Transactions in NoSQL Environments

Manuel Parmiggiani & Giacomo Garbarino

Contents

1 - Introduction

The rising of new technologies, as well as new ways to track user behaviour on the internet, have also tremendously increased the quantity of the collected data, the data amount is approximately 300 times higher than 10 years ago and continues to rapidly grow and double every three to four years.
Traditional RDBMS have the weakness of not performing very well when huge amounts of data have to be processed, the overloaded servers couldn't run properly and this can lead to performance issues, the goal of Large Scale Data Management is to overcome this flaw and enable modern systems to exploit as many machines as possible, to consume as much information as possible and to process data as fast as possible.
The first big change is in the choiche of how the data are stored, standard RDBMS stores data in tables in the form of rows and columns, giving each row an identifier that can be used both to retrieve single records as well as connect them with informations from other tables, relying mainly in SQL for querying it, while several modern data management systems propose a NoSQL environment with a level of schema information that may vary from one system to another, these kind of systems are optimized in handling huge data volumes and have a very good scalability since it's possible to partition the database, so

the increase in the data volume is handled by adding new nodes.

With the introduction of new methodologies, also a new set of issues may arise, this paper will display some of the controversies of the NoSQL environment and some possible solutions to them

2 - Basic notions about transactions

2.1 - Transactions and ACID properties

A transaction is a sequence of data operations that result in an “all or nothing” execution. If all goes well, at the end of the execution the transaction is committed and all of its operations become effective on the database, otherwise the transaction is aborted and the state of the database is reverted to the situation it had before the beginning of the transaction through a rollback operation.

The notion of transaction is very important for data systems, since transactions maintain data consistency, allow to do concurrent data access, can help the recover phase after a fault and so on.
A transaction have to satisfy 4 fundamental properties: the ACID properties.

- Atomicity: the transaction happens at once or doesn't happen at all, there is no possibility to see intermediate results made effective on the database;
- Consistency: the database has to be consistent both before and after the transaction's execution;
- Isolation: if multiple transactions are executed concurrently, the result must be the same of executing them in a sequential order, so a transaction cannot interfere with the work done from another one or see its intermediate results;
- Durability: once the transaction is committed, the results are permanently effective on the database, they persist even after a system failure.

In order to satisfy the ACID properties, relational systems make use of the Write Ahead Logging (WAL) protocol:

- all of the updates are made on a copy of the database stored in the volatile memory and written in a log file before being made effective on the real database;
- when all of the transaction's updates are written in this log file, all of them can be applied to the database by synchronizing it with its transaction-local copy;
- a log file is a sequential file which supports very fast append operations and is stored in a stable memory that theoretically cannot be affected by failures.

2.2 - The issue of Transactions in Large-Scale data management systems

In a large-scale data management system, the notion of transaction is slightly extended since there is the need to keep the database partitioned among multiple nodes due to the very large amount of data to process.
A transaction processed by two or more network nodes is called a distribuite transaction, a distribuite transaction can be seen as a set of sub-transactions obtained from the original one, each of these sub-transactions runs on a different node/server each working on a different chunk of the database.
Starting from the simple assumption that each node can guarantee the atomicity of its local execution, the goal is to set up additional measures in order to obtain a global atomicity of

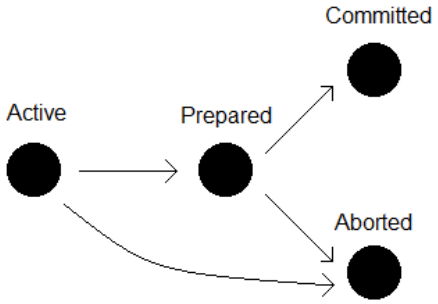
the transaction; in this situation, is challeging to handle machine, software and network failures while preserving transaction integrity.
To achieve atomicity at the global level, there is a need for a synchronization protocol that ensures a unanimous final outcome, this type of protocol will take the name of Atomic Commit Protocol (ACP) since, at the end of the execution, all of the nodes have to do the same action (commit or abort) in order to fulfill the “all or nothing” condition.
The remaining ACID properties can still be guaranteed but they are more difficult to implement, plus there is the need to ensure replica consistency too (if the system follows a replication protocol).

In order to achieve the Atomic Commit condition, many possible solutions were introduced.

3 - Two-Phase Commit protocol

3.1 - Protocol explanation

This first protocol is classified as a “master-slave protocol” in which there are 2 kind of nodes: coordinator (one, master node) and participant (many, slave nodes).
Usually the coordinator node is the first to receive the external command of executing the transaction, once its local execution is finished, that node takes the role of coordinator and runs the Two-Phase Commit (2PC) protocol [1].
The following diagram describes the state transition of each participant node during the execution of 2PC:

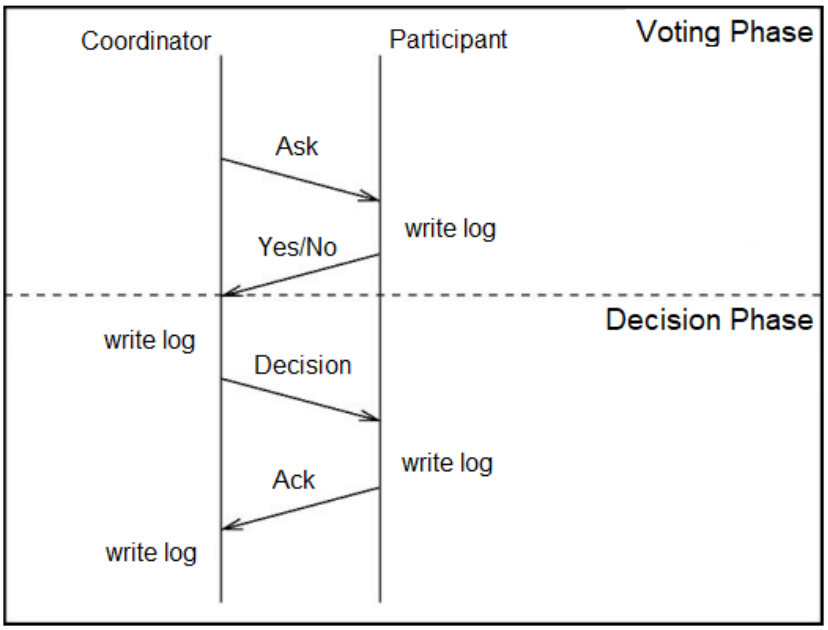


All of the participants will start the execution in the “Active” state, a participant can enter the “Prepared” state on its own and then communicate it to the coordinator, which starts executing the protocol.
As the protocol name suggests, the commit phase is divided into 2 sub-phases:

- voting phase: the coordinator node asks to all of the participant nodes if they can commit, then each participant replies to the coordinator with a positive or negative answer, participants that have voted “Yes” will immediately write a “prepare record” in a log file, move to the “Prepared” state and wait for the coordinator to give them the command to execute while participants that have voted “No” will immediately execute the abort command after voting, releasing their resources that where locked by the current transaction and moving to the “Aborted” state;
- decision phase: the coordinator node elaborates all of the responses received in the previous step, if all are positive then the coordinator sends the commit command to all of the participants, otherwhise it sends the abort command but only to the participants who answered “Yes” during the voting phase since participants who answered “No” have already performed the abort command;

At the end of this step, each participant (except for participants who answered “No” during the voting phase) sends back an acknowledgment to the coordinator in order to confirm it

has executed the requested action.
If the final decision is to commit the transaction, it means all of the participant nodes voted “Yes” and are in the “Prepared” state at the beginning of the decision phase, they will all end the execution in the “Committed” state, otherwise all of them will end the execution in the “Aborted” state;



The change in the state of the participant is determinated by the force-write on its log file, the node changes its internal state after each write.

3.2 - Recovery and open issues

Two-Phase Commit relies in the fail-recover model assuming that, if a participant node crashes, it will recover with the same intention (commit or abort) it had before the failure, this is possible thanks to the fact that nodes records the progress of the protocol execution in an internal log situated in the stable memory of the node, same for the coordinator; using log files simplifies a lot the recovery phase, since the restarted node can remember what it has already done and what is left to do just by reading its logs.
In 2PC, failures are detected by timeouts, if the coordinator or a participant takes too much time to respond when it's its turn to send a message during the protocol execution, then there is an high probability that a node failure has occurred.
The just presented protocol solves the problem of the atomic commit but there are several open points that may arise issues, the system is assumed to follow the fail-recover model but this is not always guaranteed, if a permanent failure occurs, the situation can become quite difficult to handle:

- in case of a permanent failure on the coordinator node occurring before the decision phase, participant nodes will have no idea of what they have to do, in this case using a spare coordinator can help;
- if also a participant node crashes, the result of the transaction is unknown by the system, the solution can be restarting the protocol but in some situations this isn't possibile: for example, if this double crash occurs after the coordinator has sent the decided command to all of the participants, some of them may have already committed and this means no restart or going back possibility.

