Transactions

Chris Rossbach and Calvin Lin

cs380p

Outline

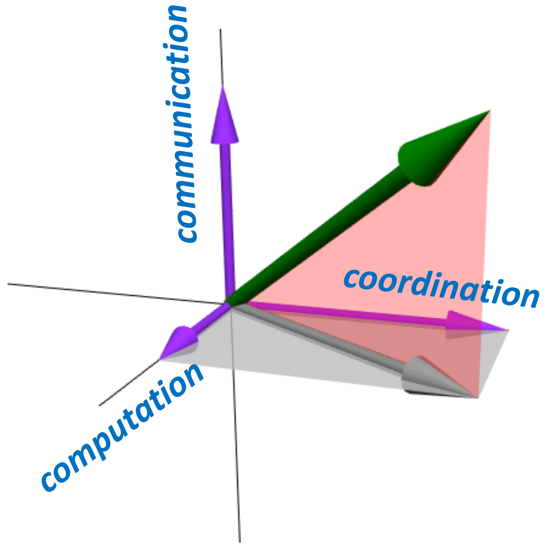
Background

Transactions

Transactions

3 Programming Model Dimensions:

- How to specify computation
- How to specify communication
- How to specify coordination/control transfer



Threads, Futures, Events etc.

- *Mostly about how to express control*

Transactions

- *Mostly about how to deal with shared state (coordination)*

Transactions

Core issue: multiple updates

Canonical examples:

```
move(file, old-dir, new-dir)
{
    delete(file, old-dir)
    add(file, new-dir)
}
```

```
create(file, dir)
{
    alloc-disk(file, header, data)
    write(header)
    add (file, dir)
}
```

Problem: crash in the middle

- Modified data in memory/caches
- Even if in-memory data is durable, multiple disk updates

Problems: Unreliability, Conflicts

Want reliable update of two resources (e.g. two disks, machines…)

- Move file from A to B
- Create file (update free list, inode, data block)
- Bank transfer (move $100 from my account to VISA account)
- Move directory from server A to B

Machines can crash, messages can be lost, or conflict

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

No.
Not even if all messages get through!

General’s paradox

Two generals on separate mountains

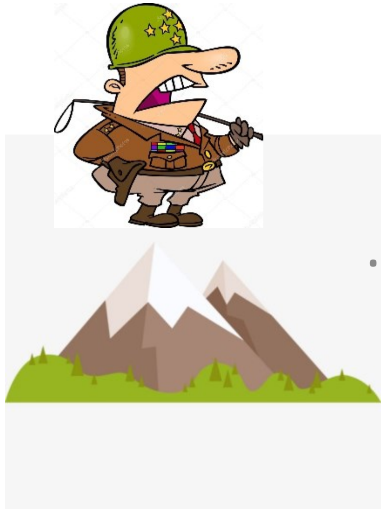
Can only communicate via messengers

Messengers can get lost or captured

Need to coordinate attack

attack at same time good, different times bad!

- Even if all messages delivered, can’t assume– maybe some message didn’t get through.
- No solution: one of the few CS impossibility results.



General A → General B: let’s attack at dawn

General B → General A: OK, dawn.

General A → General B: Check. Dawn it is.

General B → General A: Alright already—dawn.

…

Transactions can help

(but can’t solve it)

Solves weaker problem:

  - 2 things will either happen or not
  - not necessarily at the same time

Core idea: one entity has power to say yes/no for all

  - Local txn: one update (TxEND) irrevocably triggers several
  - Distributed transactions
    - 2 phase commit
    - One machine has final say for all machines
    - Other machines bound to comply

What is the role of synchronization here?

ACID Semantics

Atomic – all update

Consistent – system es

Isolated – no visibil

Durable – once dor

Are subsets ever ap

- When would ACI b
- ACD?
- Isolation only?

What are they?

- A
- C
- I
- D

```
begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;
```

Transactional Programming Model

```
begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;
```

What has changed from previous programming models?

Transactions: Implementation

Key idea: turn multiple updates into a single one

Many implementation Techniques

- Two-phase locking
- Timestamp ordering
- Optimistic Concurrency Control
- Journaling
- 2,3-phase commit
- Speculation-rollback
- Single global lock
- Compensating transactions

Key problems:

- output commit
- synchronization

Implementing Transactions

```
BEGIN_TXN();
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {
    LOCK(single-global-lock);
}
```

```
COMMIT_TXN() {
   UNLOCK(single-global-lock);
}
```

Pros/Cons?

Two-phase locking

Phase 1: only acquire locks in order

Phase 2: unlock at commit

avoids deadlock

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
BEGIN_TXN();

x = x + 1
y = y - 1

COMMIT_TXN();
```

A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

B commits changes that depend on A's updates!

Pros/Cons?

What happens on failures?

Two-phase commit

N participants agree or don’t (atomicity)

Phase 1: everyone “prepares”

Phase 2: Master decides and tells everyone to actually commit

2PC: Phase 1

1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Example—distributed FS move(foo, server1:/bar, server2:/baz):
REQ for server1: delete foo from /bar
REQ for server2: add foo to /baz

Failure case:	Success case:
server1 logs rm /bar/foo, VOTE_COMMIT	server1 logs rm /bar/foo, VOTE_COMMIT
server1 sends VOTE_COMMIT	server1 sends VOTE_COMMIT
server2 decides permission problem	server2 writes add foo to /baz/
server2 writes/sends VOTE_ABORT	server2 writes/sends VOTE_COMMIT

2PC: Phase 2

- Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Participants receive decision, write GLOBAL_* to log

2PC corner cases

Phase 1

1. Coordinator sends REQUEST to all
2. X Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2

- Y Case 1: receive VOTE_ABORT or timeout
    - Write GLOBAL_ABORT to log
    - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
    - W Write GLOBAL_COMMIT to log
    - send GLOBAL_COMMIT to participants
- Z Participants recv, write GLOBAL_* to log

- What if participant crashes at X?
- Coordinator crashes at Y?
- Participant crashes at Z?
- Coordinator crashes at W?

2PC limitation(s)

Coordinator crashes at W, never wakes up

All nodes block forever!

Can participants ask each other what happened?

2PC: always has risk of indefinite blocking

Solution: (yes) 3 phase commit!

- Reliable replacement of crashed “leader”
- 2PC often good enough in practice

Nested Transactions

- Composition of transactions
  - E.g. interact with multiple organizations, each supporting txns
  - Travel agency: canonical example
- Nesting: view transaction as collection of:
  - actions on unprotected objects
  - protected actions that my be undone or redone
  - real actions that may be deferred but not undone
  - nested transactions that may be undone
- Nested transaction may return compensating transaction
- Parent includes compensating transaction in log of parent transaction
- Invoke compensating transactions from log if parent txn aborted
- Consistent, atomic, durable, but not necessarily isolated

Concluding Remarks

- Transactions: a great abstraction
- Solve reliability and concurrency problems
- Transactional Memory: an implementation
  - Solves only concurrency problems
  - Implementable in many ways (HW, SW, hybrid,…)