CS 223 Project:
Implementation of a Two-Phase Commit Protocol in Python

Andr Rsti, Danny Ibrahim

March 2021

Abstract

We describe the implementation of a two-phase commit protocol in Python. Agents communicate through a RPC-like protocol that uses JSON for serialization. Participant nodes write to a backing PostgreSQL database for both logs and data. Using logs and the two-phase commit protocol, atomicity and consistency of writes to the database is ensured in a network of non-byzantine, but possibly failing nodes.

Contents

1 Usage

We first describe the usage of our nodes. Before running coordinator and participant nodes, some PostgreSQL databases may be started. Further preqrequisites are listed below.

1.1 Prerequisites

1. Python $\geq$ 3.7
2. Python package `psycopg2`
3. Multiple running PostgreSQL servers *(Optional; if not provided, the script will spin up its own database clusters and destroy them again upon exit)*

On MacOS with Homebrew, the following commands can be used to install all required dependencies:

```
brew install postgres
brew install python3
pip3 install psycopg2
```

Note that PostgreSQL databases that store the participant's data must have prepared transactions enabled. This is achieved by setting the configuration option `max_prepared_transactions` to a non-zero value. The following command will start a properly configured database on port 9991:

```
env PGPORT=9991
pg_ctl start -D /path/to/db/root
-o "-c max_prepared_transactions=99"
```

1.2 Starting agents

The `main.py` script is the entry point for coordinator and participant nodes. For any agent, the `--host` argument must be supplied to specify what hostname and port that agent should listen for messages on. For the coordinator node, all participant nodes must be listed using multiple `--participant` arguments. Participant nodes must be given the hostname and port of the coordinator node using the `--coordinator` argument. Participant nodes must further be assigned a consecutive index, starting from 0, with the `--node-id` argument. The `client.py` script allows sending client requests to a coordinator.

If the coordinator and participants are not given any database info, they will spin up their own database clusters, one for logs and one for data (only participants), in a temporary directory on a random free port. These database instances will be destroyed again upon shutdown. If a persistent database should be used – e.g. to test dying and recovering nodes – a database server can be set up and started ahead of time; its connection data may then be given as a connection URI using the `--data-db` and `--log-db` arguments. It is assumed that the table `log` will be used for log storage. If that table is not present, they will be created automatically.

Coordinator

```
python main.py
  --host localhost:8880
  --participant localhost:8881
  --participant localhost:8882
  --log-db postgresql://:9990
  --batch-size 10 --timeout 3
```

Participants

```
python main.py --node-id 0
  --host localhost:8881
  --log-db postgresql://:9991
  --data-db postgresql://:9992
  --coordinator localhost:8880

python main.py --node-id 1
  --host localhost:8882
  --log-db postgresql://:9993
  --data-db postgresql://:9994
  --coordinator localhost:8880
```

1.3 Using the Client

The client provides an interface for sending messages to the coordinator. It provides two modes: Demo mode and interactive mode.

1.3.1 Demo Mode

In demo mode, the client automatically sends row after row from the demonstration data set to the coordinator, waiting one second between each insert. For this, the demo database has to be set up and contain the tables tables `thermometerobservation`, `wemoobservation`, and `wifiapobservation`. Any of these three tables can be used as the demo data set.

To start the client in demo mode, you must specify the demo database connection URI using `--data-db`, the number of participant nodes you started with `--n-nodes`, and which demo table you wish to use using `--demo`:

```
python3 ./client.py
  --coordinator localhost:8881
  --demo thermometerobservation
  --data-db postgresql://:9999/observations
  --n-nodes 3
```

1.3.2 Interactive Mode

Interactive mode allows sending arbitrary SQL queries to any of the participant nodes. It continually asks for new queries, along with a node ID, until told to quit. To start in interactive mode, only the coordinator has to be given:

```
python3 ./client.py
  --coordinator localhost:8881
```

1.4 Reference of Command-Line Options

For a full reference of the command-line options of the scripts, call them with the `--help` flag.

