Fault tolerant consensus protocol for distributed database transactions

Marius Rafailescu
The Faculty of Automatic Control and Computer Science
POLITEHNICA University
Bucharest, Romania
marius.rafailescu@cti.pub.ro

Mircea Stelian Petrescu
The Faculty of Automatic Control and Computer Science
POLITEHNICA University
Bucharest, Romania
mircea.petrescu@cs.pub.ro

ABSTRACT

Defining a protocol for commit consensus problem in distributed database transactions is not an easy task because we have to think about a fault tolerant system. In this paper, I present a fault tolerant and simple mechanism designed for solving commit consensus problem, based on replicated validation of messages sent between transaction participants.

CCS Concepts

• Information systems → Distributed database transactions;

Keywords

commit consensus; distributed database transactions; fault tolerance; leader election

1. INTRODUCTION

Commit consensus was discussed in the past and several solutions where developed (Two-phase commit, Three-phase commit or Paxos) [1]. The latter is fault tolerant and with the introduction of distributed databases it was implemented in many systems, although it is not so easy to implement [2]. In recent years was defined a new algorithm, Raft, which has been developed in order to provide a consensus control for replicated state machines, intended to be easy to understand and implement [3]. In this paper is presented a new algorithm which uses a set of validator nodes, including one dispatcher. There is also presented an algorithm for dispatcher election (equivalent to leader election) made for the purpose of not rollbacking the transaction when a new dispatcher is chosen.

2. GENERAL DESCRIPTION

In distributed transactions, several participant nodes work together to get the job of the transaction done. We will consider that the transaction coordinator is the node who initiated the transaction. In order to build a fault tolerant mechanism, we define a set of nodes to validate some individual messages, the validators. One of these nodes will have, additionally, a special role and it will act as a dispatcher for all others validators (it will intermediate messages between participants and validators).

ICMSS '17, January 14-16, 2017, Wuhan, China
© 2017 ACM. ISBN 978-1-4503-4834-8/17/01…$15.00
DOI: http://dx.doi.org/10.1145/3034950.3035007

Validating means, at least, saving informations about the message being sent.

The messages of this protocol are described in figure 1.

First of all, the initiator (the first participant in the figure) sends a message “begin commit” intended to announce the other participants that transaction operations have finalized. Also, it sends one message “ready” to the dispatcher node to validate/mark the node intention to commit. The other participants will also send the “ready” message as they receive the “begin commit” message.

When the dispatcher node receives the message “ready”, it validates itself the message, then it sends the message to the other validators. When the majority of validators have validated the messages from all transaction participants, the dispatcher will send the “commit” message to all transaction participants and the “committed” message to the other validators. Validation process is actually a simple replication method, similar to what Viewstamped Replication [4] proposes.

3. COMPLEXITY

The complexity is measured by the total number of messages sent between nodes. Considering we have n participants and m validators, we have a total number of $n[2(m-1)+3]+m-2$ messages:

- $n-1$ “begin commit” messages, sent by the node which initiates the transaction to the other participant nodes;
- n “ready” messages, received by the dispatcher node from all participants;
- $2n(m-1)$ validation messages; for every “ready” message, the validation process uses $2(m-1)$ messages sent between dispatcher node and the other validator nodes;
- n “commit” messages, sent by dispatcher node to all transaction participants;
- $m-1$ “committed” messages, also sent by dispatcher node, but for all validator nodes.

Nowadays, messages are quite cheap. More important are the number of writes on the hard-disk; in this algorithm the nodes need to write on disk in two moments. First time, before sending the “ready” message in order to achieve fault tolerance in case of a participant crash and the second time, after the “commit” message, when all changes are written in database files.

