Application of Event B to Global Causal Ordering for Fault Tolerant Transactions

Divakar Yadav* and Michael Butler**

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ ,U.K
{dsy04r,mjb}@ecs.soton.ac.uk

Abstract. System availability is improved by replication of data objects in a distributed database system. It is advantageous to replicate data objects when transaction workload is predominantly read only. However, during updates, the complexity of keeping replica identical arises due to failures. A number of approaches has been proposed to make systems fault tolerant through exchange of messages. Logical clocks provide the framework to realize global causal ordering on messages. The algorithms ensuring globally ordered delivery of messages may be coupled with the provisions to provide fault tolerance in event of failures. The B Method provides state based formal notations for writing specification of software systems. Event B provides a formal approach to development of such complex system. In this paper we present a part of ongoing work in this area. The specification for global ordering of messages is presented as B Machine. The global ordering of messages may be achieved by implementing Vector Clocks. The same approach may be extended to the formal development of a fault tolerant distributed data replication system.

1 Introduction

A distributed system is a collection of autonomous computer system that cooperate with each other for successful completion of a distributed computation. A distributed computation may require access to resources located at participating sites. A typical database transaction contains a sequence of database operations. This sequence of database operations is considered as an atomic action. In order to maintain the consistency of the database either all of the operations need be done or none at all. A distributed transaction may spawn several sites reading or updating data objects. The purpose of replication of data objects at different sites is to increase the availability of systems which in turn speeds up query processing. Replication of data is advantageous when the transaction work load is

* Divakar Yadav is a Commonwealth Scholar supported by the Commonwealth Scholarship Commission in the United Kingdom.

** Michael Butler's contribution is part of the IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

predominantly read only. However in the case of updates, it is necessary to keep the replicas identical, failing to do that may lead the database in to an inconsistent state. A distributed transaction consists of communicating transaction components at participating sites. A commit or abort decision of a distributed transaction is based on the decision of components of the transaction running at participating sites. A commit is an unconditional guarantee that update to a database are permanent. To maintain consistency of a database it is necessary that a transaction commit at each participating site or at none of sites.

There exist several approaches to ensure global atomicity. Gray addressed the issue of ensuring global atomicity despite failures in [10]. The two phase commit protocol provides fault tolerance to distributed transactions despite failures. Transaction failure, site failure and network partition are causes of failures. An increasing number of components in a distributed system imply a higher probability of component failure during execution of distributed transactions. The *two phase commit* protocol ensures global atomicity through exchange of messages among the participating sites and coordinating site. The drawback of this protocol is that it is blocking because in case of failure of the coordinator site, participants wait for its recovery. The variants of this protocol were proposed to improve the performance of this protocol in [14]. The *presumed commit* protocol is optimized to handle general update transactions while *presumed abort* optimizes partial read-only transactions. Levi and others presented an *optimistic two phase nonblocking commit* protocol [12] in which locks acquired on data object at a site are released when the site is ready to commit. In case of abort of distributed transactions, a compensating transaction is executed at that site to undo the updates. A *three phase commit* protocol [18] is a nonblocking commit protocol where failures are restricted to site failures only. All of these protocols assume that mechanisms such as maintaining the database log and local recovery are present locally at each site. There are a number of communication paradigm in which commit protocols are implemented. In *centralized two phase commit* protocol no messages are exchanged among participating sites and messages are exchanged only between between the coordinator site and cohorts. In the *nested two phase commit* protocol cohorts may exchange messages among themselves. A *distributed two phase commit* eliminates the second phase as the coordinator and cohorts exchange messages through broadcasting. Every site may reach the decision to abort/commit by means of vote-abort or vote-commit message.

In a distributed system neither a global common clock nor shared memory exist. In the absence of a global clock and shared memory, an up to date knowledge of a system is not known to any process. In these systems the processes communicate through exchange of messages. These messages are delivered after arbitrary time delays. This asynchronous distributed system model may span large geographical areas. A system may be designed as a fault tolerant system either by masking failures or by following a defined sequence of steps in the process of recovery after failure. The transaction updates are visible to concurrent transactions only if it successfully commits. The update caused by a failed transaction are not made visible to other concurrent transactions. This may be

achieved if the system follows well defined steps on recovery from failures. The distributed two phase commit protocol requires broadcasting of messages among the sites. Global causal ordering of messages is used to achieve error recovery using vector clocks. A global ordering on messages may be defined by employing logical clocks. The algorithms ensuring globally ordered delivery of messages may be coupled with provisions of recovery from failures and fault tolerance in the event of process failures or network partitions. This also helps debugging distributed computations since they provides the mechanism to identify the order in which they occurred despite process failures or network partition. A good description of work on logical clocks and its application in solving varying problems of distributed computation may be found in [20], [9], [15]. There has been lot of work in development of fault tolerant protocols for distributed system, very few have been subjected to formal verification. It is desirable that model of distributed system be precise, reasonably compact and one expects that all the aspects of system must be considered in proofs because it leads to better design.