2 Implementation Details

2.1 Communication

The implemented communication protocol follows an event-based approach. On every node, a server task continuously waits for incoming connections. Upon any incoming connection with valid data, a previously registered callback method is executed depending on the received message. This is in principle similar to a remote procedure call (RPC) protocol.

Data is serialized using JSON and sent over the network as an UTF-8-encoded stream of bytes. Multiple messages may be sent over the same communication channel; in that case, messages are separated using a single zero byte (similarly to C-style null-terminated strings).

Messages contain a `kind` and a `data` field. The `kind` determines what callback is invoked on the server upon receipt of the message. The `data` field is passed to the callback method unaltered. Upon completion of the callback method, its return value is sent back to the calling client through the same communication channel, again JSON-encoded.

2.2 Log implementation

On each node, a log of transactions is written to a database. Transactions are identified by an unique ID, which is increased upon the beginning of any new transaction. The transaction logs are a simple map of transaction ID to state. This map is stored in a PostgreSQL database. If no database info is given to an agent upon startup, it will spin up its own database cluster in a temporary folder and use it as its log. Note only the coordinator and participants create a log; the client does not need to do so.

For ease of implementation, upon starting (and recovery) of the node we simply read the entire log database into a map data structure on the agent. When a forced write occurs, we write the entirety of this in-memory data structure back to the database. For longer logs, this is inefficient; but suffices for this prototype.

2.3 Two-phase commit protocol implementation

The two-phase commit protocol is intended to ensure atomicity and consistency among all running nodes in a networked system of databases. Any node may fail at any time, but we assume that all nodes are benign and follow the protocol as specified, i.e. there are no malicious/byzantine actions taken.

In our implementation, all nodes communicate through the protocol as described in section 2.1. There are three types of nodes: (possibly multiple) client(s), a coordinator, and participants. The client is the user of our system; it sends database requests, such as `INSERT`s, to the coordinator. The coordinator forwards such requests to the appropriate participant. Participants store the data. There are N participants, and each of them is identified by an integer $0, \ldots, N-1$.

2.3.1 Transaction States

The coordinator and participant nodes keep track of the state of each transaction that has not yet completed:

`BEGUN` The transaction has begun, but it is not yet complete. In this state, more SQL statements can be appended to the transaction.

`PREPARED` The transaction is completed, but it has not yet been decided whether all nodes will commit or abort it. Participants are in this state after receiving a `PREPARE` message and having performed the necessary prepare actions; the coordinator writes this state to its logs before sending those messages.

`COMMITTED` The coordinator has received affirmative prepare messages from all participants. At this point, it is decided that all participants will have to commit.

`ABORTED` The coordinator has received at least one `ABORT` message from a participant, or has not received a response at all before expiration of the timeout from at least one of the participants. All nodes will abort the transaction.

`DONE` All nodes have acknowledged that they have executed the `COMMIT` or `ABORT` for this transaction. It is safe to remove the transaction from the log, since no participant will ask about it again.

2.3.2 Coordinator

The coordinator is the central node in our network; it is either the recipient or sender of any message in the system; other nodes do not directly communicate with each other. The coordinator handles the following messages:

`EXECUTE` This is the only message that a client can send to the coordinator. It may contain any SQL statement to execute on the distributed database. It is the coordinator's task to forward such requests to the appropriate participant.

The coordinator keeps track of the number of statements received in order to batch statements into transactions. (The `--batch-size` arguments decides how many `EXECUTE`s are batched into one transaction.)

If this is the first `EXECUTE` in this transaction, a new transaction is implicitly begun. To do so, the current transaction ID is increased by one, and a new row is inserted into the transaction log, with status `BEGUN`. When a new transaction is begun, the coordinator ensures that the previous transaction is `DONE` or `COMMITTED`/`ABORTED`; otherwise it blocks and waits for the previous transaction to complete.

