpTS $\leq$B.BaseTS(x). So A.UpTS < B.UpTS $\leq$Dbase$_i$(x).ts, and the update of A is not executed on item x (see procedure ApplyUpdate). □

4.1.4 Deadlock Freedom

To prove that no deadlock occurs during the execution of the majority consensus algorithm, we first need some lemmas:

Lemma 4.1 (A,Accepted) $\in$ Bag(i) $\Longrightarrow$ (A,Accepted) $\in$ Bag(j) or $\forall$x $\in$ A.UpdateSet : Dbase$_j$(x).ts $\geq$A.UpTS.

Proof:
The dangerous actions are:

- (A,Accepted) is put into Bag(i).
  1) Node k receives request A, votes OK, and decides Accepted. If $k \neq j$, then (A,Accepted) is put into Bag(j). If k=j, then j executes the update of A. If $x \in$ A.UpdateSet and $\text{Dbase}_j(x).ts <$ A.UpTS, then Dbasej(x).ts := A.UpTS (see ApplyUpdate). So after the event $\text{Dbase}_j(x).ts \geq$A.UpTS for all $x \in$ A.UpdateSet.

- (A,Accepted) is removed from Bag(j).
  2) j receives (A,Accepted) and executes the update of A. Then after the event $\text{Dbase}_j(x).ts \geq$A.UpTS for all $x \in$ A.UpdateSet.

- $\text{Dbase}_j(x)$.ts changes its value ($x \in$ A.UpdateSet).
  2) j receives (B,Accepted). Then Dbasej(x).ts can only increase. □

Lemma 4.2 $\text{Dbase}_j(x).ts < \text{Dbase}_i(x).ts \implies$
$\exists$A: (A,Accepted) $\in$ Bag(j) and $x \in$ A.UpdateSet and A.UpTS = $\text{Dbase}_i(x)$.ts.

Proof:
The dangerous actions are:

- $\text{Dbase}_i(x)$.ts becomes greater than $\text{Dbase}_j(x)$.ts.
  2) i receives (B,Accepted), B.UpTS > $\text{Dbase}_i(x)$.ts, and B.UpTS > $\text{Dbase}_j(x)$.ts. Then Bag(j) contains (B,Accepted) (lemma 4.1).

- (A,Accepted) is removed from Bag(j).
  2) j receives (A,Accepted) and executes the update of A. $\text{Dbase}_j(x).ts <$ A.UpTS, so $\text{Dbase}_j(x)$.ts := A.UpTS, which makes the premise false.

- The value of $\text{Dbase}_i(x)$.ts changes.
  2) i receives (B,Accepted). This case is already treated. □

If a bag contains a request message A with an item x in A's BaseSet with a timestamp higher than x's timestamp in node j, then there is a B such that Bag(j) contains (B,Accepted) and x is in the UpdateSet of B and B's timestamp is the same as x's timestamp in the copy in node j:

Lemma 4.3 $x \in$ A.BaseSet and $\text{Dbase}_j(x).ts <$ A.BaseTS(x) $\implies$
$\exists$B : (B,Accepted) $\in$ Bag(j) and $x \in$ B.UpdateSet and B.UpTS = A.BaseTS(x).

Proof:
The dangerous actions are:

- Such an A is initiated.
  3) Node k initiates request A. Then A.BaseTS(x) := $\text{Dbase}_k(x)$.ts, so $\text{Dbase}_k(x)$.ts > $\text{Dbase}_j(x)$.ts. Then (B,Accepted) $\in$ Bag(j) for a B with the required properties (lemma 4.2).

- (B,Accepted) disappears from Bag(j).
  2) j receives (B,Accepted) and executes the update. Then $\text{Dbase}_j(x)$.ts := B.UpTS (=A.BaseTS(x)). So the premise becomes false. □

Lemma 4.4 A $\in$ PendingSet(i) $\implies \exists$j : A $\in$ Bag(j) or (A,Decision) $\in$ Bag(i).

Proof:
The dangerous actions are:

- A is put into PendingSet(i).
  1) i receives A, votes OK or PASS, and puts A into PendingSet(i). Then i also sends a message with A to some node (places a message with A into some bag).
- A is removed from Bag(j).
  1) j receives A and votes. If j decides, it sends decision messages to all other nodes. If j does not decide, it sends a request message with A to another node.
- (A,Decision) disappears from Bag(i).
  2) i receives (A,Decision). Then A is removed from PendingSet(i). □

If there is still a request message, then there is still an undeferred request message, or a decision message.

Lemma 4.5 A $\in$ Bag(i) $\Longrightarrow$
$\exists$B,j : B $\in$ Bag(j) and not Deferred(B)
or $\exists$C,k : (C,Decision) $\in$ Bag(k).

Proof:
Let A be the request message with the smallest timestamp, and let A $\in$ Bag(i). Suppose A is deferred. Then $\exists$x $\in$ A.BaseSet with A.BaseTS(x) > Dbase$_i$(x).ts, or $\exists$B $\in$ PendingSet(i) with B.UpTS < A.UpTS and A and B conflict. In the first case there is still a decision message in a bag (lemma 4.3). In the second case there is also a decision message (lemma 4.4). □

Theorem 4 If there is still a request message, a pending request or a decision message, an event of the majority consensus algorithm is possible.

Proof:
A decision message can always be received. If there is a pending request, there is also a request message or decison message (lemma 4.4). If there is a request message, then there is an undeferred request message, or a decision message (lemma 4.5). □

4.1.5 Termination

Each event of the majority consensus algorithm initiates, votes on, decides, or executes one request. So events may be considered as associated with one request.

Lemma 5.1 Only finitely many events are associated with each request.

Proof:
A request A is created only once. Each node receives at most once a request message of A, because there is only one cop of a request message, and this is sent only to nodes that have not yet voted on it. And the number of nodes is finite. Also a decision about this request is taken only once (theorem 1), so a finite number of decision messages is created, one for each node, except for the node that made the decision. □

Theorem 5 The majority consensus algorithm terminates if no more new requests are generated.

Proof:
With each request finitely many events are associated (lemma 5.1). An event starts with the reception of a message, or consists of the generation of a new request. If a finite number of requests was generated in a finite time, and each message sent is received in a finite time, then the algorithm terminates in finite time. □

4.1.6 Convergence of database copies

Theorem 6 If no more new requests are generated, all database copies converge to the same value.

Proof:
For each item x consider the accepted request A with $x \in$ A.UpdateSet and with the highest timestamp. If (A,Accepted) arrives at a node i, then $\text{Dbase}_i(x)$.ts < A.UpTS, for otherwise there would be a request B with a higher timestamp that updates x (lemma 1.12). And decision messages are never lost, so each node receives (A,Accepted). So A is executed on x, and no other update will be executed afterwards. So for each item x in each copy the final update is the update of the request with the highest timestamp that has x in its UpdateSet. □

4.1.7 Correct decisions

When a request A is accepted, there is no node that would vote REJ on this request. This means that there is no node that has a higher timestamp for an item in its copy than the timestamp of that item in A.BaseSet.

Theorem 7 When node i accepts a request A, there is no node j with $x \in$ A.UpdateSet and $\text{Dbase}_j(x)$.ts > A.BaseTS(x).

Proof:
Suppose there is such a node j. Then a request B has been executed in j with $x \in$ B.UpdateSet and B.UpTS = $\text{Dbase}_j(x)$.ts (lemma 1.12). So there is a node k that voted OK for A and for B. Suppose k voted first on A. Then A was pending when k voted on B (lemma 1.3), so K does not vote OK on B. Contradiction.
Suppose k voted first on B and then on A. Then B is pending at k or B has already been executed at k when k votes on A. In both cases k does not vote OK on A. Contradiction.
□

4.1.8 Every request is decided

Theorem 8 Every request is decided.

Proof:
If every node receives a request sent to it in finite time, and procedure Majority gives eventually OK or PASS, a decision will be made in finite time, and all nodes will hear of this decision in finite time. □

4.2 Proof for model 2

In model 2 a request may be lost in a node (event 4). Therefore a node with a pending request may repeat sending this request (event 5). A node stores its vote on a request and uses this vote when another copy of this request arrives (event 1). A node now decides "Rejected" if a majority of OK votes is no more possible (procedure Decide). If a request is pending in a node that decides about this request, the pending request is removed (procedure Execute).

4.2.1 More decisions about one request

Here we cannot prove that only one decision about a request is made. There may be more request messages corresponding to a request, so a decision about a request could be made more than once. Instead we prove that all decisions made about a request are the same.

Theorem 1* All decisions made about a request are the same.

Proof:
Suppose there are two different decisions about a request A. Then one VoteSet contained a majority OK and another contained a majority REJ/PASS. So there was a node that voted both OK and REJ or PASS on the same request. But this is not possible. □

4.2.2 Serializability

The proof of serializability in model 2 is almost the same as in model 1. The proofs of some lemmas are changed slightly:
In the proof of Lemma 2.3 is added:

- A is removed from PendingSet(i).
  2).....
  1) i receives A, decides, and A is also pending at i. Then A.DecisionKnown(i) becomes true.

