Formal Development of Fault Tolerant Transactions for a Replicated Database using Ordered Broadcasts

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. Data replication across several sites improves fault tolerance as available sites can take over the load of failed sites. Data is usually accessed within a transactional framework. However, updating replicated data within a transactional framework is a complex affair due to failures and conflicting transactions. Group communication primitives have been proposed to support transactions in a asynchronous distributed system. In this paper we outline how a refinement based approach with Event B can be used for the development of a reliable replicated database system that ensure atomic commitment of update transactions using group communication primitives.

1 Introduction

A replicated database system can be defined as a distributed system where copies of the database are kept across several sites. Data access in a replicated database can be done within a transactional framework. It is advantageous to replicate the data if the transaction workload is predominantly read only. However, during updates, keeping the replicas in a consistent state arises due to race conditions among conflicting update transactions. A distributed transaction may span several sites reading or updating data objects. A typical distributed transaction contains a sequence of database operations which must be processed at all of the participating sites or none of the sites to maintain the integrity of the database. The strong consistency criterion in the replicated database requires that the database remains in a consistent state despite transaction failures. The possible causes of the transaction failures include bad data input, time outs, temporary unavailability of data at a site and deadlocks.

No common global clock or shared memory exist in a distributed system. The sites communicate by the exchange of messages which are delivered to them after arbitrary time delays. In such systems up-to-date knowledge of the system is not known to any process or site. This problem can dealt by relying on group communication primitives that provide higher guarantees on the delivery of messages. Group communication has also been investigated in Isis [5], Totem [14] and Trans [12]. The protocols in these systems use varying broadcast primitives and address group maintenance, fault tolerance and consistency services. The transaction semantics in the management of replicated data is also considered in [3, 16]. In addition to providing fault tolerance, one of the important issues to be addressed in the design of replica control protocols is consistency. The *One Copy Equivalence* [4] criteria

* 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).

requires that a replicated database is in a mutually consistent state only if all copies of data objects *logically* have the same identical value.

Distributed algorithms can be deceptive, may have complex execution paths and may allow unanticipated behavior. Rigorous reasoning about such algorithms is required to ensure that an algorithm achieves what it is supposed to do [11]. Group communication services have been studied as a basic building block for many fault tolerant distributed services, however the application of formal methods providing clear specifications and proofs of correctness is rare [6]. Some of the important work on the application of formal methods to group communication services in order to verify the properties of algorithm are given in [7, 17]. The work reported in [7] uses I/O automata for the specifications and proves properties about all trace behavior of the automation. In [17] formal results are provided that defines whether or not a totally ordered protocol provides a causal order. They provide a proof of correctness by doing proofs by hand. Instead, our approach of specifying the system and verification is based on the technique of abstraction and refinement. This technique is supported by the Event B [13], an event driven approach used together with B Method [1]. This formal approach carries a step-wise development from initial abstract specifications to a detailed design of a system in the refinement steps. Through the refinement proofs we verify that design of detailed system conforms to the abstract specifications. The refinement approach of Event B has also been used for the formal development of fault tolerant communication systems [9]. We have used the Click'n'Prove [2] B tool for proof obligation generation and for discharging proof obligations.

The remainder of this paper is organized as follows: Section 2 outline the system model, Section 3 describes group communication primitives and its application to replicated database, Section 4 outline the formal development of a system of total order broadcast and Section 5 concludes the paper.

2 System Model

Our system model consist of a sets of sites and data objects. Users interact with the database by *starting transactions*. We consider the case of full replication and assume all data objects are updateable. The *Read Anywhere Write Everywhere* [4, 15] replica control mechanism is considered for updating replicas. A transaction is considered as a sequence of read/write operations executed atomically, i.e., a transaction will either *commit* or *abort* the effect of all database operations. The following types of transactions are considered for this model of replicated database.

- *Read-Only Transactions* : These transaction are submitted locally to the site and *commit* after reading the requested data object locally.
- *Update Transactions* : These transactions update the requested data objects. The effect of update transactions are global, thus when committed, all replicas of data objects maintained at all sites must be updated. In case of abort, none of the sites update the data object.

Let the sequence of read/write operations issued by the transaction T_i be defined by a set of objects $objectset[T_i]$ where $objectset[T_i] \neq \varnothing$. Let the set $writeset[T_i]$ represents the set of object to be *updated* such that $writeset[T_i] \subseteq objectset[T_i]$.