The B Method is proof based method for the rigorous development of systems. In this paper we outline the formal development of a system for global causal ordering of messages using vector clocks. This is a part of on-going work on the formal development of a fault tolerant distributed data replication system.

2 The B Method and Event B

Formal methods provides a systematic approach to the development of complex systems. Formal methods use mathematical notations to describe and reason about systems. B Method [1] a model oriented formal notation developed by Abrial. The B Method provides a state based formal notation based on set theory for writing abstract models of systems. A system may be defined as an abstract machine . Abstract machine contains *sets,variables,invariants, initialization* and a set of *operations* defined on variables. The *set* clause contains user defined sets that can be used in rest of machine. The variables describe the state of machine. The *invariants* are first order predicates and these invariants are to be preserved while updating the variables through the operations. The operations can have input and output parameters. Operation of machines are defined through generalized substitution. The B method allows specifications of abstract model to be written and support the stepwise refinement. At each refinement step we get more concrete specification of system. The B Method requires the discharge of proof obligations for consistency checking and refinement checking. Consistency checking involves showing that a machine preserves invariants when operations are invoked. Refinement checking involves showing that specifications at each refinement step are valid. The B Tools (Atelier B, Click'n'Prove,B-Toolkit) also provides an automatic and interactive prover. Typically the majority of proof obligations are proved by automatic prover, however some of the complex proof obligations needs to be proved interactively.

Though a significant work has been done in the area of message passing systems, logical clocks, recovery, checkpointing and fault tolerance yet application of

proof based formal method to this work is rare to our knowledge. B can be used to provide formalization of protocols and algorithms of distributed system. Event B was introduced for modeling of distributed system. In Event B [2] operations are referred to as events which occurs spontaneously rather then being invoked. These events are guarded by predicates and these guards may be strengthened at each refinement steps. Applications of the B method to distributed system may be found in [6], [7], [17].

3 Global Ordering of Messages

A distributed program is composed of finite set of processes. The processes communicate with each other through exchange of messages. A class of problems relating such message passing system may be solved by defining global ordering on the messages. The messages are delivered to recipient process following their global order. The logical clocks such as Lamport Clock [11], Vector Clock [8] provides the mechanism to ensure globally ordered delivery of messages. A critical review of logical clocks can be found in [3], [16].

The execution of a process is characterized by sequences of events. These events can be either *internal events* or *message events*. An internal event represents a computation milestone achieved in a process, whereas message events signifies exchange of messages among the processes. *Message Sent* and *Message Receive* are message events respectively occurring at a process sending a message and receiving a message. The causal ordering of messages was proposed in [4]. Protocols proposed in [5], [19] use logical clocks to maintain the causal order of messages. A *happened before* relation defines the causal relationships between the events [11]. The happened before relation ($\rightarrow$) is defined as $a \rightarrow b$ where event a happened before b. The events a and b are either of following,

- a,b are internal events of a same process such that a , $b \in P_i$ and a happened before b.
- a,b are message events at different processes such that $a \in P_i$, $b \in P_j$, where a is *Message Send* event occurring at process P_i and b is *Message Receive* event occurred at P_j while sending a message m from process P_i to P_j.

Later we lift the *happened before* relation ($\rightarrow$) to define a global ordering on messages. The happened before relation is transitive i.e. if event a happened before b and b happened before c then a is said to happened before c.

$$a \rightarrow b \bigwedge b \rightarrow c \Rightarrow a \rightarrow c$$

We can further define the causally related and concurrent events using this relation. The two events a and b are causally related if either $a \rightarrow b$ or $b \rightarrow a$. Event a causally affects b if $a \rightarrow b$. The two events a and b are concurrent ($a \parallel b$) if $a \nrightarrow b$ and $b \nrightarrow a$. Therefore for any two events a and b there exist three possibilities i.e. either $a \rightarrow b$ or $b \rightarrow a$ or $a \parallel b$. The global ordering of messages deals with the notion of maintaining the same causal relationship that holds on *Message Send* and *Message Receive* relationship in their processes. In a broadcast network it is required that any recipient of a message must receive

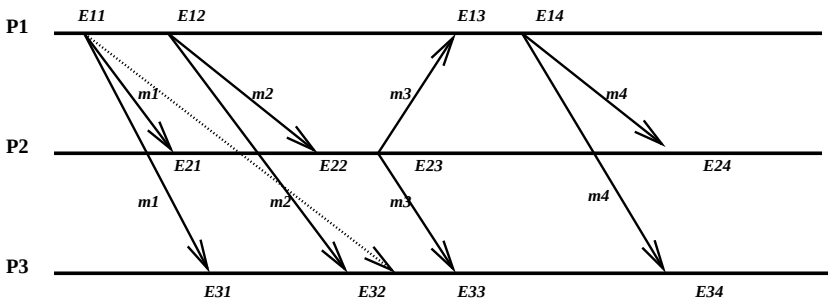

Fig. 1. Message Ordering

all the messages which are ordered before this message. As shown in figure 1, process *P1* first broadcasts message *M1*, then *P1* broadcasts the message *M2*. Process *P2* broadcasts message *M3* after receiving message *M1* and *M2* from process *P1*. Process *P1* again broadcasts message *M4* after receiving *M3* from *P2*. The global ordering among the messages may be defined on set of messages. The message *M1* is said to be ordered before *M2* as their exists causal relationship among the corresponding *Message Send* events: *Message Send* event of message *M1* happened before *Message Send* event of message *M2* in process *P1*. Therefore all recipients of message *M1* and *M2* must receive these messages in the order they were sent i.e. process *P3* must receive *M1* before receiving *M2* as shown in figure. If the message *M1* is delayed (shown as dotted lines) and delivered after delivery of message *M2*, it represents a violation of the global ordering. For any two messages M_i and M_j, message M_i is ordered before M_j ($M_i \rightarrow M_j$) if the *Message Send* event of M_i happened before the *Message Send* event of M_j and the sender of both messages is the same. If the sender process of these messages is different then message M_i will be ordered before M_j if the *Message Receive* event of M_i happened before the *Message Sent* event of M_j in the process sending message M_j. The global order of messages may be defined as follows. Suppose messages *M1* and *M2* are sent by processes *P1* and *P2* respectively. Message *M1* is ordered before *M2* ($M1 \rightarrow M2$) iff either of following holds.

- *Send(M1)* $\rightarrow$ *Send(M2)*, where sender(*M1*)=sender(*M2*) and *M1* is sent before *M2*.
- *Receive(M1)* $\rightarrow$ *Send(M2)*, where sender(*M1*) $\neq$ sender(*M2*) and *M1* is received by sender of *M2* before *M2* is sent.

Send(M) and *Receive(M)* are events representing sending and receipt of message *M* respectively. The two messages *M1* and *M2* are defined as parallel messages ($M1 \parallel M2$) when no partial ordering exist among them i.e. $\neg (M1 \rightarrow M2) \wedge \neg (M2 \rightarrow M1)$ holds. These messages may be delivered to a recipient process in any order. As shown in figure 1, $(M1 \rightarrow M2)$ holds as $E11 \rightarrow E12$, $(M2 \rightarrow M3)$ holds as $E22 \rightarrow E23$, $(M3 \rightarrow M4)$ holds as $E13 \rightarrow E14$. Due to transitivity condition