3.3 - Two-Phase Commit variants

This protocol also offers many variants that can lead to a better overall performance, the 2 most common variants are Presumed Abort and Presumed Commit that reduce the cost of the execution respectively during an abort scenario and a commit scenario.

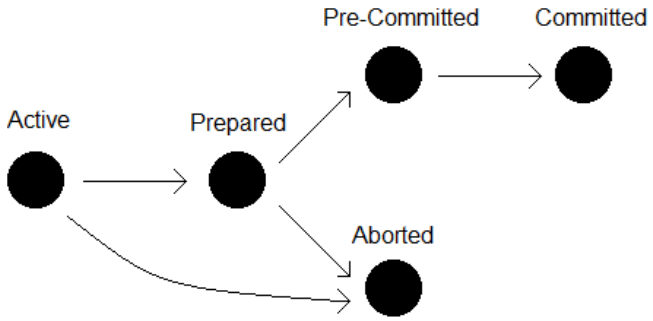
- In presumed abort, when the coordinator decides to abort the current transaction, it simply sends the abort command to all of the interested nodes, then the transaction is considered completed and it's discarded by the coordinator without waiting for acknowledgment to come back nor write about it on the log file. Also the participants avoid write on their local log file when they receive an abort command so when a node recover after a failure, if it finds no informations about what it was doing before crashing, the transaction is presumed to be aborted;
- In presumed commit, the line of action is the same except the presumed status of the transaction is commit instead of abort, the coordinator also has to keep track at least of the commit decision since it's risky to send a commit command by default after resuming the execution;

In real systems, due to the better performance of its variants, pure 2PC is never implemented, the best option seems to be presumed abort since it has an average cost that is even less than the cost of its counterpart (presumed commit) but there are even better performing variants of 2PC not covered in this paper.

4 - Three-Phase Commit protocol

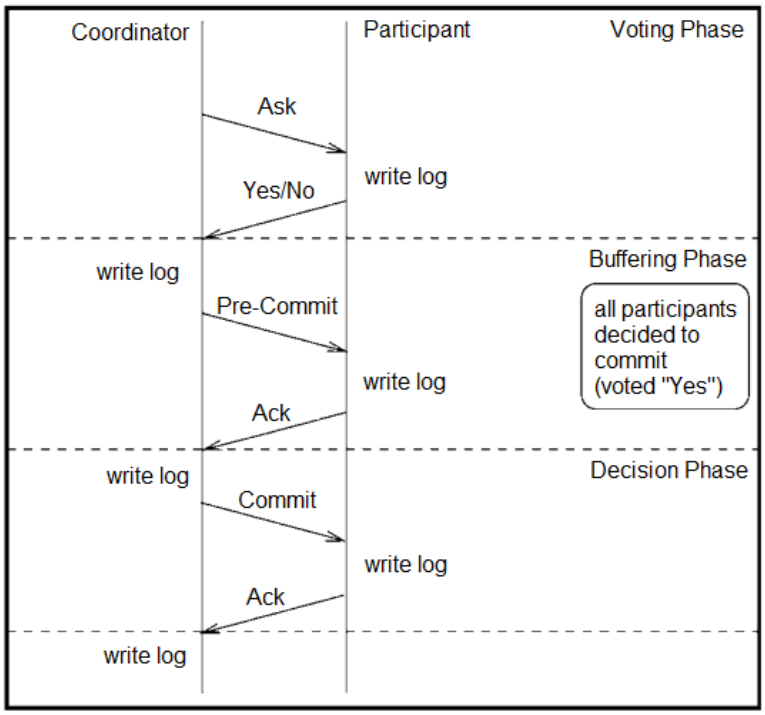
4.1 - Differences with 2PC

Three-Phase Commit (3PC) [2] was introduced as a refined implementation of 2PC that solves its shortcomings.
The most important difference between 2PC and 3PC is the fact that 3PC is non-blocking, an ACP is considered non-blocking if it never requires nodes to wait unitil a failed node has recovered.
In order to achieve this non-blocking behaviour, the first thing to do is specify a set of node states in a way that the 2 final states (committed and aborted) are not reachable anymore from the same state:



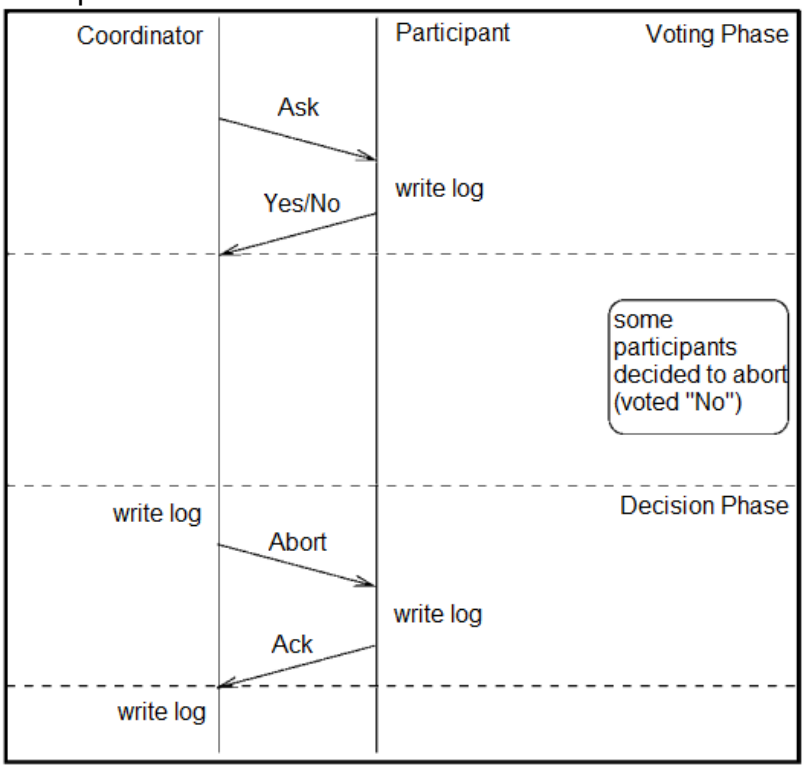
The difference from 2PC is obtained by placing a new state between the “Prepared” state and the “Committed” state, in this way the 2 final states are not adjacent anymore and, if a node recovers in the “Pre-Committed” state, it can deduce on its own that all of the other nodes have voted “Yes” during the voting phase and the coordinator agreed on the commitment of the transaction, because there is no way it can be in such state if not.
This non-adjacency of the final states (plus the non-adjacency of a non-committable state to a commit state) is what 2PC lacks to achieve the non-blocking nature 3PC has.

4.2 - Protocol explanation



The addition of the new state leads to the addition of a new phase (the buffering phase) before the decision phase: here, in case of an unanimous positive result of the voting phase, the result is firstly propagated to all the participants and, only after they send back the acknowledgment of having received it, the coordinator send the order to execute such action. As for the state transition of the participant during the execution, the state is “Active” at the beginning of the voting phase, “Prepared” after sending the vote, “Pre-Committed” after receiving the pre-commit command during the buffering phase and “Committed” at the end of the decision phase.

In case of an aborting transaction, the sequence of messages is the same of 2PC's scenario: the participant begins the execution in the “Active” state and advance to the “Prepared” state after sending its vote, then it receives the abort command since not all the participants have voted “Yes” and advance to the “Aborted” state after doing the abort action and updating its log file.



This protocol is not vulnerable anymore to the one node failure but it's still vulnerable to network partitions, indeed it's been demonstrated how 3PC still shows some blocking aspects in case a network partition occurs.

5 - Paxos

5.1 - Paxos Consensus Algorithm

Paxos [3] came out from the study of the general consensus problem, in which several

processes have to agree on some value even in presence of faults: in particular, assuming a number F of nodes fails at most, 2F+1 nodes are needed in order to achieve consensus. Paxos follows an egalitarian principle, stating that every node can act as a coordinator, the protocol execution begins when a node takes the role of leader and starts sending messages to other nodes (which will take the name of acceptors) in order to achieve consensus for the current transaction, if the leader fails, a new leader is elected among the remaining nodes.
The leader node intiates a ballot, performing the following phases:

- phase 1a: the leader chose a ballot number and sends it to all of the acceptors;
- phase 1b: the acceptor receiving the ballot number discards the message if it has already performed an action for that ballot number or a greater one, otherwhise it replies to the leader with a message containing the highest ballot number it has received and the largest ballot number it has sent (the latter is present only if the node has ever played the role of leader);
- phase 2a: when the leader receive back a phase 1b message from the majority of the acceptors (half of them + 1) for its ballot number, if none of the acceptors has sent any phase 2b message yet, the leader starts sending phase 2a messages to the acceptors in order to make its own value accepted, otherwhise it sends phase 2a messages containing the value associated with the maximum ballot number;
- phase 2b: the acceptor receiving the phase 2a message discards it if it has already performed an action for that ballot number or a greater one, otherwhise it accepts the just received value and replies to the leader confirming it;
- phase 3: when the leader receive back a phase 2b message from at least the majority of the acceptors, it knows that the value has been chosen and notifies it to all of them by sending a message.

