LTAT.06.007 Distributed Systems

Lecture 10 – Replica consistency

Huber Flores, PhD
ASSOCIATE PROFESSOR

Tartu, Estonia, Delta building (auditorium 1021) 11/04/2022
Recap

- **Consistency**
  - Metrics (Numerical deviation, Order of operations, etc.)
  - CONIT

- **Consistency models**
  - Data-centric
  - Client-centric


diagram:

- User writes `Write(x, 1)`
- Write becomes visible
- Time `t`

Done
Think about

Back-up nodes

Online book shop

Server 1: Service payment
Server 2: Service payment
Server N: Service payment

Charge
Success

<5, B> g ← g + 45
<8, A> g ← g + 50
<9, A> p ← p + 78
<10, A> d ← d + 558
<5, B> g ← g + 45
<6, B> p ← p + 70
<7, B> d ← d + 412
<8, A> g ← g + 50
<9, A> p ← p + 78
<10, A> d ← d + 558
Think about

Back-up nodes

Either operations succeed or fail.
Agenda

• **Goal:** To study the (data) properties that need to be guaranteed for replicas to be fully reliable

• **Content:**
  - Replica consistency
  - ACID operations (Transactions)
  - Two commit phase protocol (2PC)

**After this lecture, you should be able to:**

• To explain the properties that make transactions consistent in a distributed system

• To explain the commit (abort) process of transactions from multiple replicas.
Replica consistency

Consistency model

- **Replica consistency** = Consistent copies of the same data in every replica
- Degree in which the data becomes visible to every node in the system (same order and value)
- Consistency depends on the context/application, e.g., weak vs sequential

Transactions

- Not enough agreeing about the (order) operations over a data store
- Operations must ensure to guarantee specific properties
  - Databases
- **Transaction** = A group of operations that modify some data has exclusive access to it and all operations **complete successfully**, or **none does**
ACID

Properties

- **(A)tomicity** guarantees that partial failures are not possible
  - Committed successfully
  - Rolled back (as if they never happened)

- **(C)onsistency** guarantees application-level invariants to be preserved
  - “C” in ACID has nothing to do with consistency models – “Correctness”

- **(I)solation** guarantees that the concurrent execution of transaction does not cause any race condition

- **(D)urability** guarantees that once the data store commit the transactions, the changes persisted

Replication

- Every replica should be consistent with other replicas and ensure ACID
  - Consistent = is the same state? (when exactly?)
  - Consistent = read operations return the same result?
ACID Properties

• **(A)**tomicity guarantees that partial failures are not possible
  - Committed successfully
  - Rolled back (as if they never happened)

• **(C)**onsistency guarantees application-level invariants to be preserved
  - “C” in ACID has nothing to do with consistency models

• **(I)**solation guarantees that the concurrent execution of transaction does not cause any race condition

• **(D)**urability guarantees that once the data store commit the transactions, the changes persisted

Replication

• Every replica should be consistent with other replicas and ensure ACID
  - Consistent = is the same state? (when exactly?)
  - Consistent = read operations return the same result?
Isolation

Essence

A set of concurrently running transactions that access the same data can run into all sorts of race conditions

- A **dirty-write** happens when a transaction overwrites the value written by another transaction that has not been committed yet.

- A **dirty-read** happens when a transaction observes a write from a transaction that has not completed yet

- A **fuzzy-read** happens when a transaction reads an object’s value twice, but sees a different value in each read because a committed transaction updated the value between the two reads

- A **phantom-read** happens when a transaction reads a set of objects matching a specific condition, while another transaction adds, updates, or deletes an object matching the same condition.
Isolation

Essence

• An isolation level protects against one or more types of race conditions
• The stronger the isolation level is, the more protection it offers against race conditions, but the less performant it is
• **Serializability** is the only isolation level that guards against all possible race conditions.

Caveat

• Serializability is slow as it requires coordination, which creates *contention* in the system
• Isolation level may not be required (but check)
• Jepsen ([http://jepsen.io/consistency](http://jepsen.io/consistency)) provides a good formal reference of the existing isolation levels – although most vendors also document their product isolation level provided
Isolation

Strong isolation

How to achieve Serializability?

