Distributed Systems Lab 09

Distributed commit

During the recent labs, we presented various techniques commonly employed in distributed systems for implementing supporting functionality. The following labs will in turn focus on popular core distributed algorithms. In particular, this one presents two such algorithms, 2PC and 3PC, which address the so-called distributed commit problem. We will use files contained in this package, which you should download and extract locally.

Learning section

Consider a distributed system consisting of multiple separate processes that execute **transactions**, that is, each process atomically executes some operations (you can think of transactions in relational SQL-based databases). Having a request that consists of multiple transactions (one transaction for each of the processes), the distributed system is required to either successfully process all of these transactions or none at all. In other words, the system is required to implement a **distributed commit**.

There are many real-world applications that require distributed commit. For instance, to process more data than a single machine can handle, relational databases partition tables' rows into multiple independent shards (usually, the rows are partitioned by a *partition key* which is a hash/number that can quickly calculated). Then, the database has to guarantee that an SQL transaction is committed if and only if every shard successfully commits its part.

Assumptions

In the following discussion, we assume the crash-recovery failure model (individual processes crash and recover, but there are no Byzantine failures and no network failures). Moreover, we assume that the processes of the system implement individual recovery after a failure (e.g., using local stable storage).

One process of the system is designated to be a **Transaction Manager**, **TM** for short, which coordinates the system. Clients contact *TM* when they request transactions. For the sake of simplicity, in the following description, *TM* is considered to be a special separate process designated before the system is launched. However, in general, the role of *TM* can be also dynamically assigned at runtime to one of the system processes, and every request from clients can be served by a different *TM*.

Two-phase commit (2PC)

As the name suggests, the **two-phase commit (2PC)** protocol comprises two phases. The first one, the **prepare commit phase**, begins when *TM* receives a transaction request from a client. Then *TM* sends a message to each process, asking whether it can execute its part of the transaction. If some part cannot be executed (e.g., it would violate constraints), the process replies negatively to *TM*; otherwise, it replies positively and—what is very important—it locks resources required to commit the

transaction (but does not commit it yet): if a process replies positively, it *must* be able to commit the transaction when it will be ordered to actually do so (it *must not change its mind later*).

When *TM* receives responses from all processes, and they all are positive, *TM* sends a message to each process to commit their transactions; otherwise, when there is at least one negative response, *TM* sends a message to each process to abort the transactions. The moment *TM* makes the decision, the second phase of the protocol, the **commit phase**, begins. The processes execute the action (commit/abort), acknowledging it to *TM*. The 2PC protocol ends when *TM* receives all acknowledgments and notifies the client whether the transaction has been committed or aborted. If we assume that a process always eventually recovers, *TM* can reply to the client already when it decides to commit or abort the transaction (i.e., at the beginning of the *commit phase*).

As it can be easily seen, 2PC guarantees atomicity of the transaction, because always either all or none of the individual transactions are committed. Failures of *TM* and processes can be discovered by implementing timeouts on communication between them. When a failure is detected, the system should operate as follows.

Failure of a process

When *TM* discovers during the *prepare commit phase* that some process has failed, it decides to abort the transaction and sends corresponding messages to other processes. When the failed process recovers, it contacts *TM* and learns that the transaction has been aborted.

If a process fails during the *commit phase*, when it recovers it contacts *TM* and learns whether the transaction is to be committed or aborted, and commits or aborts the transaction, respectively. *TM* may wait for the process to recover before it replies to the client (in such a case, the protocol may block here).

Failure of *TM*

When processes discover during the *prepare commit phase* that *TM* has failed, they contact each other to verify that they are indeed in the *prepare commit phase* (i.e., none of them has received a commit/abort message), and abort the transaction. When *TM* recovers, it contacts the processes and learns that the transaction has been aborted.

When *TM* fails during the *commit phase*, the processes contact each other to learn about the decision of *TM* (if it is the *commit phase*, then at least one process has received a commit/abort message), and they execute the decision.