Lemma 2.5 is changed to:

Lemma 2.5* $\exists n$: (OK,i) $\in$ VoteSet(n) and x $\in$ A.BaseSet $\Longrightarrow$
$\text{Dbase}_i(x).\text{ts} \geq \text{A.BaseTS}(x)$.

And the proof is changed to:

- The first (OK,i) is put into a VoteSet of A.
  1) i receives request A for the first time, and votes OK. i has called procedure DetermineVote because timestamps of requests are unique (lemma 2.2). Procedure DetermineVote only produces OK if all base variables .....

In Lemma 2.9 and 2.10 (OK,i) $\in$ A.VoteSet and (OK,i) $\in$ B.VoteSet becomes: $\exists n$: (OK,i) $\in$ A.VoteSet(n) and $\exists n$: (OK,i) $\in$ B.VoteSet(n).
The proofs of Lemma 2.9 and 2.10 are changed in the same way as the proof of Lemma 2.5.
Changes in the proof of Lemma 2.11 are:
If A and B were both accepted, there is a node j with (OK,j) in a VoteSet of A and (OK,j) in a VoteSet of B.
.....

4.2.3 Update sequence on one copy of a data item

Updates on one copy of a data item are executed in a fixed order. The proof is equal to the proof in 4.1.3.

4.2.4 Deadlock Freedom

The proof for model 2 that no deadlock occurs is slightly different from the proof given for model 1.

Lemma 4.1* (A,Accepted) $\in$ Bag(i) $\Longrightarrow$ (A,Accepted) $\in$ Bag(j) or $\forall$x $\in$ A.UpdateSet: Dbase$_j$(x).ts $\geq$A.UpTS.

Proof:
Equal to the proof of Lemma 4.1.

Lemma 4.2* Dbase$_j$(x).ts < Dbase$_i$(x).ts $\Longrightarrow$
$\exists$A : (A,Accepted) $\in$ Bag(j) and x $\in$ A.UpdateSet and A.UpTS = Dbase$_i$(x).ts.

Proof:
Equal to the proof of lemma 4.2.

Lemma 4.3*

x $\in$ A.BaseSet and Dbase$_j$(x).ts < A.BaseTS(x) $\Longrightarrow$
$\exists$B : (B,Accepted) $\in$ Bag(j) and x $\in$ B.UpdateSet and B.UpTS = A.BaseTS(x).

Proof:
Equal to the proof of lemma 4.3.

Lemma 4.4* If A $\in$ Bag(i) and there are no pending messages or decision messages, then A is not deferred.

Proof:
Suppose A is deferred. Then there is an x $\in$ A.BaseSet with A.BaseTS(x) > Dbase$_i$(x).ts. So there is still a decision message (lemma 4.3*) □

Theorem 4* If there is a request message, pending request or decision message, an event of the majority consensus algorithm is possible.

Proof:
A decision message can always be received.
If node i contains a pending request, node i may send a request message with this request.
If there is a request message in Bag(i), and there are no pending messages or decision messages in the system, the request message is undeferred w.r.t. i (lemma 4.4*). □

4.2.5 Termination

For model 1 we proved that only finitely many requests are associated with each request. This is not so in Model 2. Nodes with a pending request A may generate new copies of request A and send them to other nodes. Now we can prove in the same way as in 4.1.5 that with each request **copy** only finitely many events are associated. To prove termination we now need two extra assumptions:

1. In finite time after the creation of a request a decision about this request reaches each node,
2. A node with a pending request A sends only finitely many request messages A in finite time.

For the first assumption it is needed that not all request messages are lost, one request message succeeds in obtaining enough votes, and no decision messages are lost. If these assumptions hold, the majority consensus algorithm of model 2 terminates if no more new requests are created.

4.2.6 Convergence of database copies

The proof is the same as for model 1.

4.2.7 Correct decisions

Theorem 7 is changed to:

Theorem 7* When a request is accepted for the first time, there is no node j with $x \in$ A.UpdateSet and $\text{Dbase}_j(x)$.ts > A.BaseTS(x).

The proof stays the same, but "k voted on A (or B)" should be interpreted as "k voted for the first time on A (or B)".

A consequence of this theorem is that the first VoteSet of a request in which a majority is built up that leads to a decision "Accepted" contains no REJ votes. Later VoteSets may contain REJ votes. Suppose a node does not vote on the copy of the request that produces the first decision receives this decision. Suppose the node executes the update, and then receives another copy of this request. The node will then vote REJ, because the base variables of this request are now obsolete. However, there is only a minority of nodes that may vote REJ, for a majority of the nodes already have a vote OK for this request, and will use this old vote if they receive another copy of this request.

4.2.8 Every request is decided

If a newly initiated request is lost by the first node that receives it, then it is not pending anywhere, and cannot be re-sent. So we must assume that a request is not lost in the first node that receives it, or that the node that initiated the request keeps a copy of the request and sets a timeout.