Pessimistic (two-phase locking - 2PL) or an optimistic concurrency (multi-version concurrency control - MVCC) control mechanism.
## Atomicity

### Essence

- Transactions either succeed and their changes are committed, or transactions fail without any side effect on the data store.

- If a transaction updates data on multiple nodes, this implies:
  - Either all nodes must commit, or all must abort.
  - If any node crashes, all must abort.

- Ensuring this is the **atomic commitment** problem (sounds similar to consensus?)
Distributed consensus: when can it be reached

Formal requirements for consensus

- Processes produce the same output value
- Every output value must be valid
- Every process must eventually provide output
## Atomic commit versus consensus

<table>
<thead>
<tr>
<th>Consensus</th>
<th>Atomic commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>One or more nodes propose a value</td>
<td>Every node votes whether to commit or abort</td>
</tr>
<tr>
<td>Any one of the proposed values is decided</td>
<td>Must commit if all nodes vote to commit; must abort if &gt;=1 nodes vote to abort</td>
</tr>
<tr>
<td>Crashed nodes can be tolerated as long as there is a quorum working in place</td>
<td>Must abort if a participating node crashes</td>
</tr>
</tbody>
</table>
Distributed commit protocols
Distributed commit protocols

Problem

Have an operation being performed by each member of a process group, or none at all.

- **Reliable multicasting**: a message is to be delivered to all recipients.
- **Distributed transaction**: each local transaction must succeed.
Two-phase commit protocol (2PC)

**Essence**

The client who initiated the computation acts as coordinator; processes required to commit are the participants.

- **Phase 1a**: Coordinator sends VOTE-REQUEST to participants (also called a pre-write)
- **Phase 1b**: When participant receives VOTE-REQUEST it returns either VOTE-COMMIT or VOTE-ABORT to coordinator. If it sends VOTE-ABORT, it aborts its local computation
- **Phase 2a**: Coordinator collects all votes; if all are VOTE-COMMIT, it sends GLOBAL-COMMIT to all participants, otherwise it sends GLOBAL-ABORT
- **Phase 2b**: Each participant waits for GLOBAL-COMMIT or GLOBAL-ABORT and handles accordingly.
Two-phase commit protocol (2PC)

Client

Coordinator

A

B

Begin $T_1$

...typical transaction execution...
Two-phase commit protocol (2PC)

Client

Begin $T_1$

...typical transaction execution...

Coordinator

Commit $T_1$

A

B

Phase 1

Commit $T_1$

prepare

ok

ok

Commit or abort?

Commit

Commit
Two-phase commit protocol (2PC)

- **Phase 1**: Begin T₁
- **Phase 2**: commit or abort?

...typical transaction execution...

- **Commit T₁**
- **Prepare**
- **ok**

Commit or abort?
2PC - Finite state machines

(a)

Coordinator

(b)

Participant
Two-phase commit protocol (2PC)

**Essence**

The client who initiated the computation acts as **coordinator**; processes required to commit are the **participants**.

- **Phase 1a**: Coordinator sends VOTE-REQUEST to participants (also called a pre-write)
- **Phase 1b**: When participant receives VOTE-REQUEST it returns either VOTE-COMMIT or VOTE-ABORT to coordinator. If it sends VOTE-ABORT, it aborts its local computation
- **Phase 2a**: Coordinator collects all votes; if all are VOTE-COMMIT, it sends GLOBAL-COMMIT to all participants, otherwise it sends GLOBAL-ABORT
- **Phase 2b**: Each participant waits for GLOBAL-COMMIT or GLOBAL-ABORT and handles accordingly.
2PC – Failing participant

Analysis: participant crashes in state S, and recovers to S

- **INIT**: No problem: participant was unaware of protocol

- **READY**: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make) log the coordinator’s decision

- **ABORT**: Merely make entry into abort state idempotent, e.g., removing the workspace of results

- **COMMIT**: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.
What if the coordinator crashes?

• Coordinator writes its decision to disk

• When it recovers (if recovers), read decision from disk and send it to the replicas (or abort if no decision was made before crash)

• **Problem:** If the coordinator crashes after prepare, but before broadcasting decision, no other nodes do not know it has decided.