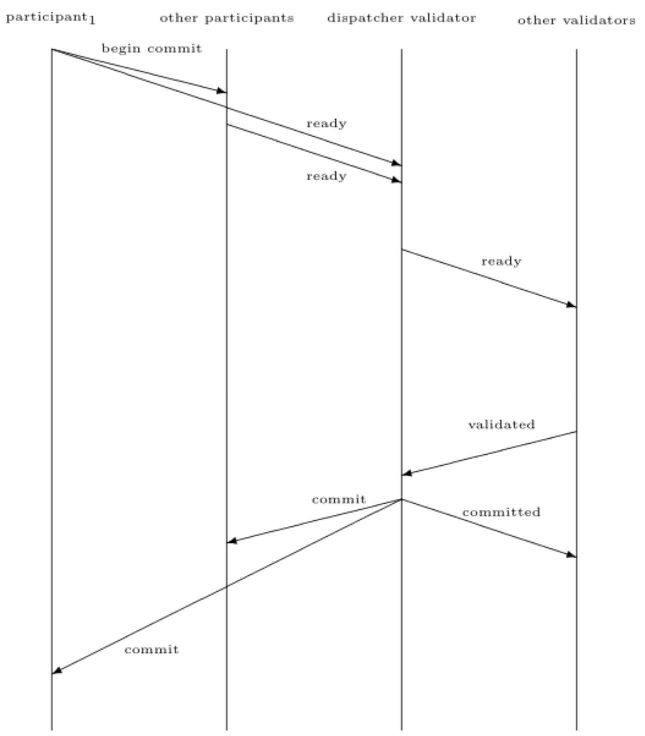

Figure 1. Algorithm messages illustration

4. FAULT TOLERANCE

One important result of previous work on consensus algorithms is that no completely asynchronous consensus protocol can tolerate even a single unannounced process death [5]. That is why it is necessary to use timeout-based methods in order to ensure correct termination for such an algorithm.

In this paper, the algorithm do not consider Byzantine failures, assuming that all the messages are delivered correctly and exactly once. Byzantine failures refer some issues initially presented under the name of “Byzantine Generals Problem” [6]. This describes the situation when a group of generals, each controlling some troops, must agree on the next action they should take. This is done by sending messengers that can fail their goal (similar to distributed nodes failures). Moreover, another problem is that some generals may be traitors, in group or individually, falsifying messages in order to confuse the other loyal generals.

To verify the fault tolerant behavior we have to take into consideration all the cases when one node can interrupt its execution.

4.1 Validator node problems

A blocked validator will be identified by the dispatcher validator, based on the heartbeat messages that the latter will regularly send; when this happens, the validator is taken out of the network automatically. Whenever a validator stops its execution, the system continues to work correctly if the initial majority can be achieved.

After the problem is solved, the validator can re-enter in the network and first of all it has to find out which node is the dispatcher (it will start a communication with other node and the latter will respond with the information regarding the dispatcher). After the dispatcher has been found, the node has to get the list of current states of transactions that are about commit. After this list is saved in the local memory, the new validator sends a confirmation message and after that it will be accepted in the network.

4.2 Dispatcher node problems

When the dispatcher has a technical problem and stops from working, based on the heartbeat messages, the other validators will start an election for a new dispatcher. The system will be blocked for some moments, until the election ends. After that, the transactions will continue to run in the normal way. The proposed algorithm is presented in section 5.

The advantage of this algorithm is that the flow can continue normally after the new dispatcher is chosen. To ensure this, valid candidates can only be those nodes which received all the “ready” messages send by dispatcher before crash.

When the dispatcher crashes before sending a “ready” message to validators, the transaction will be eventually rollbacked. Somehow, this case can be limited if we consider an additional logic: after the new dispatcher is chosen, it verifies the “ready” messages for a specific transaction and starts a communication with those participant nodes which are not validated; if those participants nodes are running, the transaction will be committed. This case can also be solved based on “ACK” messages, sent by dispatcher as a confirmation for “ready” messages after these are validated - most useful when the first “ready” message is lost. The dispatcher can also crash right before sending the “commit” or “committed” messages.

This cases are solved by using a special consistency related procedure, presented in detail in section 5.1.1.

4.3 Participant node problems

When a participant node stops after it transmitted the “ready” message, then there are no problems. It is its responsibility to assure that, in case of a crash after the “ready” message was received and validated, the transaction will be eventually committed. This is achieved by using redo logs, files in which all the modifications on database resources are being saved.