A transaction T_i is a read-only transaction if $writeset[T_i] = \varnothing$. Similarly a transaction T_i is a update transaction if its $writeset[T_i] \neq \varnothing$. Two update transactions T_i and T_j are in *conflict* if the sequence of operations issued by T_i and T_j are defined on set of object $objectset[T_i]$ and $objectset[T_j]$ respectively and $objectset[T_i] \cap objectset[T_j] \neq \varnothing$.

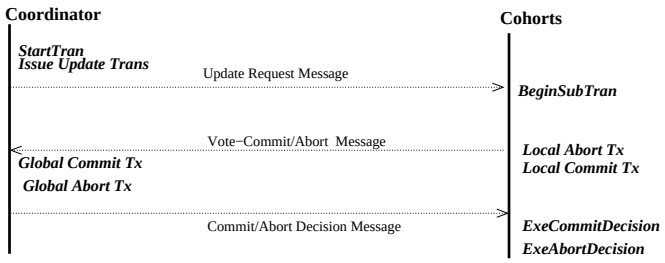

Fig. 1. Events of Update Transaction

In order to meet the strong consistency requirement where each transaction reads the correct value of a replica, *conflicting* transactions need to be executed in isolation. In our model, we ensure this property by not *issuing* a transaction at a site if there is a conflicting transaction that is *active* at that site. In our model the transactions are executed as follows.

- A read-only transaction T_i is executed locally at the initiating site of T_i(also called the coordinator site of T_i) by acquiring locks on the data object defined by $objectset[T_i]$.
- An update transaction T_i is executed by broadcasting its $objectset[T_i]$ to the participating sites. On delivery, a participating site S_j initiates a subtransaction T_{ij} by acquiring locks on $objectset[T_i]$. If the objects are currently locked by another transaction, the T_{ij} is blocked.
- The coordinator site of T_i waits for the vote commit/abort messages from all participating site. A global commit/abort message is broadcasted by coordinator site of T_i only if it receives all local commit message from all participating sites or at-least one vote-abort message from participating sites.

The commit or abort decision of global transaction T_i is taken at the coordinator site within the framework of a two phase commit protocol as shown in Fig. 1 as follows. A global transaction T_i *commits* if *all* T_{ij} *commit* at S_j. The global transaction T_i *aborts* if *some* T_{ij} *aborts* at S_j.

In [20] we have presented a formal refinement based approach using Event B to model and analyze distributed transaction. In our abstract model, an update transaction modifies the abstract one copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of atomic update of a one copy database. Through the refinement proofs, we verify that the design of the replicated database confirms to the one copy database abstraction despite transaction failures at a site. In this model we assume that the sites communicate by a reliable broadcast which eventually deliver messages without any ordering guarantees. Therefore, the conflicting operations of update transactions originating from different sites may arrive at different sites in the different order. This may lead to the deadlocks among the conflicting transactions which results in unnecessary abort of various transactions. The abort of these conflicting update transactions may be avoided if a reliable broadcast also provide higher ordering guarantees on the message delivery. We are currently investigating and formalizing the group communication primitive with reference to the the update transactions.

3 Ordering Properties

In our model of replicated databases [20] we assume that the sites communicate by exchange of messages using a reliable broadcast. A reliable broadcast [8] eventually deliver the messages to all participating sites and satisfies following properties.

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement*: All correct processes delivers a same set of message, i.e. if a process delivers a message m then all correct processes eventually delivers m.
- *Integrity* : No spurious messages are ever delivered, i.e., for any message m, every correct process delivers m at most once and only if m was previously broadcast by $sender(m)$.

A reliable broadcast imposes no restriction on the order in which messages are delivered to the processes. This may lead to the blocking of conflicting transactions and the sites may abort one or more of the conflicting transaction by timeouts. For example, consider two conflicting update transactions T_i and T_j initiated at site S_i and S_j respectively. Both of the transactions may be blocked in the following scenario :

- S_i starts transaction T_i and acquire locks on $objectset[T_i]$ at site S_i. Site S_i broadcast update messages of T_i to participating sites. Similarly, another site S_j starts a transaction T_j , acquires locks on $objectset[T_j]$ at site S_j and broadcast update messages of T_j to participating sites.
- The site S_i delivers update message of T_j from S_j and S_j delivers update message of T_i from S_i. The T_j is blocked at S_i as S_i waits for vote-commit from S_j for T_i. Similarly, T_i is blocked at S_j waiting for vote-commit from S_i for T_j

In order to recover from the above scenario where two conflicting transaction are blocked, either or both transactions may be aborted by the sites. This problem can greatly be simplified by assuming a stronger notion of reliable broadcast that provide higher order guarantees on message delivery. Various definitions of ordering properties have been discussed in [8]. A reliable broadcast can be used to deliver messages to the processes following a *FIFO Order*, *Local Order*, *Causal Order* or a *Total Order*. An informal specifications of these ordering properties are given below.