We must also assume that not all request messages disappear. Then one message will in finite time be decided, and all nodes will hear of this decision in finite time.

Chapter 5

Discussion

We formalized the majority consensus algorithm of Thomas [Thom79], and gave formal proofs for the claims in [Thom79], and some other claims. In most of our proofs we used the basic ideas of the informal proof in [Thom79], although several details had to be added to the algorithm nd (clearly) the invariants were not derived there. Essential assumptions in our proof are the following.
For both models the Majority function must satisfy the following properties:

- If Majority (V1) = OK and Majority(V2) = OK, then $\exists i$: (OK,i) $\in$ V1 and (OK,i) $\in$ V2.
- If Majority(V1) = OK and Majority(V2) = PASS, then $\exists i$: (OK,i) $\in$ V1 and (Vote,i) $\in$ V2 and Vote $\neq$ OK.
- If $\forall$V2 $\supseteq$ V1 : Majority(V2) $\neq$ OK, then Majority(V1) = PASS.

For model 1:

- every message arrives in finite time at its destination.

For model 2:

- no decision messages are lost,
- a request message is not lost before the first node has voted on it, or the creating node keeps a copy and sets a timer,
- a node with a pending request A only sends finitely messages with A in a finite time,
- not all copies of a request message are lost,
- every message that is not lost arrives in finite time at its destination.

We proved that on each copy of a data item a subset of the same set of updates is performed in the same order. If no more new requests are issued, the last update of each copy is the same. So a weak form of mutual consistency is reached.

Thomas [Thom79] claims that serializability, as defined by him, also guarantees internal consistency. But this is not the case. If we look at a whole copy of the database, not at one item, updates are not performed in the same order at all sites. Suppose we have two data items x and y, with predicate x+y $\leq$10, and two updates A and B. A reads x and y with timestamps 1 and 1 and values x=5 and y=5. A gets a timestamp 2 and computes a new value: y := 2. B reads x and y with timestamps 1 and 2 and values x=5 and y=2. B gets timestamp 3 and computes a new value: x := 8. If B was initiated at node i, then A was executed at i before B was initiated there. But there may be another node j that first receives B and then A. Then after execution of B and before execution of A x=8 and y=5 at j.

Luckily a small modification of the algorithm suffices to guarantee internal consistency. Each update that modifies a variable that is present in a predicate, should also write all other variables that are present in the predicate. Then changes in the values of different variables of a predicate do not occur independently any more. In our example node j would first execute update B: x := 8 and y := 2. Upon reception of update A node j would not update x and y any more, because their timestamps are now no more smaller than the timestamp of A.

For model 2 we proved that, if the first decision about a request is Accepted, no node has voted REJ on this copy of the request. So we could omit the condition that all decisions on one request should be the same, and decide Rejected after one REJ vote. But then also another modification to the algorithm is needed. If the first decision about a request is Rejected, then also all other decisions about this request will be Rejected. A request is rejected because it has obsolete base variables, caused by the execution of a conflicting request, or it obtained a majority of PASS votes. But if the first decision is Accepted, a later decision may be Rejected. If a node i executes an update of a request A, and then receives another copy of A, it will vote REJ, because the base variables of A are now obsolete. If another node j first receives a decision message (A,Rejected), j discards its pending message A too early. This pending message should be kept until j receives (A,Accepted), otherwise j could wrongly vote OK on a request B that conflicts with A. Therefore each copy of a request message should have an identification, e.g. a node id plus a sequence number, and a node should only discard a pending message (A,id) if it receives a decision with the same id.

In model 2 a node keeps a copy of each vote it has made. If all decisions about a request A must be the same, these votes must be kept until there are no more request messages of A present. This is difficult to detect. If we only require that all decisions about a request, made before all nodes know the decision, are the same, the votes may be discarded when all nodes know the decision. This could be done by sending the decision message over a logical ring instead of broadcasting it.

In the modification we proposed, where also for model 2 one REJ leads to a decision Rejected, votes need not be kept at all. Nodes may determine their vote anew every time a request arrives. A node will never first vote REJ and then something else. But it is possible that a node first votes PASS, and later, when the conflicting request is rejected and removed from the PendingSet, votes OK. So then it is possible that the first decision is Rejected because there was a majority of PASS votes, and a later decision is Accepted. If one node votes REJ, the decision cannot become Accepted any more.

The algorithm is resilient to a number of error types. It is sufficient that one request message succeeds in acquiring a majority voteset. So a succession of pairs of nodes must be able to communicate. At the moment that two nodes communicate over a link, all other nodes and links may be down.