**Appendix D. Two-phase commit (2PC)**

We discuss two-phase commit (2PC) here as a possible distributed transactions technique, but emphasize that it is unsuitable for distributed services. If we discuss distributed transactions during an interview, we can briefly discuss 2PC as a possibility and also discuss why it should not be used for services. This section will cover this material.

Figure D.1 illustrates a successful 2PC execution. 2PC consists of two phases (hence its name), the prepare phase and the commit phase. The coordinator first sends a prepare request to every database. (We refer to the recipients as databases, but they may also be services or other types of systems.) If every database responds successfully, the coordinator then sends a commit request to every database. If any database does not respond or responds with an error, the coordinator sends an abort request to every database.

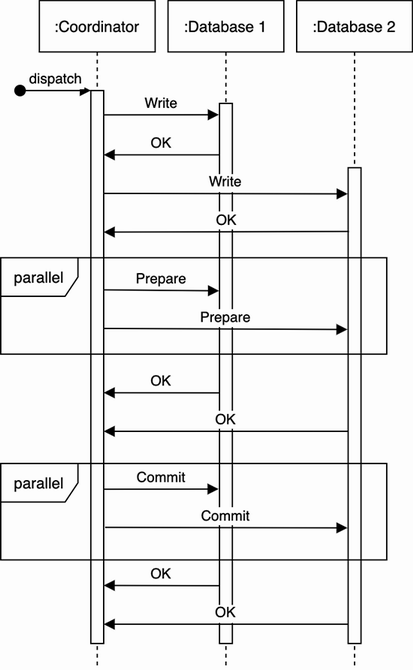


Figure D.1 A successful 2PC execution. This figure illustrates two databases, but the same phases apply to any number of databases. Figure adapted from *Designing Data-Intensive Applications* by Martin Kleppmann, 2017, O’Reilly Media.

2PC achieves consistency with a performance tradeoff from the blocking requirements. A weakness of 2PC is that the coordinator must be available throughout the process, or inconsistency may result. Figure D.2 illustrates that a coordinator crash during the commit phase may cause inconsistency, as certain databases will commit, but the rest will abort. Moreover, coordinator unavailability completely prevents any database writes from occurring.

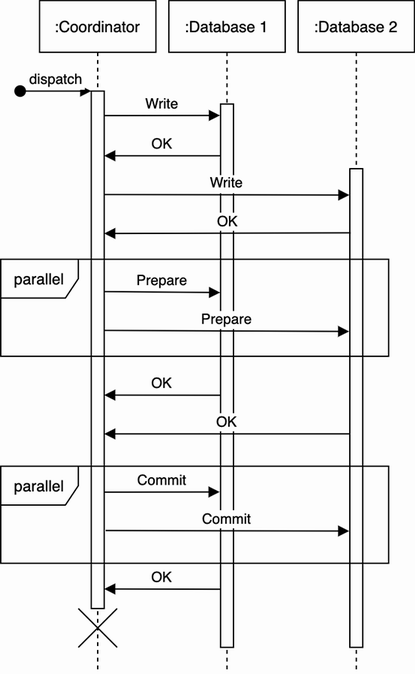


Figure D.2 A coordinator crash during the commit phase will cause inconsistency. Figure adapted from *Designing Data-Intensive Applications* by Martin Kleppmann, O’Reilly Media, 2017.

Inconsistency can be avoided by participating databases neither committing nor aborting transactions until their outcome is explicitly decided. This has the downside that those transactions may hold locks and block other transactions for a long time until the coordinator comes back.

2PC requires all databases to implement a common API to interact with the coordinator. The standard is called X/Open XA (eXtended Architecture), which is a C API that has bindings in other languages too.

2PC is generally unsuitable for services, for reasons including the following:

* The coordinator must log all transactions, so during a crash recovery it can compare its log to the databases to decide on synchronization. This imposes additional storage requirements.
* Moreover, this is unsuitable for stateless services, which may interact via HTTP, which is a stateless protocol.
* All databases must respond for a commit to occur (i.e., the commit does not occur if any database is unavailable). There is no graceful degradation. Overall, there is lower scalability, performance, and fault-tolerance.
* Crash recovery and synchronization must be done manually because the write is committed to certain databases but not others.
* The cost of development and maintenance of 2PC in every service/database involved. The protocol details, development, configuration, and deployment must be coordinated across all the teams involved in this effort.
* Many modern technologies do not support 2PC. Examples include NoSQL databases, like Cassandra and MongoDB, and message brokers, like Kafka and RabbitMQ.
* 2PC reduces availability, as all participating services must be available for commits. Other distributed transaction techniques, such as Saga, do not have this requirement.

Table D.1 briefly compares 2PC with Saga. We should avoid 2PC and prefer other techniques like Saga, Transaction Supervisor, Change Data Capture, or checkpointing for distributed transactions involving services.

Table D.1 2PC vs. Saga

| 2PC | Saga |
| --- | --- |
| XA is an open standard, but an implementation may be tied to a particular platform/vendor, which may cause lock-in. | Universal. Typically implemented by producing and consuming messages to Kafka topics. (Refer to chapter 5.) |
| Typically for immediate transactions. | Typically for long-running transactions. |
| Requires a transaction to be committed in a single process. | A transaction can be split into multiple steps. |