• Replicas participating in transaction cannot commit or abort after responding “ok” to the **prepare** request (otherwise atomicity may be violated)

• Algorithm is block until coordinator recovers
2PC – Failing participant

Alternative

When a recovery is needed to READY state, check state of other participants → no need to log coordinator’s decision

Recovering participant P contacts another participant Q

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Result

If all participants are in the READY state, the protocol blocks. Apparently, the coordinator is failing. Note: The protocol prescribes that we need the decision from the coordinator.
Coordinator in Python

```python
class Coordinator:
    def run(self):
        yetToReceive = list(participants)
        self.log.info('WAIT')
        self.chan.sendTo(participants, VOTE_REQUEST)
        while len(yetToReceive) > 0:
            msg = self.chan.recvFrom(participants, TIMEOUT)
            if (not msg) or (msg[1] == VOTE_ABORT):
                self.log.info('ABORT')
                self.chan.sendTo(participants, GLOBAL_ABORT)
                return
            else: # msg[1] == VOTE_COMMIT
                yetToReceive.remove(msg[0])
                self.log.info('COMMIT')
                self.chan.sendTo(participants, GLOBAL_COMMIT)
```
```python
class Participant:
    def run(self):
        msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if not msg:  # Crashed coordinator - give up entirely
            decision = LOCAL_ABORT
        else:  # Coordinator will have sent VOTE_REQUEST
            decision = self.do_work()
            if decision == LOCAL_ABORT:
                self.chan.sendTo(coordinator, VOTE_ABORT)
            else:  # Ready to commit, enter READY state
                self.chan.sendTo(coordinator, VOTE_COMMIT)
                msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if not msg:  # Crashed coordinator - check the others
            self.chan.sendTo(all_participants, NEED_DECISION)
        while True:
            msg = self.chan.recvFromAny()
            if msg[1] in [GLOBAL_COMMIT, GLOBAL_ABORT, LOCAL_ABORT]:
                decision = msg[1]
                break
        else:  # Coordinator came to a decision
            decision = msg[1]
        while True:  # Help any other participant when coordinator crashed
            msg = self.chan.recvFrom(all_participants)
            if msg[1] == NEED_DECISION:
                self.chan.sendTo([msg[0]], decision)
```
Linearizability

**Essence**

Multiple nodes concurrently accessing replicated data. How do we define “consistency” here?

Linearizability (aka strong consistency):

- **Informally:** every operation takes effect *atomically* sometime after it started and before it finished

- All operations behave as if executed on a single copy of the data (even if there are in fact multiple replicas)

- Consequence: every operation returns an “up-to-date” value (aka strong consistency)

Note: Linearizability $\neq$ Serializability!

Combination of both $\rightarrow$ **Strict serializability / one-copy serializability**
Read-after-write consistency (revisited)

Client

set(x, v₁)

(t₁,set(x, v₁))

A

ok

B

ok

C
Read-after-write consistency (revisited)

Client

set(x, v₁)

(t₁, set(x, v₁))

ok

get(x)

→ v₁

A

B

C

(t₀, v₀)

(t₁, v₁)

get(x)

ok

→ v₁

ok

→ v₁
From the client’s point of view

We want client 2 to read value written by client 1, even if the clients have not communicated.
Operations overlapping in time

set(x, v_1) → v_1

g(x)

Which operation takes effect first?
Not linearizable, despite quorum

Client 1

set(x, v₁)

(t₁, set(x, v₁))

Ok

Client 2

Client 3
Not linearizable, despite quorum

Client 1

Client 2

Client 3

set(x, v)

get(x)

A

B

C

Client 1

(t₁, set(x, v₁))

(t₀, v₀)

(t₁, v₁)

Ok

Ok

Ok

get(x)
Not linearizable, despite quorum

Client 1

\( \text{set}(x, v_1) \)  
\((t_1, \text{set}(x, v_1))\)

Ok

Client 2

\( \text{get}(x) \)  
\((t_0, v_0)\)

Client 3

\( \text{get}(x) \)  
\((t_0, v_0)\)

A

B

C

\( \text{set}(x, v_1) \)

Ok

Ok

Ok

Ok

\( \text{get}(x) \)  
\((t_1, v_1)\)
Not linearizable, despite quorum

- Client 2’s operations finishes before client’s 3 operation starts
- Linearizability therefore requires client 3’s operation to observe a state no older than client 2’s operation
- This example violates linearizability because $v_0$ older than $v_1$
Making quorum linearizable