The EXECUTE is then forwarded to the appropriate participant node, which will (tentatively) insert the data into their database.

If the number of issued EXECUTEs is equal to the batch size, the transaction completion process is started, as described in the next paragraph.

Transaction Completion Process When the coordinator decides to try to commit a transaction, the transaction log is first updated to the new status PREPARED. Following this, a PREPARE message along with the current transaction ID is sent to *all* participant nodes.

After this, a busy loop will wait for the everyone_prepared event; this event is set by the callback for PREPARE messages and indicates the receipt of PREPARE messages from all participants. Once all participants replied, or after a timeout occurs while waiting for some participant's reply, a decision is made to either commit or abort the transaction. A commit occurs if and only if a PREPARE COMMIT message was received by all participants. Before continuing, the decision to commit or abort is (forcibly) written to the log.

The coordinator then sends either a COMMIT or ABORT message to all participants. This completes the transaction completion process. The event handler for the DONE messages may later remove the transaction from the log completely.

PREPARE PREPARE messages are received by participant nodes after the coordinator started the transaction completion process. A participant may either suggest to COMMIT or ABORT. Upon receipt of such a message, we set a flag in a local data structure, indicating that a PREPARE message has been received by the sender node. If we observe that this data structure indicates that all participants have replied with a PREPARE, we fire the everyone_prepared event.

Note that the data structure keeping track of the prepared nodes is in-memory only. If the coordinator crashes, this data structure will be recreated by re-sending the PREPARE messages to nodes.

DONE Receiving this message from a participant indicates that the commit/abort has been successfully executed by that node. We store this information in a local data structure. When, upon receipt of the "last" DONE message, we observe that all nodes have acknowledged completion of a particular transaction, we can safely remove that transaction from the log.

Recovery Upon startup, the coordinator reads the log from the database into its memory. For each transaction, it (re-)issues the protocol flow as if it had just received the message(s) that lead to that transaction's logged state: If a transaction is stored as PREPARED, it sends PREPARE messages to all participants. If a transaction's state is COMMITTED or ABORTED, it sends the respective COMMIT or ABORT messages to participants. Note that this recovery process should only occur if the coordinator node has died in the middle of a transaction; transactions that complete normally (DONE message received by all participants) are removed from the log completely. Note further that some of the participants may receive messages twice; the coordinator may die between writing the logs and sending messages, or it may die after having sent the messages. Participants therefore simply ignore messages that they have already handled.

2.3.3 Participant

Participants react to the following incoming messages:

EXECUTE A new row of data is stored in the local database. This write to the database is tentative; it may not be committed yet. If the received transaction ID with the EXECUTE message is not equal to the current transaction ID in the node, a new transaction is implicitly started. Before doing so, it is ensured that the previous transaction completed regularly. (A sane coordinator will ensure this.)

PREPARE Check whether the current transaction ID is equal to the transaction ID in the prepare message. If it is, decide to commit this transaction and do the necessary prepare work, using PostgreSQL's PREPARE TRANSACTION statement. Update the transaction's state to PREPARED, and send back a PREPARE COMMIT message.

If, however, the transaction ID appears to be old, check whether a commit/abort decision is stored in the local log, and return this information to the coordinator.

COMMIT, ABORT Commit/abort the currently active transaction if the received transaction ID is equal to the message's transaction ID. Reply to the coordinator with a DONE message.

Recovery As with the coordinator node, participant nodes enter recovery mode when they are started. They iterate over all the log entries read from the database. For each transaction, a participant follows the protocol flow as if it had just received the message that led to the sate stored in the transaction log.

2.3.4 Client

Since we implement only EXECUTEs, but no querying of the database, the client only needs to send messages, but does not implement handlers for the receipt of any messages. The client only sends EXECUTE messages to the coordinator. The handling of these messages is described in section 2.3.2.