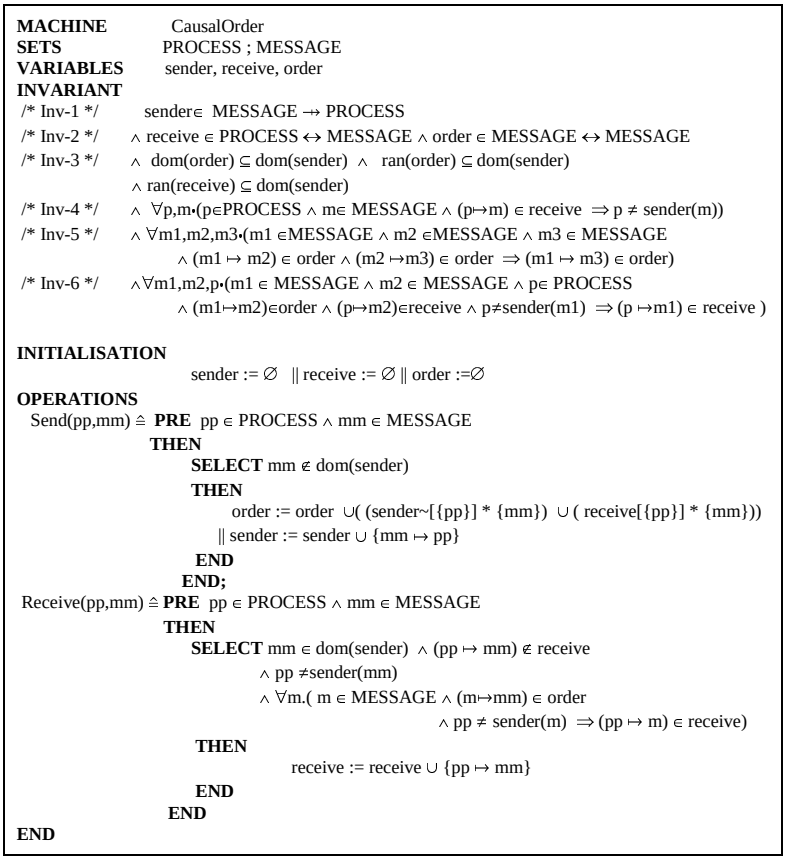

Fig. 2. Abstract Model of Causal Order in B

(M1 → M3) ,*(M1 → M4)* and *(M2 → M4)* also holds. The abstract model of causal order of messages is presented in figure 2 as a B Model. Knowledge of B syntax is assumed. The brief description of this machine is given below.

- PROCESS and MESSAGE are defined as sets. The *sender* is a partial function from MESSAGE to PROCESS. The *receive* is a relation between PROCESS and MESSAGE ((p↦m) ∈ receive indicates that process p has received message m). The *order* is a relation between MESSAGE and MESSAGE. (Shown as *Inv-1* and *Inv-2* in the invariant clause of the model)
- A *sent* message is not received by its sender and all received message must be messages whose *Message Send* event is recorded. Similarly, ordering of messages can be defined only on those messages whose *Message Send* event is recorded. (Shown as *Inv-3,Inv-4*)
- The invariant contains a predicate which requires that transitivity property on messages should be maintained.(Shown as *Inv-5*)

- For any message whose *Message Receive* event happened at a process, that process must have received all the messages *ordered before* that message. (Shown as *Inv-6*)
- *Send* and *Receive* are events of messages defined as operations. These events are guarded by predicates. In the event of sending a message *mm* by process *pp*, all messages sent and received by process *pp* are *ordered before* the message *mm*.
- In the event of receipt of a message *mm* by a process *pp*, it must ensured that all messages *ordered before mm* has been received by process *pp*. This condition is satisfied by a predicate in the guard of operation *Receive(pp,mm)*.

4 Vector Clocks

Logical clocks are viable solution to causally order various events and to ensure globally ordered delivery of messages to processes [5], [19]. Scalar and Vector Clocks are widely referred to as logical clocks. Scalar clocks, introduced by Lamport in [11], uses an integer value to timestamp an event whereas vector clocks, introduced in [8], [13], uses a vector of integers to timestamp an event. A vector clock may be defined as a function which assign a vector of integer to an event called timestamp. For every process P_i, there exist a clock VT_{Pi} which maps an event to a vector of integer. Suppose set E_{Pi} defines the sequence of events produced by process P_i. The clock function may be defined as $VT_{Pi} : E_{Pi} \rightarrow \mathbb{V}$, where $\mathbb{V}$ is a set of vectors. The clock VT_{Pi} assigns a time stamp VT_{Pi} (e_{ij}) to event e_{ij} where $e_{ij} \in E_{Pi}$.

In a system of vector clocks, every process maintains a vector of size N where N is the total number of processes in the system. Process P_i maintains a vector clock VT_{Pi} where VT_{Pi}(i) is the local logical time at P_i while VT_{Pi}(j) represents the process P_i's latest knowledge of the time at process P_j. More precisely VT_{Pi}(j) (i$\neq$j) represents the time of occurrence of an event at process P_j when the most recent message was sent from P_j to P_i directly or indirectly. In this model, vector clocks are used to timestamp *message send* and *message receive* events only. The following rules are used to update the vector clock of process and timestamping a message in the event of *message sent* and *message receive*.