In case of failure on the leader node, another one is elected (and this is again a distribuite consensus problem), also Paxos guarantee consistency even if more than one node believe to be the leader, so only one unique value will be accepted as the final one.
During the normal fault-free execution, the initial leader(the one with ballot number 0) receives a value and sends to the acceptors a phase 2a message containing that value associated with ballot number 0, then acceptors reply with a phase 2b message to the leader in order to confirm they have accepted the value and the leader sends a phase 3 message to confirm it to all the acceptors when it receives back a phase 2b message from the majority of the acceptors.

5.2 - Paxos Commit

Paxos Commit uses multiple instances of the Paxos Algorithm in order to decide if the transaction has to be committed or aborted in a fault-tolerant way, filling what 2PC cannot guarantee.

- the execution starts when some node running the transaction locally decides to commit and sends to the initial leader a begin commit message;
- the leader then send a prepare message to all of the other nodes;
- each node decide if it wants to commit or abort the transaction, nodes that want to commit enter in the “Prepared” state, while nodes that want to abort the transaction enters in the “Aborted” state after performing the abort action;
- each node runs the Paxos Algorithm and sends to the acceptors a phase 2a message with ballot number 0 and the value “Prepared” or “Aborted” based on its decision about the transaction;
- for each instance of the Paxos Algorithm, the leader receives back from the acceptors the value they have agreed about and send it back to each node;

- the transaction is committed if every instance of the Paxos Algorithm ended with “Prepared” as the accepted value, otherwhise the transaction is aborted;

6 - Conclusions

6.1 - The large-scale compromise

In orer to know what properties of the system can be exploited to achieve having ACID properties even in a large-scale distribuited environment, it's necesssary to have a look to what the CAP theorem states.
In distributed systems, there are 3 features impossible to achieve all at the same time, while designing the system's architecture, the choice is to pick 2 among:

- Consistency: all clients see the same data at the same time;
- Availability: the system continues to operate even in presence of node failures;
- Partition tolerance: the system continues to operate in spite of network failures;

In the case of distribuited data systems, partition tolerance is the one compulsory needed, so the tradeoff is between consistency and availability; in order to maintain the ACID properties, availability has to be sacrificed in favor of consistency but even by doing that there are still issues in guaranteeing isolation:

- The first issue regarding isolation is the creation of a global schedule that has to be followed by all of the nodes, the system have to pay particular attention to the conflicting operations since their possible execution orders lead to different final states of the database;
- The management of locks is not centralized so deadlock situations are more difficult to handle, for example in this situation with two transactions T and U

T	U
Write(A) at X locks A	
	write(B) at Y locks B
Read(B) at Y waits for U	
	Read(A) at X waits for T

  each of the two transactions will wait forever for the other one to release its lock;

Achieving isolation in a distribuited data system can be quite difficult but it's not impossible, however, even by doing that there is still a last issue to deal with: by sacrificing availability in favor of consistency a delay in transactions' responses is unavoidable because, in order to maintain the system consistent, there is the necessity of waiting a small amount of extra time after a transaction ends because the system has to return in a consistent state (i.e. All of the nodes/replicas have finished their local execution of the transaction) before being able to start executing a new transaction.
So, maintaining ACID properties can be possible with a lot of precautions like the ones previously discussed but at the cost of the system's performance, this is not so good since large-scale distribuited systems are often employed in contexts where quick responses are needed and there is a very huge amount of data and transactions to deal with.

After all, the true optimal solution seems to be giving up achieving ACID because high availability and high performance are often critical requirements in this type of systems, this doesn't mean all of the ACID perks will be lost, they simply will be replaced with some less restrictive versions of the same concepts:

- Basically Available: the database appears to work most of the time;

- Soft-state: nodes have not to be write-consistent and the replicas have not to be mutually consistent at the same time;
- Eventual consistency: all of the nodes will show consistency at some later point;

BASE increases the performance of operations with a more optimistic approach for data access, avoiding the locking of the accessed resources and moving the moment in which the system will have to be consistent to a second moment.