In other cases, the transaction will be rollbacked:

- when a participant stops before sending “ready” message;
- when a participant stops right after it sends “ready” message and the dispatcher crashes before validating the message. Here we can use the logic presented previously in section 4.2, but in order to commit the transaction eventually, it is mandatory that the participant node to be running after a new dispatcher is elected.

5. DISPATCHER ELECTION

Sometimes the dispatcher can have technical problems, not being able to work properly. In this cases, we have to apply a clear procedure in order to choose a new dispatcher. The proposed algorithm consists in a finite local mechanism which collects all necessary informations from other nodes in order to apply eventually a random roulette wheel selection algorithm:

1. There has to exist one node which will have tocoordinate the roulette selection; this node is chosen randomly and can advance to the next step of the algorithm only by satisfying some condition; for instance, one condition can be based on

generating a random number in (0, 1) interval, but greater than a chosen threshold, let's say 0.85; when this condition is satisfied, the node becomes candidate and sends the generated number to other nodes;

2. When one node receives the number from another node, it has to vote for the sender node if it has not voted before for a bigger value in the current round of vote; moreover, it sends its greatest random generated number in its response;
3. If the candidate node receives all the positive votes, it will effectively become the coordinator which will run the random roulette wheel selection algorithm using all the numbers received from other nodes;
4. The winner will be the new dispatcher; the coordinator will send it a message and after receiving it, the dispatcher will broadcast a special/heartbeat message to announce its new role.

5.1 Additional specifications

There are some special cases which need additional specifications:

1. Valid candidates can only be those nodes which received all the messages send by dispatcher before crash; every message sent for validation will have an unique ID generated using a monotonically increasing function. So every node will also send in its response the ID of the last message received from the last dispatcher. Using this information, the coordinator will run the final selection algorithm only with those nodes which have the biggest message ID.
2. If we consider the solution based on random number generation, we have to admit that multiple nodes might generate nearly in the same time some numbers greater than the chosen threshold.

   The probability for one node to generate a random number greater than 0.85 is 0.15. To limit this probability, we could take into consideration changing this condition; we can redefine the rule such as one node need to generate one number greater than threshold multiple times sequentially; so the probability will be 0.0225 for two numbers and 0.003375 for three numbers - using the formula for independent events probability [7];

   Also it is necessary to specify that every node which managed to satisfy the condition, votes first of all for itself and it will vote for other node only if the number received is greater than its own number; moreover, when such a node receives a negative response, it can revert its state as long as this means that its number is lower;

   Furthermore, it could be a problem if two nodes generate the same big number. The solution consists in generating random numbers with many decimal places (6 would be sufficient). However, if this happens and no coordinator can be selected, the algorithm will continue with another round of vote;
3. One node can become coordinator by waiting responses from all the nodes, using a timeout - when all the nodes respond or the timeout elapse, the candidate verifies the votes and it will become coordinator if it has the majority of them and, eventually, it will run the roulette wheel with all numbers from other nodes;
4. If one node obtains the majority of votes and becomes coordinator, but after that receives a proposal with a greater number, it will send a negative response with a special flag saying that it is already coordinator in the current round of vote;
5. If one node which satisfies the launch condition has technical problems, it will stop until the problem is resolved, but the algorithm will continue through a new round of vote;
6. If one node does not generate a value bigger than threshold, then it will continue to generate numbers until it gets a proper number or until he receives a message from other node;
7. If the coordinator suffers a problem, the algorithm will continue through a new round of vote;
8. If the node which is chosen to be the new dispatcher suffers a problem right before the coordinator announce it, the coordinator will choose another dispatcher if it does not receive a heartbeat message;
9. After the new dispatcher is chosen, it is necessary to apply an additional logic regarding validated messages consistency;
10. Every node knows anytime how many nodes are up; at any moment of time when the majority can not be satisfied, the system will block until some nodes are recovered.

5.1.1 *Messages consistency check*

As mentioned in section 4.2, after the new dispatcher is chosen, is necessary to apply a procedure in order to keep consistency regarding all the messages sent between validators and between dispatcher and participants.