(ABD Algorithm)

Client 1

Client 2

Client 3

set(x, v₁)

get(x)

get(x)

get(x)

set(x, v₁)

set(x, v₁)

get(x) -> v₁

get(x) -> v₁

get(x) -> v₁

Ok

Ok

Ok

Ok

Ok

Ok

Ok

Ok

Ok

(t₁, set(x, v₁))

(t₁, v₁)

(t₀, v₀)

(t₁, set(x, v₁))
Eventual consistency

### Essence

**Linearizability advantages:**
- Makes a distributed systems behave as if it were non-distributed
- Simple for application use

**Drawbacks:**
- **Performance** cost: lots of messages and waiting for responses
- **Scalability** limits: leader can be a bottleneck
- **Availability** problems: if you can’t contact quorum of nodes, you can’t process any operations

**Eventual consistency:** a weaker model than linearizability, different trade-off choices
Eventual consistency

[source] https://9to5mac.com/2021/05/21/best-calendar-app-for-the-mac/
Eventual consistency

**Essence**

Replicas process operations based on their local state

If there are no more updates, eventually all replicas will be in the same state. (No guarantees how long it might take.)

Strong eventual consistency:

- **Eventual delivery**: every update made to one non-faulty replica is eventually processed by every non-faulty replica.

- **Convergence**: any two replicas that have processed the same set of updates are in the same state (even if updates were processed in different orders)
Consistent recovery state

Recovery

When a failure occurs, we need to bring the system into an error-free state:

- **Forward error recovery**: Find a new state from which the system can continue operation
- **Backward error recovery**: Bring the system back into a previous error-free state

Practice

Use backward error recovery, requiring that we establish recovery points

Observation

Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover
Consistent recovery state

Requirement

Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line

Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.
## Coordinated checkpointing

### Essence

Each process takes a checkpoint after a globally coordinated action.

### Simple solution

Use a two-phase blocking protocol:
- A coordinator multicasts a checkpoint request message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a checkpoint done message to allow all processes to continue

### Observation

It is possible to consider only those processes that depend on the recovery of the coordinator, and ignore the rest
Independent checkpointing

Essence
Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let CP\textsubscript{i} (m) denote m\textsuperscript{th} checkpoint of process Pi and INT\textsubscript{i} (m) the interval between CP\textsubscript{i} (m-1) and CP\textsubscript{i} (m).
- When process P\textsubscript{i} sends a message in interval INT\textsubscript{i} (m), it piggybacks (i,m).
- When process P\textsubscript{j} receives a message in interval INT\textsubscript{j} (n), it records the dependency INT\textsubscript{i} (m) → INT\textsubscript{j} (n).
- The dependency INT\textsubscript{i} (m) → INT\textsubscript{j} (n) is saved to storage when taking checkpoint CP\textsubscript{j} (n).

Observation
If process P\textsubscript{i} rolls back to CP\textsubscript{i} (m-1), P\textsubscript{j} must roll back to CP\textsubscript{j} (n-1).
Cascaded rollback

Observation

If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time. We have a so-called cascaded rollback.
Message logging

**Alternative**

Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint → store messages in a log.

**Assumption**

We assume a piecewise deterministic execution model:
- The execution of each process can be considered as a sequence of state intervals
- Each state interval starts with a nondeterministic event (e.g., message receipt)
- Execution in a state interval is deterministic

**Conclusion**

If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.
Message logging and consistency

When should we actually log messages?

Avoid orphan processes:
- Process Q has just received and delivered messages \( m_1 \) and \( m_2 \).
- Assume that \( m_2 \) is never logged.
- After delivering \( m_1 \) and \( m_2 \), Q sends message \( m_3 \) to process R.
- Process R receives and subsequently delivers \( m_3 \): it is an orphan.
Summary

• Learned about the properties that transactions need to guaranteed when applied to different replicas
• Leaned the concepts of Linearizability and Serializability
• Explore the 2PC protocol to achieve consistency in transactions
References

Part of this material is inspired by:

• Distributed Systems course given by Dr. Martin Kleppmann (University of Cambridge, UK)

• Understanding Distributed Systems, Version 1.1.1., Roberto Vitillo, 2021

Next lecture

Naming (Mohan Liyanage)
Questions?

E-mail: huber.flores@ut.ee