- While sending a message M from process P_i to P_j, sender process P_i updates its own time(i^{th} entry of vector) by updating VT_{Pi}(i) as VT_{Pi}(i) := VT_{Pi}(i) + 1. The message time stamp VT_M of message M is generated as VT_M(k) := VT_{Pi}(k), $\forall$ k $\in$ (1..N), where N is number of processes in system. Since a process P_i increments its own value only at the time of sending a message, VT_{Pi}(i) indicates number of messages sent out by process P_i.
- The recipient process P_j delays the delivery of message M until following conditions are satisfied.
  - VT_{Pj}(i)= VT_M(i) - 1
  - VT_{Pj}(k)$\geq$ VT_M(k), $\forall$k $\in$ (1..N) $\wedge$ (k $\neq$ i).

  The first condition ensures that process P_j has received all but one message sent by process P_i. The second condition ensures that process P_j has

received all messages received by sender P_i before sending the message M. These conditions ensures global ordering on messages.
- The recipient process Pj updates its vector clock VT_{Pj} at *message receive* event of message M as $VT_{Pj}(\mathrm{k}) := \max(VT_{Pj}(\mathrm{k}), VT_M(\mathrm{k}))$. Therefore in vector clock of process P_j, $VT_{Pj}(\mathrm{i})$ indicates the number of messages delivered to process P_j sent by process P_i.

Part of the B refinement of the abstract model of causal order of messages through vector clocks is shown in the figure 3 and figure 4. Figure 3 contains invariants, variables and initialization clause. The operations are shown in figure 4. A brief description of refinement steps are given below.

- Vector time of a process is represented by a variable *VTP*. The timestamp of a message is represented by a variable *VTM*. *VTP* and *VTM* are defined as functions as shown in invariant *Inv-7*, and *Inv-8*. Other conditions required for vector clock implementations are shown in invariants *Inv-10*. Vector timestamp of each process is initialized with value '0'. Initialization of variable VTP and VTM is shown in the initialization clause.
- A new variable *buffer* is introduced in refinement. The *buffer* is a relation between PROCESS and MESSAGE (*Inv-9*). The messages arriving at a process are initially buffered. The buffered messages are received by a process on satisfying the conditions as defined in vector clocks.
- The *Send*, *Arrive* and *Receive* events of a message at a process are shown as operations in figure-4. At the time of sending a message *mm*, process *pp* increments its own clock value VTP(pp)(pp) by one. The VTP(pp)(pp) represents the number of messages sent by process *pp*. The modified vector timestamp of process is assigned to message *mm* giving vector timestamp of message *mm*.
- The messages may arrive at a process in any order but their *Message Receive* event occurs at that process only if it has received all but one message

VARIABLES sender, receive, order, buffer, VTP, VTM
INVARIANT
/*Inv-7*/ VTP ∈ PROCESS → (PROCESS → ℕ)
/*Inv-8*/ ∧ VTM ∈ MESSAGE ⇸(PROCESS → ℕ)
/*Inv-9*/ ∧ buffer ∈ PROCESS ↔ MESSAGE ∧ ran(buffer) ⊆ dom (sender)
/*Inv-10*/ ∧ ∀m1,m2,p·(m1∈ MESSAGE ∧ m2∈ MESSAGE ∧ p ∈ PROCESS
∧ (m1 ↦ m2) ∈ order ⇒ VTM (m1)(p) ≤ VTM(m2)(p))
INITIALISATION

VTP := PROCESS * { PROCESS * {0}}
||VTM := ∅
|| sender :=∅ || buffer := ∅ || receive :=∅ || order := ∅

Fig. 3. Refinement using Vector Clocks : Invariants

OPERATIONS

Send(pp,mm) ≙ **SELECT** mm ∉dom(sender)
THEN
LET nVTP
BE nVTP = VTP(pp) ⩤ { pp ↦ VTP(pp)(pp)+1}
IN VTM(mm) := nVTP || VTP(pp) := nVTP END
|| sender := sender ∪ {mm ↦ pp}
END ;

Arrive(pp,mm) ≙ **SELECT** mm ∈ dom(sender) ∧ (pp ↦ mm) ∉ buffer
∧ (pp ↦ mm) ∉ receive ∧ pp ≠ sender(mm)
THEN
buffer := buffer ∪ {pp ↦ mm}
END ;