After the new validation leader is found, before continuing the consensus algorithm, its task is to ensure that all the validators share the same state (the same list of pending, waiting for commit, transactions).

The first thing it has to do is to send his list of pending transactions to all other validators. The latter will answer with an "ACK" message as confirmation for applying all the necessary changes in the local memory.

In the next step, the new dispatcher prepares the "commit" messages which need to be sent, if there are any. When it broadcasts its role, it will also send the "commit" messages to transaction participants and "committed" messages to other validators. If some participants received from the old dispatcher the "commit" message, they would simply ignore the message.

The two-step validation presented here is useful in multiple cases:

- when the old dispatcher crashes right before sending all the "commit" messages to participants;
- when it crashes after sending all the "commit" messages, but right before sending the first "committed" message;
- when the old dispatcher crashes after sending all the "commit" messages, but right before sending all the "committed" messages, the new dispatcher will simply take no action regarding that committed transactions. This is possible because the dispatcher sends "commit" messages before "committed" messages. Based on election candidate eligibility described in section 5.1, the new validation leader will have its pending transactions list modified accordingly with the lastest message sent by the old dispatcher.

There may be cases when the old dispatcher crashes before sending one "ready" message for validation. As a response to the first dispatcher heartbeat (when it broadcasts its role), participants may retry all the “ready” messages for uncommitted transactions, related on the idea mentioned in section 4.2.

5.1.2 *Final algorithm*

The modified algorithm is the following:

1. Each node generates random numbers in (0, 1) interval. If three consecutive generated numbers are greater than the chosen threshold, then it will send to other nodes the greatest generated number and it votes for itself;
2. When one node receives the number generated by other node, it has to vote for sender node if it has not voted before for a bigger value in current round of vote; moreover, it sends its greatest random generated number and the ID of the last message received from the broken dispatcher in its response;
3. The node which receives all the positive votes will be the coordinator in order to run the random roulette wheel selection using all the numbers received from other nodes; the winner will be the new leader;
4. When one node receives a negative vote response, it will invalidate its state;
5. After the leader is chosen, the coordinator will send it a message and after receiving it, the leader will broadcast a special/heartbeat message and will apply the logic regarding validated messages consistency.

To minimize the time a node must wait in order to become coordinator, we may consider that the roulette wheel selection can be started right after the candidate collects the majority of votes (half + 1). This aproach implies a change in every node behavior because it is mandatory to vote only one time in a round, for the first candidate node. In this way, a single node will be elected as coordinator.

As we can see, we identify multiple states a node can be in:

1. Basic mode when a node simply votes for others;
2. Candidate mode when a node already satisfied the launch condition;
3. Coordinator mode when a node received the majority of votes;
4. Dispatcher, final state;

Also, we can see the type of messages which are send between nodes:

1. Proposal message;
2. Positive vote message;
3. Negative vote message;
4. Announcement message;
5. Heartbeat message (first time combined with consistency check message);

6. CONCLUSIONS

The presented consensus algorithm has an advantage consisting in its capacity to continue without rolbacking transactions when the dispatcher fails. I think this algorithm is quite simple and in the future I will continue with a practical algorithm analysis in order to identify potential problems.

7. ACKNOWLEDGMENTS

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU 187/1.5/S/155420.

8. REFERENCES

[1] Gray, J. and Lamport, L. Consensum on transaction commit. 2004

[2] Chandra, T., Griesemer, R. and Redstone, J. 2007. Paxos made live - an engineering perspective. *ACM Principles of distributed computing*, pages 398–407, 2007.

[3] Ongaro, D. and Ousterhout, J. In search of an understandable consensus algorithm (extended version). 2014.

[4] Liskov, B. and Cowling, J. Viewstamped replication revisited. 2012.

[5] Fischer, M. J., Lynch, N. A. and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):398–407, 1985.

[6] Lamport, L., Lamport, R. and Pease, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[7] Skorokhod, V. *Basic Principles and Applications of Probability Theory*. Springer, 2005