Distributed Transactions

Diya Joy and Peter Gunarso
Transaction Models
ACID vs. BASE

- ACID: Atomicity, Consistency, Isolation, Durability
  - Provides a consistent system
  - Sacrifices high performance for ease of programming
- BASE: Basically-Available, Soft-state, Eventually consistent
  - Provides high availability
  - Ensuring consistency is up to application programmers

CAP theorem: It is impossible to achieve both consistency and availability in a partition tolerant distributed system
ACID vs. BASE

(a) The ACID approach.

```plaintext
1 // ACID transfer transaction
2 begin transaction
3  Select bal into @bal from accnts where id = sndr
4  if (@bal >= amt)
5     Update accnts set bal -= amt where id = sndr
6  // To enforce atomicity, we use queues to communicate
7  // between partitions
8  Queue message(sndr, rcvr, amt) for partition(accnts, rcvr)
9 end local--transaction
10 // Background thread to transfer messages to other partitions
11 begin transaction // distributed transaction to transfer queued msgs
12   <transfer messages to rcvr>
13 end transaction
14 // A background thread at each partition processes
15 // the received messages
16 begin local--transaction
17   Dequeue message(sndr, rcvr, amt)
18   Select id into @id from accnts where id = rcvr
19   if (@id ≠ 0) // if rcvr’s account exists in database
20     Update accnts set bal += amt where id = rcvr
21 else // rollback by sending the amt back to the original sender
22     Queue message(rcvr, sndr, amt) for partition(accnts, sndr)
23 end local--transaction
24 // total--balance using the BASE approach
25 // The following two lines are needed to ensure correctness of
26 // the total--balance ACID transaction
27 <notify all partitions to stop accepting new transfers>
28 <wait for existing transfers to complete>
29 begin transaction
30   Select sum(bal) from accnts
31 end transaction
32 <notify all partitions to resume accepting new transfers>
```

(b) The BASE approach.
When applications need higher performance than ACID provides, it's often because of the needs of just a few transactions.
Salt: Best of ACID and BASE
How can we use both ACID and BASE in the same application?

- Use BASE transactions for certain performance-critical transactions
- Keep ACID transaction for the majority
New Terms

- **ACID transactions**: The standard ACID transactions
- **BASE transactions**: Retain atomicity for transactions, but allow isolation to be specified at finer granularity (salt isolation)
- **Alkaline sub-transactions**: BASE transactions are composed of a sequence of alkaline sub-transactions

**Salt isolation**: A property that allows BASE transactions to achieve high concurrency by observing each other’s internal states
- Does not affect the isolation guarantees of ACID transactions.
Types of Locks

- **ACID locks**: Writes conflict with reads and writes, reads do not conflict with each other.
- **Alkaline locks**: Conflict with all other transactions except read-read, long-term locks only held until transaction completes.
- **Saline locks**: Isolate ACID transactions from BASE transactions, conflict with ACID for non-read but never conflict with other saline/alkaline locks.
ACID Locks

(a) BASE waits until ACID commits.
Alkaline/Saline Locks

(b) $\text{BASE}_2$ waits only for $\text{alkaline}_1$…
Saline Locks

(c) ... but ACID must wait all of BASE out.
Problem: Indirect Dirty Reads

Fig. 4: ACID\textsubscript{1} indirectly reads the uncommitted value of x.
Discussion

Indirect dirty reads are possible through naive usage of our locking mechanism.

- What could we have done to prevent the indirect dirty read in the previous example?

```
BASE_1
Lock on x
w(x) ... (other operations) ... commit

BASE_2
Lock on x
Lock on y
r(x) w(y) commit

ACID_1
Lock on y
r(y) commit
```

- ACID lock
- alkaline lock
- saline lock
- waiting to acquire lock
Solution: Saline Locks

Fig. 5: How Salt prevents indirect dirty reads.
Locking Rules: Saline Locks

**Read-after-write across transactions:** A BASE transaction $B_1$ that reads a value $x$, which has been written by another BASE transaction $B_2$, cannot release its saline lock on $x$ until $B_2$ has released its own saline lock on $x$.

**Write-after-read within a transaction:** An operation that writes a value $x$ cannot release its saline lock on $x$ until all previous read operations within the same BASE transaction have released their saline locks on their respective objects.
Salt Performance: Latency

Fig. 7: Performance of ACID and Salt for TPC-C.

Fig. 8: Performance of ACID and Salt for Fusion Ticket.
Salt Performance: Throughput

Fig. 11: Effect of contention ratio on throughput.

Fig. 13: Effect of read-write ratio on throughput.
Real-World Database Transactions

ACID (SQL databases):

- MySQL
- SQLite
- Microsoft SQL
- Oracle

BASE (NoSQL databases):

- MongoDB
- Cassandra
- Redis
Discussion Questions

Salt has demonstrated much lower latency than ACID and much higher throughput in certain cases.

- What other metrics/features of Salt would you want to know about if choosing to use it?
- Why isn’t Salt more widely adopted?
- Are there situations where Salt performance would not be as high as expected?

https://tinyurl.com/cse550transactions
Fig. 2: A Salt implementation of the simple banking application

```plaintext
// BASE transaction: transfer
begin BASE transaction
try
begin alkaline-subtransaction
Select bal into @bal from accnts where id = sndr
if (@bal >= amt)
  Update accnts set bal -= amt where id = sndr
// To enforce atomicity, we use queues to communicate between partitions
Queue message(sndr, rcvr, amt) for partition(accnts, rcvr)
end alkaline-subtransaction
catch (Exception e) return // do nothing
if (@bal < amt) return // constraint violation
try
begin alkaline-subtransaction
Update accnts set bal += amt where id = rcvr
end alkaline-subtransaction
catch (Exception e) // rollback if rcvr not found or timeout occurs
begin alkaline-subtransaction
Update accnts set bal += amt where id = sndr
end alkaline-subtransaction
end BASE transaction

// ACID transaction: total-balance (unmodified)
begin transaction
Select sum(bal) from accnts
commit
```

(b) The BASE approach.
Distributed Transactions
Quick Recap

- Distributed vs Centralized Systems
  - Reliability
- How can we provide atomic transactions in a distributed system?
  - Atomic Commitment Protocols
Goals of an ACP

Every operation ends in either *Commit*, or *Abort*

1. All processes that reach a decision reach the same one
2. Decisions cannot be reversed
3. *Commit* can only be reached if all processes agree
4. If there are no failures and all processes agree, then there will be a *commit*
5. The protocol will always advance when it can (no blocking)
Assumptions

- No data replication
- Fail-stop
- Dropped Messages
2PC - Rules

1. Phase 1 - Vote
   a. Coordinator sends out a VOTE-REQ to all participants
   b. Participants vote YES or NO to the coordinator’s suggestion

2. Phase 2 - Commit
   a. If all participants vote YES, coordinator sends COMMIT to participants
   b. Participants receive COMMIT and act accordingly
Should I do another presentation for 550?
VOTE-REQ:
Do another 550 Presentation?
2PC

YES: Let's do it!

Coordinator

Diagram showing a 2PC protocol with one coordinator and three participants.
We're doing another 550 presentation!
2PC Blocking

1. Waiting on Messages
   ○ Network Failure

2. Recovery
   ○ Node Failure
2PC - Timeouts

- 2PC runs in lock-step
  - Nodes have to wait for messages from other nodes
  - Can assume that each node knows how to contact with their peers
- On a timeout, nodes try to find out what the coordinator’s decision was from peers
2PC - Timeouts

🤔 Should I do another presentation for 550?
2PC - Timeouts

Coordinator

VOTE-REQ:
Do another 550 Presentation?

A
B
C
2PC - Timeouts

Coordinator

YES:
Let’s do it!

A

B

C
2PC - Timeouts

Coordinator

COMMIT:
We’re doing another 550 presentation!
DECISION REQ: What did I miss?
2PC - Timeouts

Coordinator

COMMIT:
We’re doing another 550 presentation!

A → B → C
2PC - Timeouts

Coordinator

COMMIT:
We’re doing another 550 presentation!
2PC - Timeouts

DECISION REQ:
What did I miss?

Coordinator

A → B

A → C
2PC - Timeouts

Coordinator

A

B

C

¯\_(ツ)_/¯

¯\_(ツ)_/¯

¯\_(ツ)_/¯
2PC - Recovery

- If a node fails, it will stop
  - When it turns on, what happens?
  - 2PC requires all nodes to participate - can only be at most one step behind

- Nodes keep a log of everything they’ve seen
  - If they use the log to restore their state, we’re looking at the same situation as a timeout!
2PC - Reflection

- **Pros**
  - Provides Consistency!
  - “Resilient” to “failures”

- **Cons**
  - Susceptible to blocking

- **Variants**
  - Decentralized 2PC
    - Everyone broadcasts to everyone all the time
    - More messages, less time
  - Linear 2PC
    - Pass along decisions like a fire line
    - Less Messages, more time
3PC

- 2PC has a lot of blocking 😞
  - Can we do better?

- 2 Flavors of 3 Phase Commit
  - Non Blocking - assume no network failures
  - Some Blocking - assume network failures can happen
3PC - Some Blocking

0. Leader Election

1. Voting Phase
   a. Coordinator sends VOTE-REQ to all participants
   b. Participants vote YES or NO, send vote back to coordinator

2. Pre-Commit Phase
   a. Coordinator collects all votes, if there is any NO vote, coordinator sends ABORT. Otherwise sends PRE-COMMIT
   b. Participants receive PRE-COMMIT, respond with an ACK

3. Commit Phase
   a. Coordinator waits for a majority of ACKs, and then sends COMMIT
   b. Participant receive COMMIT, update accordingly
3PC High Level Idea

- Extra phase adds to shared knowledge
- PRE-COMMIT phase allows for recovery during the COMMIT phase
  - When a participant wants to COMMIT, it knows all other participants agreed to the PRE-COMMIT
3PC - Reflection

- **Pros**
  - Also provides consistency
  - Resilient to failures
  - Blocks less!

- **Cons**
  - A lot more overhead
    - 3RTTs
    - 6n messages
    - Is the time saved from less blocking worth the extra overhead?
In most practical applications, the circumstances under which 2PC causes blocking are sufficiently rare that blocking is usually not considered a big problem. Consequently, almost all systems we know of that employ atomic commitment protocols use some version of 2PC.⁶
Discussion

1. We’ve talked a lot about fail-stop semantics and why they don’t apply generally - what are cases where it might be okay to look for fail-stop semantics?
   ○ How deep should protocol designers try to think about failures?

2. 2PC and 3PC are both ways to build up a consistent, distributed log. Paxos is another way to build up a consistent, distributed log.
   ○ How do these technologies compare to each other? Is one strictly better than the rest?

3. Variants of 2PC are used by popular DBMS’s today, and there hasn’t been much success in moving away from it. Why do you think 2PC is still used, and do you think it will be replaced? Do you think it should be replaced?

https://tinyurl.com/cse550transactions