Receive(pp,mm) ≙ **SELECT** (pp ↦ mm) ∈ buffer ∧ (pp ↦ mm) ∉ receive ∧ pp ≠ sender(mm)
∧ ∀p.(p ∈ PROCESS ∧ p ≠ sender(mm) ⇒ VTP(pp)(p) ≥ VTM(mm)(p))
∧ VTP(pp)(sender(mm)) = VTM (mm)(sender(mm))-1
THEN
receive := receive ∪ {pp ↦ mm} || buffer := buffer - {pp ↦mm}
|| VTP(pp) := VTP(pp) ⩤ ({q | q ∈ PROCESS∧VTP(pp)(q) < VTM(mm)(q)} ◁ VTM(mm))
END
END

Fig. 4. Refinement using Vector Clocks : Operations

from the sender of that message. Vector timestamp of recipient process and message are also compared to ensure that all messages received by sender of message before sending it, are also received at the recipient process. These conditions are included as a guard in *Receive* operation. It can be noticed that the guard involving the variable *order* in the abstract model is replaced by a guard involving comparison of vector timestamp of message and process in the refinement.[1]

– The replacement of the guard involving variable *order* in abstract model with guards involving comparison of vector timestamp in refinement generates proof obligations. These proof obligations can be discharged interactively using a B Prover.

5 Conclusions

The abstract model in figure-2 of causal order provides a clear specification of causal ordering property on messages. We are currently working on formal development of fault tolerant distributed data replication system and we are finding that the abstract model of causal ordering is much easier to work with

[1] (f ⩤ g) represents function f overridden by g. (s ◁ f) represents function f is domain restricted by s.

than the vector clock model when proving the correctness of recovery mechanism. In this paper we outlined how the abstract causal order is in turn correctly implemented by the vector clock system. Our experience shows that abstraction and refinement are valuable techniques for modeling and verification of complex systems.

References

1. J R Abrial. The B Book : Assigning Programs to Meaning, Cambridge University Press,1996.
2. J R Abrial. Extending B without changing it.(For Distributed System).Proc. of 1st Conf. on B Method,pp 169-191,1996
3. R Baldoni, M Raynal. Fundamental of Distributed Computing : A Practical tour of Vector Clock Systems. IEEE Distributed System online, Vol 3, No2 , Sept 2002.
4. K P Birman, T A Joseph. Reliable communication in the presence of failures. ACM Transaction on Computer System,pp47-76 Vol5,No.1,1987.
5. K P Birman, A Schiper, P Stephenson. Lighweight causal and atomic group multicast. ACM Transaction on Computer System,Vol9,No 3,pp 272-314, 1991.
6. M Butler. An Approach to Design of Distributed Systems with B AMN. Technical Report,Electronics and Computer Science,Univerity of Southampton,1996
7. M Butler, M Walden. Distributed System Development in B. Proc. of Ist Conf. in B Method, Nantes,pp155-168,1996
8. C Fidge. Logical Time in Distributed Computing System. Computer, Vol 24,no 8,pp. 28-33,1991.
9. H Garcia-Molina, J D Ullman , J Widem. Database System : A complete Book. Pearson Education, 2002.
10. J N Gray. Notes of Database Operating System. in Operating System : An Advance Course, Springer Verlag,New york,pp 393-481,1979.
11. L Lamport. Time,Clocks and Ordering of events in a Distributed Sytem. Communication of ACM,Vol21,No.7,pp558-564,July 78
12. E Levy, H F Korth, A Silberschatz. An optimistic commit protocol for distributed transaction management.Proceedings of ACM SIGMOD,1991
13. F Mattern. Virtual Time and Global states of Distributed Systems. Parallel and Distributed Algorithm, Elsevier Science,North Holland,pp215-226,1987.
14. C Mohan, B Lindsey, R Obermark. Transaction management in R* Distributed Database. ACM TODS,11(4):378-396,1986.
15. M T Ozsu, P Valduriez. Distribted Database Systems. Prentice Hall,1999
16. M Raynal, M Singhal. Logical Time : Capturing casuality in Distributed System. IEEE Computer 29(2) : 49-56,IEEE,Feb 1996.
17. A Rezazadeh, M Butler. Some Guidelines for formal developement of web based application in B Method. Proc. of 4th Intl. Conf. of B and Z users,Guildford,LNCS,Springer,pp 472-491,April 2005.
18. D Skeen. Non Blocking Commit Protocol. ACM SIGMOD,Intl. Conf. on Management of Data,pp133-142,1981
19. A Schiper, J Eggli, A Sandoz. A new algorithm to introduce causal ordering. Proc. Intl. Workshop on Distributed Algorithms,Springer-Verlag,NewYork,pp 219-232,1989.
20. M Singhal, N Shivratri. Advanced Concept in Operating System. Tata McGraw Hill ,Delhi, 2001