6.2 - Performance issues

In large-scale data management systems, good performance and fast responses are very important to achieve since the system has to operate in a scenario when data is accessed very frequently and even saving a small amount of time could be critical since it impacts the response time of all the executed transactions, so it's necessary to make a comparison among the protocols presented so far in order to state which of them is theoretically the best one.
The efficiency of an ACP is measured based on how the protocol behaves towards these 3 main performance issues [1]:

- Efficiency during normal processing, based on message complexity (number of messages needed to be exchanged between the nodes), log complexity (frequency at which informations are written into the log file in the stable memory in order to being available in case of failures) and time complexity;
- Resilience to failures, referred to the types of failures the protocol can tolerate and the effects of them on the involved nodes and on the system;
- Independent recovery, referred to the time required by the system for being fully operational again after a failure occurred, assuming that each node has stored locally all of the information it needs to fully recover without external communication;

In 2PC, 4 messages are exchanged between the leader node and a single participant node, so the total number of messages needed by the normal execution of 2PC is 4N (with N = number of participants).
Speaking of log complexity, each participant force-writes on its log 2 times during the normal execution, so there are a total of 2N force-writes by participants + 1 force-write the leader does before sending the prepare message to all of the participants.
In case of an aborting transaction, the number of messages is 4N - A and the number of force-writes is 2N + 1 - A (where A = number of nodes that decided to abort and won't need the second message from the leader nor will force-write on their log before performing the abort action).

2PC variants show even a better performance than 2PC does:

- Presumed Commit has the same performance of 2PC in case of an aborting transaction but optimizes the message and logging cost during the commit scenario, here the number of messages needed drops to 3N and the number of force-writes to C+2 due to the optimizations done in the decision phase;
- On the other hand, Presumed Abort has the same performance of 2PC in case of a committing transaction and is optimized in the handling of aborting transactions;

Since they have more phases and message exchanges, 3PC and Paxos Commit show an higher message complexity tran 2PC does.
With respect to the other performance issues, they have a better fault tolerance and a faster recovery, so there is the need to compare these two perks and state which of the

three protocols suits better nowadays systems.

Both efficiency during normal processing and efficiency during the recovery phase are important but the reference scenario suggests to consider the former as more important. In particular, faults exists but they are not very frequent, so the performance in absence of faults has to be taken more in consideration.
The first comparison to do is between 2PC and 3PC: being 3PC a more sophisticated version of 2PC, it has an higher message complexity so it's also more difficult to implement and maintain, plus the fact that 3PC is not totally partition-tolerant and still shows some blocking aspects when a network partition occurs, that's the reason why it was never implemented in any commercial database system (as explained in section “key applicartions” of [2], 3CP is still important in theoretical computer science since it gives a good idea of an ACP's behavior and explains the restraints in trying to solve the atomic commitment problem).
Paxos has the perk of never block the execution as long as the majority of the processes are correct but require more messages than 2PC does and this seems to be not a good give-and-take although Paxos is resilient to failures.
As explained in section 5 of [3], 2PC can be seen as a degenerate case of Paxos with a single acceptor so it's implementation cost is minimal with respect to the Paxos Commit average implementation cost, also, it was demonstrated that the performance of 2PC doesn't degenerate so much in presence of failures (see the experiment done in [4] over 2PC and its variants, which includes also a variant not covered in this paper).
Since it solves the distribuite consensus problem, the Paxos algorithm was adapted to solve the atomic commit problem too and, in fact, it solves it, but in some distribuite systems is set aside in favor of 2PC; however, Paxos finds its use in replication control and ensuring replica consistency.

Considering what has been said so far, 2PC seems to be the best solution: being the faults quite rare, they don't impact so much on the total execution time, expecially by using the optimized variants 2PC offers, so several nowadays systems like MongoDB implement 2PC as their atomic commit solution.
On the other hand, based on system requirements, the poor scalability of 2PC could make Paxos preferable, systems like Cassandra and Neo4j indeed use Paxos instead of 2PC.

An hybrid solution is not feasible since ACPs are incompatible with each other, they cannot be mixed together in the same system without the arise of conflicts.
In conclusion, there is no “ultimate best solution” that suits all the existing systems, the decision has to be taken case by case keeping in consideration both the advantages each solution offers and the needs of the system it will be implemented on.

References

1. Yousef J. Al-houmaily, George Samaras
   Two-Phase Commit;
2. Yousef J. Al-houmaily, George Samaras
   Three-Phase Commit;
3. Jim Gray, Leslie Lamport
   Consensus of Transaction Commit;
4. M.L. Liu, D. Agrawal, A. El Abbadi
   The Performance of Two-phase Commit Protocols in the Presence of Site Failures;