- *FIFO Order* : If a particular process broadcasts a message *M1* before it broadcasts a message *M2*, then each recipient process delivers *M1* before *M2*.
- *Local Order*: If a process delivers *M1* before broadcasting the message *M2*, then each recipient process delivers *M1* before *M2*.
- *Causal Order* : If the broadcast of a message *M1 causally precedes* the broadcast of a message *M2*, then no correct process delivers *M2* unless it has previously delivered *M1*.
- *Total Order* : If two process *P1* and *P2* both deliver the messages *M1* and *M2* then *P1* delivers *M1* before *M2* if and only if *P2* also delivers *M1* before *M2*.

A *Causal Order Broadcast* is a reliable broadcast that satisfies the *causal order* requirement. The notion of causality is based on *causal precedence relation* ($\rightarrow$) defined by the Lamport [10]. It is also shown in the [8] that a *causal order* combines the properties of both FIFO and Local order. The FIFO Order and Local Order is shown in the Fig. 2. The dotted lines represents the delivery of message violating the respective order. Similarly, a *Total Order Broadcast*[1] is a reliable broadcast that

[1] The *Total Order Broadcast* is also known as Atomic Broadcast. Both of the terms are used interchangeably. However we prefer the former as the term *atomic* suggests the *agreement* property rather than *total order*.

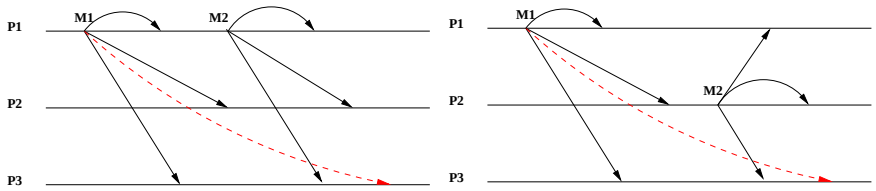

Fig. 2. [a]. FIFO order [b]. Local Order

satisfies the *total order* requirement. The *Agreement* and *Total Order* requirements of Total Order Broadcast imply that all correct processes eventually deliver the same *sequence* of messages [8]. As shown in the Fig. 3[a] the messages are delivered conforming to both causal and a total order. However, as shown in Fig. 3[b] the delivery order respects a total order but violates causal order. It can be noticed that the causality among the broadcast of message *M2* and *M3* is not preserved for delivery.

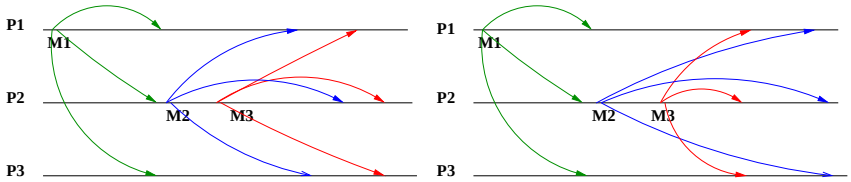

Fig. 3. [a]Total Order and a Causal Order [b]Total Order but not a Causal Order

Processing Transactions on Ordered Broadcast : In a replicated database that uses a reliable broadcast without ordering guarantees, the conflicting operations of the transactions mays arrive at different sites in different orders. This may lead to unnecessary aborts of the transaction due to blocking. The abortion of the conflicting transaction can be avoided by using a *total order broadcast* which delivers and execute the conflicting operations at all sites in the same order. Similarly, a *causal order broadcast* captures conflict as causality and transactions executing conflicting operations are executed at all sites in the same order. In [19], we used a refinement approach with Event B for formal development of broadcast system which deliver messages satisfying various ordering properties. In the abstract model we outlined how a causal order on the message can be constructed and in the refinement we show how it can correctly be implemented by vector clocks. In [18] we present a model of causal order broadcast which does not deliver messages to the sender. In a separate development in [19] we also outline the construction of abstract total order on messages and its implementation using sequencer numbers in the refinements. Lastly, we also formally develop a system where message are delivered following both a total and a causal order[2].

4 Total Order Broadcast

In this section we outline the incremental development of a system of total order broadcast. The operations of an update transaction are communicated by the co-

[2] A reliable broadcast that satisfies both causal and total order is also called *Causal Atomic Broadcast*.

ordinator site to the participating sites by broadcast of an update message. The abortion of the conflicting transaction can be avoided by using an underlying system of *total order broadcast* which delivers and execute the conflicting operations at all sites in the same order.

Our system model consists of a set of sites communicating by a reliable broadcast. The key issues with respect to a system of total order broadcast are ; how to build an order on messages? and what information is necessary for defining an abstract total order? Our approach of building a total order is based on the notion of *sequencer*, where a designated site called *sequencer* site is responsible for building a total order. A *sequencer* site may also take the role of a *sender* and *destination* in addition to the role of *sequencer*. The protocol consists of first broadcasting an *update message* m to all destinations including the sequencer. Upon receiving m, sequencer assigns it a sequence number and broadcast its sequence number to all destinations through *control messages*. Each destination site deliver m in the order of sequence numbers.

Abstract Model of Total Order Broadcast : The abstract B model of a total order broadcast is given in Fig. 4 and 5. In the abstract model a total order is built on the message when it is delivered to any site in the system for the first time. The specification consists of four variables *sender*, *order*, *deliver* and *delorder*. The *sender* is a partial function from *MESSAGE* to *SITE*. The mapping $(m \mapsto s) \in$ *sender* indicates that message m was sent by the site s. The variable *order* is defined as a relation among the messages. A mapping of the form $(m1 \mapsto m2) \in$ *order* indicates that message *m1* is *totally ordered before* *m2*. In order to represent the

MACHINE *TotalOrder*
SETS *SITE*; *MESSAGE*
VARIABLES *sender, order, delorder, deliver*
INVARIANT $sender \in MESSAGE \rightarrow SITE$
$\wedge$ $order \in MESSAGE \leftrightarrow MESSAGE$
$\wedge$ $delorder \in SITE \rightarrow (MESSAGE \leftrightarrow MESSAGE)$
$\wedge$ $deliver \in SITE \leftrightarrow MESSAGE$
INITIALISATION
$sender := \emptyset \quad || \quad order := \emptyset \quad ||$
$delorder := SITE \times \{\emptyset\} \quad || \quad deliver := \emptyset$

Fig. 4. Abstract Model of Total Order : Initial Part

delivery order of messages at a site, variable *delorder* is used. A mapping $(m1 \mapsto m2) \in delorder(s)$ indicate that a site s has delivered *m1* before *m2*. The variable *deliver* represent the messages delivered to a site following a total order. A mapping of form $(s \mapsto m) \in$ *deliver* represents that a site s has delivered m.

The event *Broadcast* given in the Fig. 5 models the broadcast of a message. Similarly the event *Order* models the construction of a total order on *first ever* delivery of a message to any site in the system. The *TODeliver* models the delivery of the messages when a total order on the message has been constructed.

Constructing a Total Order : The event *Order* models the delivery of a message (mm) at a site (ss) when it is delivered for the *first* time. The following guards of

Broadcast ($ss \in SITE$, $mm \in MESSAGE$) $\cong$
WHEN $mm \notin dom(sender)$
THEN $sender := sender \cup \{mm \mapsto ss\}$
END;

Order ($ss \in SITE$, $mm \in MESSAGE$) $\cong$
WHEN $mm \in dom(sender)$
$\wedge$ $mm \notin ran(deliver)$
$\wedge$ $ran(deliver) \subseteq deliver[\{ss\}]$
THEN $deliver := deliver \cup \{ss \mapsto mm\}$
$\|$ $order := order \cup (ran(deliver) \times \{mm\})$
$\|$ $delorder(ss) := delorder(ss) \cup (deliver[\{ss\}] \times \{mm\})$
END;

TODeliver ($ss \in SITE$, $mm \in MESSAGE$) $\cong$
WHEN $mm \in dom(sender)$
$\wedge$ $mm \in ran(deliver)$
$\wedge$ $ss \mapsto mm \notin deliver$
$\wedge$ $\forall m.(m \in MESSAGE \wedge (m \mapsto mm) \in order \Rightarrow (ss \mapsto m) \in deliver)$
THEN $deliver := deliver \cup \{pp \mapsto mm\}$
$\|$ $delorder(ss) := delorder(ss) \cup (deliver[\{ss\}] \times \{mm\})$
END

Fig. 5. Abstract Model of Total Order : Events

this event ensures that the message(mm) has not been delivered elsewhere and that each message delivered at any other site has also been delivered to the site(ss):

$$mm \notin ran(deliver)$$
$$ran(deliver) \subseteq deliver[\{ss\}]$$

Later in the refinement we show that this is a function of a designated site called *sequencer*. As a consequence of the occurrence of the *Order* event, the message mm is delivered to site ss and the variable *order* is updated by mappings in ($ran(deliver)$ $\times$ $\{mm\}$). This indicates that all messages delivered at any site in the system are *ordered* before mm. Similarly, the delivery order at the site ss is also updated such that all messages delivered at ss precedes mm. It can be noticed that the total order for a message is built when it is delivered to a site for the *first* time.

The event $TOdeliver(ss,mm)$ models the delivery of a message mm to a site ss respecting the *total order*. As the guard $mm \in ran(deliver)$ implies that the mm has been delivered to at least one site and it also implies that the total order on the message mm has also been constructed. Later in the refinement we show that site ss represents a site other than the *sequencer*. The guards of the event ensure that message mm has already been delivered elsewhere and that all messages which precedes mm in the abstract total order has also been delivered to ss.

After constructing an abstract model of a total order we verify that this model preserves the total order properties. The agreement and total order requirements imply that all correct process eventually deliver all messages in the same order [8]. Therefore we add following invariant to our model as a primary invariant.

$$\forall(m1, m2, p).((m1 \mapsto m2) \in delorder(p) \Rightarrow (m1 \mapsto m2) \in order)$$

This invariant states that if two messages, irrespective of the sender, are delivered at any site then their delivery order at that site corresponds to the abstract total order. In order to discharge the proof obligations associated with this invariant we also discover new invariants. The process of discovery of invariants is explained in [19]. Similarly, in order to verify that our model also preserves the transitivity property, we added following invariants to our model and discharge the proof obligations associated with the invariant.

$$\forall(m1, m2, m3).((m1 \mapsto m2) \in order \wedge (m2 \mapsto m3) \in order \Rightarrow \ (m1 \mapsto m3) \in order)$$

Overview of the Refinement Chain : Instead of presenting the full refinement chain in the detail we will just briefly present the overview of the refinement steps. Our refinement chain consists of six levels. A brief outline of each level is given below.

L1 This consist of abstract model of total order broadcast. In this model, the abstract total order is constructed when a message is delivered to a site for the first time. At all other sites a message is delivered in the total order.

L2 This is a refinement of abstract model which introduces *sequencer*. In this refinement we demonstrate that the total order is built by the *sequencer*. In the refined specifications of the *Order* event we outline that the *first ever* delivery of a message is done at the sequencer. In order to do that the guards of *Order* event in the abstract specification [$mm \notin ran(deliver) \wedge ran(deliver) \subseteq deliver[pp]$ are replaced by $[ss{=}sequencer \wedge (sequencer \mapsto mm) \notin deliver]$. Similarly a guard $ss \neq sequencer$ is added in the specifications of *TODeliver* event. Thus on the occurrence of *TODeliver* event a message is delivered to the sites other than the sequencer.

L3 This is a very simple refinement giving a more concrete specification of the *Order* event. Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer. Recall that a total order in the abstract specifications are constructed as below which state that all messages delivered at any process are ordered before the new message *mm*.

$$order := order \cup (ran(deliver) \times \{mm\})$$

In the refined specifications of *Order* event the *total order* is constructed as below stating that all messages delivered to the sequencer are ordered before the new message *mm*.

$$order := order \cup (deliver[\{sequencer\}] \times \{mm\})$$

L4 In this refinement we introduce the notion of *computation* messages. The computation message are those message which need to be delivered following a total order. Global sequence numbers of the computation message are generated by the sequencer. The delivery of the messages is done based on the sequence numbers. In this intermediate refinement step, the sequence number of computation messages are assigned by the sequencer. This refinement introduces the following new variables.

$$computation \subseteq MESSAGE$$
$$seqno \in computation \nrightarrow Natural$$
$$counter \in Natural$$

The variable *seqno* is used to assign sequence numbers to the computation messages. The *counter* is updated by one each time a *computation* message is assigned a sequence number.