Failure of both *TM* and a process

In the most pessimistic scenario, *TM* fails directly after it decides to commit the transaction and sends such a message to a process, which commits the transaction and then also fails. In this case, other processes contacting each other are unable to learn about the decision, as from their perspective both commit and abort decisions are possible. The only way to proceed is to wait until the process or *TM* recovers.

Although this situation also blocks, it is different from the one when *TM* waits for the recovery of a process that failed during the *commit phase*: then the decision is known and if we assume that every process eventually recovers, *TM* does not have to wait for the process before it replies to the client. Here, in contrast, the decision is unknown as long as both *TM* and the failed process are unresponsive, and thus the system cannot proceed. 2PC is thus a **blocking** commit protocol.

Three-phase commit (3PC)

The reason why 2PC blocks is that there is a state which may result in both commit and abort: only one decision is actually the correct one, but from the perspective of the responsive processes both decisions are possible. To solve this problem, the **three-phase commit (3PC)** protocol introduces an extra phase between the *prepare commit phase* and the *commit phase*: when *TM* decides whether the transaction will be committed or aborted, it sends to each process a message announcing this decision (but still not asking to commit it if it is a commit decision). Only when all processes acknowledge receptions of the decision, does *TM* transition to the *commit phase* and asks processes to commit the transaction.

This additional phase allows to survive simultaneous failures of multiple processes and *TM*, as long as a majority of the processes is up and reachable. In other words, 3PC is considered a **nonblocking** commit protocol. The details can be found in the recommended literature.

Practical aspects of xPC protocols

In theory, 2PC does not scale well because already two crashes (of *TM* and one process) can block the system. (Or one crash if *TM* is also one of the system's processes.) The word *crash* sounds abrupt, but it can be, for instance, just a deployment of a new version of the software, as the properties of crash-recovery algorithms are usually exploited to implement hassle-free redeployment of individual processes of a system. Moreover, this allows also to shut down physical machines and take them for maintenance without disrupting operations of the system. It is especially useful in large-scale deployments, as although hardware may seem to be fairly reliable, the large scale increases the risk of some hardware failure significantly.

However, although 3PC fixes the blocking issue of 2PC, it is rarely employed in real-world systems. This is due to its significant communication overhead: 3PC requires 3 round-trips with $6n$ messages for n processes in total. What is more, each round-trip involves contacting all processes, and thus the whole system proceeds as fast as its slowest process. These issues of 3PC make 2PC the preferred distributed commit protocol for real-world systems (which usually implement also additional mechanisms to minimize the risk of a failure which could block the system).

Small Assignment

Your task is to implement the lacking functionalities of *DistributedStore*, a distributed storage system based on 2PC (the two-phase commit protocol). The system stores products (the `Product` struct) and allows changing their prices by specifying a product type, for example, *increase price of all computers by 10*. The information about the prices of the products is stored in distributed shards, following some partition key. *DistributedStore* shall use 2PC to atomically perform updates on all shards. There will only be at most one transaction in progress at any given time.

Every process must respond to `ProductPriceQuery` messages and return the current price of the queried product. `None` shall be returned if the product is not in the store.

Processes of the system shall deny transactions (by voting to abort them) that would result in nonpositive prices of products (i.e., the prices shall be always greater than `0`).

In this assignment you should assume a crash-stop model—you should not store anything in stable storage or support recoveries. You also should not implement any timeouts (we accept that a crash can prevent the system from making progress).

This Small Assignment is worth **2 points**. To run the system, you should use the executor system you implemented as the first Large Assignment.

Additional Homework

If you wish to get alternative information on 2PC and 3PC, we recommend reading the *Distributed commit* chapter in the Distributed Systems book by M. van Steen and A. S. Tanenbaum. You can get a free digital copy of the 3rd edition of the book at the author's website (or even a more recent 4th edition).

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek.