Crash-Recovery Distributed System

Survey Paper

Submitted to:

Dr. Niraj Mittal

Submitted by:

Divya Channegowda,
Niraj Agarwal

Date Submitted:

October 20, 2005

TABLE OF CONTENTS

1.0 INTRODUCTION 1

1.1 CRASH NO RECOVERY AND CRASH RECOVERY MODEL 1

2.0 PROBLEM DEFINITION 2

2.1 CONSENSUS 2
2.2 RELIABLE BROADCAST 2
2.3 ATOMIC BROADCAST 2

3.0 SYSTEM CHARACTERISTICS 4

3.1 PROCESSES 4
3.2 FAILURE DETECTOR 4
3.3 STORAGE 4
3.4 CHANNEL 5

4.0 ALGORITHM 6

4.1 CONSENSUS 6
4.2 RELIABLE BROADCAST 8

5.0 REFERENCES 9

1.0 INTRODUCTION

The design and verification of a distributed system is viewed as complex endeavor. There are different open issues which have practical implication and required to be solved in distributed system with different characteristics. Following are the few important issues:

- Consensus
- Reliable Broadcast
- Atomic Broadcast

The details are discussed in next section.

1.1 CRASH NO RECOVERY AND CRASH RECOVERY MODEL

Initial work in distributed system were done with the assumption that the process crashes and does not recover. In real world, however a process may crash and recover after some time. Such networks are called the crash recovery network. In this survey paper, we focus only on the problems in the crash recovery model. The next section defines the problem in the crash recovery environment.

2.0 PROBLEM DEFINITION

2.1 CONSENSUS

With consensus, each process proposes a value and processes must reach a unanimous decision on one of the proposed values. Any algorithm-solving Consensus must satisfy the following properties:

- **Uniform Validity**: If a process decides v then some process must have previously proposed v.
- **Agreement**: Good processes do not decide different values.
- **Termination**: If all good processes propose a value, then they all eventually decide.

A stronger version of consensus, called uniform consensus, requires:

- **Uniform Agreement**: Processes do not decide different values.

2.2 RELIABLE BROADCAST

Reliable Broadcast guarantees that

1. all correct processes deliver the same set of messages,
2. all messages broadcast by correct processes are delivered, and
3. no spurious messages are ever delivered.

Formally, Reliable Broadcast is defined in terms of two primitives, R-broadcast (m) and R-deliver (m) where m is a message drawn from a set of possible messages. When a process executes R-broadcast (m), we say that it R-broadcasts m, and when a process executes R-deliver (m), we say that it R-delivers m. Reliable broadcast properties are discussed later in this section.

2.3 ATOMIC BROADCAST

Atomic Broadcast (also called Total Order Broadcast) is a Reliable Broadcast that satisfies:
If two correct processes p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'. The properties are discussed next.

Properties:
To define the Reliable Broadcast properties, [5] define different flavor of the basic properties:

- Validity (V): If a correct process broadcasts a message *m*, then it eventually delivers *m*.
- Agreement (A): If a correct process delivers a message *m*, then every correct process eventually delivers *m*.
- Integrity (I): For any message *m*, every correct process delivers *m* at most once, and only if *m* was previously broadcast by *sender (m)*.

Note that no restriction is made on the order of message delivery in a reliable broadcast.

The different flavors of the properties discussed above include:

- General (1): This property considers the process those does not crashes.

- Uniform (2): This property considers the process those are correct. In other words, processes those crashes and recovers.
- Strong Uniform (3): This property considers the process those might crash forever in the future.

Note that using the above combination there are nine different properties (V1, V2, V3, A1, A2, A3, I1, I2, I3). The authors then combine the properties to form the weakest to the strongest:

- V1, A1, I1 (weakest)
- V2, A2, I2 (uniform)
- V3, A3, I3 (strongest)

It is possible to combine properties of different kinds. For instance, one could define the *weakly uniform reliable broadcast* by combining properties V.1, A.2, and I.2. This specification can be interesting in the context of replication. If the client crashes, then it is not necessary for the replicas to deliver the request and send back a reply, unless the client recovers and broadcasts again its request.

Total Broadcast (TO): Similarly to define the total order broadcast, three flavors (TO1, TO2, TO3) are defined.

3.0 SYSTEM CHARACTERISTICS

3.1 PROCESSES

The processes are classified as follows in the crash recovery model (the terms differ in different papers)

1. Good Process:
   a. **Always-up**: Process p never crashes
   b. **Eventually-up**: Process *p* crashes at least once, but there is a time after which *p* is permanently up.
2. Bad process:
   a. **Eventually-down**: There is a time after which process *p* is permanently down.
   b. **Unstable**: Process *p* crashes and recovers infinitely many times.

3.2 FAILURE DETECTOR

In the crash-recovery model, a process may keep on crashing and recovering indefinitely (unstable process). Such process may be as useless to an application as one that permanently crashes. For example, an unstable process can be up just long enough to be considered operational by the failure detector, and then crash before "helping" the application and this could go on repeatedly. At any given point in time, no implementation can predict the future behavior of a process *p* that has crashed in the past but is currently "up". Will *p* continue to repeatedly crash and recover? Or will it stop crashing? This leads to redefine the completeness and the accuracy properties. The consensus algorithms proposed in [1] requires the failure detector properties (defined in the crash sensitive) to be modified for the crash recovery model.
The new failure detectors for crash recovery model should satisfy the following properties:

Monotonicity: At every good process, eventually the epoch numbers are non-decreasing.
Completeness: For every bad process *b* and for every good process *g*, either eventually *g* permanently suspects *b* or *b*'s epoch number at *g* is unbounded.
Accuracy: For some good process *K* and for every good process *g*, eventually *g* permanently trusts *K* and *K*'s epoch number at *g* stops changing.

This failure detector does not output lists of processes suspected to be crashed or unstable. Instead, at each process *p*, the output of failure detector consists of two items, *trustlist; epoch*, where *trustlist* is a set of processes and *epoch* is a vector of integers indexed by the elements of *trustlist*. Process *q belongs to trustlist* if *p* believes that *q* is currently up, and *epoch*[*q*] is *p*'s rough estimate of how many times *q* crashed and recovered so far (it is called the *epoch number of q at p*).

To solve consensus problem, [2], [3] use the failure detector which satisfies the following properties:
- Strong completeness
- Eventually weak accuracy

3.3 STORAGE

A process is equipped with two local memories: a volatile memory and a stable storage. When it crashes, a process looses the content of its volatile memory; the content of its stable storage is however not affected by the crash and can be retrieved by the process upon recovery.
Is stable storage always necessary when solving consensus? If not, under which condition(s) can it be completely avoided? This issue is discussed in [1].

Let p be the processes which are always up and q be the bad processes.

If $p \leq q$ then consensus *cannot be solved without stable storage* even using ◊P (the *eventually Perfect failure detector*).

If $p > q$ then consensus *can be solved without stable storage* using ◊S.

3.4 Channel

The channel (a.k.a link) properties are common across all the papers, which assume every message *m* includes the following fields: the identity of its sender, denoted *sender (m),* and a local identification number, denoted *id(m).* These fields make every message unique. Channels can loose or drop messages and there is no upper bound on message transmission delays. Following are the properties of the link.

***No** creation:* If *p,* receives a message *m* from *p,* at time *t,* then *p,* sent *m* to *p,* before time *t.*
Finite duplication: If *p'* sends a message *m* to *p,* only a finite number of times, then *p,* receives *m* only a finite number of times.
Fair loss: If *p'* sends a message *m* to *p,* an infinite number of times and *p,* is correct, then *p,* receives *m* from *p'* an infinite number of times.
These properties characterize the links between processes and are independent of the process failure pattern occurring in the execution.

4.0 ALGORITHM

4.1 CONSENSUS

An algorithm for solving Consensus in the Crash/Recovery model with stable storage has been proposed in [1][2].

Solving Consensus without Stable Storage as proposed in [1] is as follows:

In each round r, initially the coordinator c broadcasts a NEWROUND message to announce a new round, and each process sends its current estimate of the decision value together with a timestamp indicating in which round it was obtained to c; c waits until it obtains estimates from a majority of processes; it selects one with the largest timestamp and sends it to all processes; every process that receives this new estimate updates its estimate and timestamp accordingly, and sends an acknowledgement to c; when c receives this acknowledgement from a majority of processes, it sends its estimate as the decision to all processes and then it decides.

In each round, a process p accesses the stable storage twice: first to store the current round number, and later to store the new estimate and its corresponding timestamp. Upon recovery, p reads the stable storage to restore its round number, estimate, and timestamp, and then restarts the all phases of the algorithm with these values.

The consensus algorithm proposed in [2], also proceeds in consecutive asynchronous rounds. Each process p_i consists of a local variable est_i, which contains its local estimate of the destination value. On each round r, a predetermined process p_c (c = (r mod n) + 1) is assigned to be the current initiator. At the beginning of r, the current initiator p_c sends its value est_c to all. During each round r, a process votes either for deciding est_c at the current round or for proceeding to the next round. During the current round, p_i considers est_c as the decision value as soon as $\left| endor\sin g_groupi \right| \succ n/2.$ It proceeds to the next round if $\left| next_round_groupi \right| \succ n/2.$ In order to prevent infinite blocking situations, the protocol introduces an asymmetry in the automaton. A process can first vote for a decision to be taken during the current round and then vote for proceeding to the next round.

Replaying lost messages

When a process p_i recovers, it first retrieves the last set of data that have been logged: r_i, est_i and $state_i$. Then, it resumes its execution from round r_i and replays it according to the value of $state_i$. If $state_i$ = q1, it issues a vote for deciding in this round; if $state_i$ = q2, it issues a vote for proceeding to the next round. Note that these votes have possibly been issued before crashing (their re-sending ensures the protocol will not block). Moreover, p_i indicates in its vote it is recovering. This is done by broadcasting STATE(p_i, r_i, est_i, $state_i$, $recovery_i$), where $recovery_i$ is a boolean whose value is *true* (when the process is not recovering, $recovery_i$ is *false*). When a process p_k receives such a message from p_i (with $recovery_i$ = true), it answers by replaying the last vote it had previously sent: this is done by sending back to p_i the message STATE(p_k, r_k, est_k, $state_k$, false). When p_i receives it, it first joins round r_k if $r_k > r_i$. Then, it processes the message received as indicated in the previous paragraph.

Several Consensus algorithms suited to Crash/Recovery distributed systems without a stable storage have been proposed [1,3]

Solving Consensus without Stable Storage as proposed in [1]

Processes proceeds in asynchronous rounds. Each round consists of two stages. In the first stage each process sends WAKE UP message to the coordinator *c*, so that the coordinator can start the current round (if it has not started yet). Coordinator sends a NEWROUND message indicating that it has started a new round and each process sends to *c* its current estimate of decision value along with the timestamp. Then *c* waits for estimates from max(nb+1, na) processes. Then c checks whether during the collection of estimates it detected the recovery of a process that never recovered before (Rc not equal PrevRc). If so, c restarts the first stage from scratch. Otherwise, c chooses the estimate with the largest timestamp as its new estimate and proceeds to the second stage.

In the second stage, coordinator broadcasts its new estimate of the decision value to all processes. And each process sets its current decision value with the one received from the coordinator and sends an ACK to *c*. Coordinator wait for max(nb+1, na) ACK messages and during the collection of ACK messages if it detects the recovery of a process which never recovered before, restarts the second stage from the beginning. Otherwise, *c* finally broadcasts its estimate as the decision value and decides accordingly.

A round r can be interrupted in the following cases.

- If a process suspects *c* of round r, as the epoch number of *c* is non decreasing.
- A process detects the recovery of coordinator *c*
- A process receives a message from round r' > r

When a process p aborts a round r, it jumps to a lower round such that

- *p* trusts the coordinator c0 of round r0,
- *p* has not detected a recovery of c0 and
- *p* has not (yet) received any message with a round number higher than r0.[1]

Consensus Algorithm as proposed in [3] is as follows:

The consensus algorithm is based on the rotating coordinator paradigm and proceeds in asynchronous rounds. Every process p_i manages a variable estimate i, which is p_i's current estimation of the decision value, and a variable r_i representing p_i's current round number. In every round, there is one process that plays the role of the coordinator, and this process is known a priori to all processes, e.g., in round 0 the coordinator is p_1 , in round 1 the coordinator is p_2, etc. In every round r, the coordinator p_c tries to impose its estimate c as the decision value, by sending estimate c to all processes. When a process p_i receives estimate c from the coordinator, p_i forwards estimate c to all processes. A process decides on estimate c as soon as it has received estimate c from a majority of processes. The protocol terminates in the first round if the first coordinator, p_1, is not suspected. Otherwise (i.e., if p_1 is suspected), the processes proceed to the second round, and so on. Before proceeding from a round to another, the estimates of the processes are updated, so as to satisfy the following invariant:
If some process has decided on estimate c in round r, then any process that proceeds to round r + 1, starts round r + 1 with estimate c as its current estimate.
In phase 1 of every round r, the consensus algorithm tries to decide on the estimate value of the coordinator p_c of round r. r is used to define a new consensus try, to be performed in round r + 1. The initial value of process p_i for the consensus of round r + 1 is the estimate of p_i at the end of phase 2 of round r.

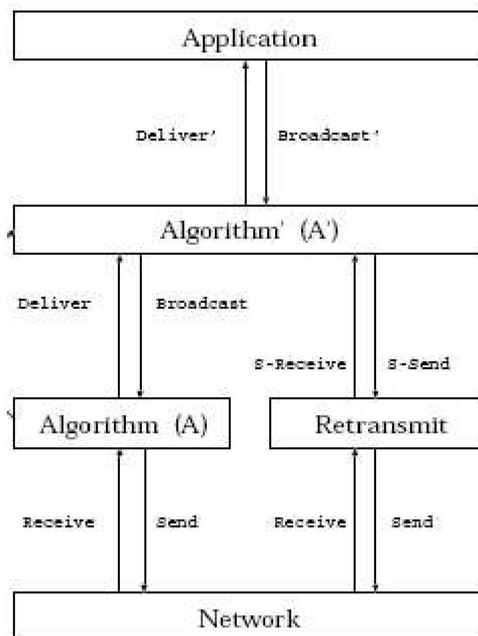
All algorithms solving consensus with stable storage require a majority of good processes and rely on the semantics of stubborn communication channels. The algorithm proposed in [1] uses a hybrid fail failure detector which satisfies strong completeness regarding Eventually Down processes and handles the detection of Unstable processes by providing an estimate count of the number of recoveries of all processes. Where as the protocol described in [2] uses a failure detector satisfying Strong Completeness and Eventual Weak Accuracy.
Both the algorithms proposed in [1][2] require each process to log the critical data in every round.

All the consensus algorithms above explained, also satisfies Uniform Agreement property.

4.2 RELIABLE BROADCAST

Reliable Broadcast using the Transmission module
A *retransmit* module is proposed in [5], which make sure that a message is delivered to the process. In case a process crashes and then recovers back, the retransmission module handle the situation by making sure that any message sent are delivered to the processes. This is achieved using acknowledgment.

Assuming that basic reliable broadcast algorithm **A** (satisfying weakest properties) is available, using *retransmit* module, a transformation module is defined which modifies A to **A'** that satisfies the uniform properties, as shown in the figure below. As mentioned above the no crash recovery algorithm A, is transformed to crash recovery algorithm using the Retransmit module, which retransmits the messages until the acknowledgement is reached. Using the same principle a uniform reliable broadcast algorithm (URB) can be transformed to strong uniform reliable broadcast algorithm (SURB).



5.0 REFERENCES

[1] Aguilera M.K., Chen W. and Toueg S., Failure Detection and Consensus in the Crash-Recovery Model. In *Proc. 11th Int. Symposium on Distributed Computing (DISC'98, formerly WDAG)*, Springer-Verlag, LNCS 1499, pp. 231-245, Andros, Greece, September 1998.

[2] M. Hurfin, A. Mostefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems, IEEE Comput., pages 280–286, Soc, Los Alamitos, CA, 1998.

[3] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report TR-97/239, EPFL – D´epartment d'Informatique, Lausanne, Switzerland, Aug. 1997.

[4] Rachid Guerraoui, Michel Hurfin, Achour Mostefaoui, Riucarlos Oliveira, Michel Raynal, and Andre Schiper. Consensus in Asynchronous Distributed Systems: A Concise Guided Tour.

[5] R. Boichat and R. Guerraoui. Reliable broadcast in the crash-recovery model. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nuremberg, Germany, Oct 2000.

[6] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(4), 2003.