Table 1. Proof Statistics

Model	Total POs	POs by Interaction	Percent. automatic
Total Order Broadcast	106	27	74
Causal Order Broadcast	80	26	67
Total Causal Order Broadcast	163	67	58

L5 In this refinement we introduce the notion of *control* messages. We also introduce the relationship of each *computation* message with the *control* messages. This refinement consists of following new state variables typed as follows,

$$control \subseteq MESSAGE$$
$$messcontrol \in control \rightarrowtail computation$$

The variables *control* and *computation* are used to cast a message as either a computation or a control message. The set *control* contains the control messages sent by the sequencer. The variable *messcontrol* is a partial injective function which defines relationship among a control message and its computation message. It implies that there can only be a one control message for each computation message and vice-versa. The set *ran(messcontrol)* contains the computation messages for which control messages has been sent by the sequencer.

L6 A new event *Receive Control* is introduced. We illustrate that a process other than sequencer can deliver a *computation* message only if it has received *control* message for it. It can be noticed that the message delivery to the sites other than the sequencer is done using the sequence number generated by the sequencer.

5 Conclusions

In this paper we have outlined our refinement based approach for formal development of a fault tolerant models of replicated database system. We have presented the abstract B models of total order broadcast and outlined how an abstract total order can be refined by the concrete sequence numbers in the refinement steps. The abortion of the conflicting update transactions originated at different sites may be avoided if the updates are delivered to the participating sites in a total order. However, the total order broadcast does not preserve the causality among the transactions. In [19], we used a refinement approach with Event B for formal development of broadcast system which deliver messages satisfying various ordering properties. We have developed the separate models of total order broadcast, causal order broadcast and total causal order broadcast. The work was carried out on the Click'n'Prove B tool. The tool generates the proof obligations for refinement and consistency checking. These proofs helped us to understand the complexity of the problem and the correctness of the solutions. They also helped us to discover new system invariants which provide a clear insight to the system. The overall proof statistics for various developments including a total order broadcast is given in the Table 1. Our experience with these case studies strengthens our belief that abstraction and refinement are valuable technique for modelling complex distributed system.

References

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

2. Jean-Raymond Abrial and Dominique Cansell. Click'n' Prove: Interactive proofs within set theory. In *TPHOLs*, pages 1–24, 2003.
3. Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par*, volume 1300 of *Lecture Notes in Computer Science*, pages 496–503. Springer, 1997.
4. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
5. Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.
6. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
7. Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.
8. V. Hadzilacos and S.Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94 -1425, Cornell University,NY, 1994.
9. Linas Laibinis, Elena Troubitsyna, Sari Leppänen, Johan Lilius, and Qaisar A. Malik. Formal service-oriented development of fault tolerant communicating systems. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 261–287, volume 4157 of Lecture Notes in Computer Science, Springer,2006.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
11. Leslie Lamport and Nancy A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. 1990.
12. P. M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):17–25, 1990.
13. C Metayer, J R Abrial, and L Voison. Event-B language. RODIN deliverables 3.2, http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, 2005.
14. Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, 1996.
15. M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition.* Prentice-Hall, 1999.
16. Andre Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communication of the ACM*, 39(4):84–87, 1996.
17. C. Toinard, Gerard Florin, and C. Carrez. A formal method to prove ordering properties of multicast systems. *ACM Operating Systems Review*, 33(4):75–89, 1999.
18. Divakar Yadav and Michael Butler. Application of Event B to global causal ordering for fault tolerant transactions. In *Proc. of Workshop on Rigirous Engineering of Fault Tolerant System,REFT05*, pages 93–103,Newcastle upon Tyne, 19 July 2005 , http://eprints.ecs.soton.ac.uk/10981/.
19. Divakar Yadav and Michael Butler. Formal specifications and verification of message ordering properties in a broadcast system using Event B. In *Technical Report,School of Electronics and Computer Science, University of Southampton, Southampton,UK*, May 2007, http://eprints.ecs.soton.ac.uk/14001/.
20. Divakar Yadav and Michael Butler. Rigorous design of fault-tolerant transactions for replicated database systems using Event B. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 343–363, volume 4157 of Lecture Notes in Computer Science, Springer,2006.