

Distributed Transaction Processing with MySQL XA

Sergei Golubchik
Senior Software Developer
MySQL AB

Mark Matthews
Software Development Manager
MySQL AB

MySQL XA: Session Outline

- The problem:
 - What is Distributed Transaction Processing ?
 - Why “distributed transactions” ?
 - X/Open DTP model
 - XA interface
- Solution:
 - MySQL XA
 - Java Transactional API
- MySQL “internal XA”

Who am I ?

- My name is Sergei Golubchik
- I am Senior Software Developer in MySQL AB
- Originally from Ukraine, now living in Kerpen
- Working on MySQL server internals since 1998
- Full-time MySQL AB employee since March 2000
- Main project — Fulltext Search, but also during these years worked with almost every part of MySQL server source

What is DTP

- Distributed Transaction Processing
- Maintaining **consistency** between several separated databases
- Example: wire transfer
 - Bank 1:
 - UPDATE account SET balance=balance - 100000
WHERE account_nr=12345678
 - Bank 2:
 - UPDATE account SET balance=balance + 100000
WHERE account_nr=9876543

How to do it wrong

- Bank 1:
 - UPDATE account SET balance=balance - 100000
WHERE account_nr=12345678
 - COMMIT
- Bank 2:
 - UPDATE account SET balance=balance + 100000
WHERE account_nr=9876543 → **ERROR**
- Bank 1:
 - Oops

How to do it **wrong**

- Bank 1:
 - UPDATE account SET balance=balance – 100000
WHERE account_nr=12345678
- Bank 2:
 - UPDATE account SET balance=balance + 100000
WHERE account_nr=9876543
 - COMMIT
- Bank 1:
 - **ERROR** (e.g. HD crash, power failure, or alien attack)
- Bank 2
 - Oops

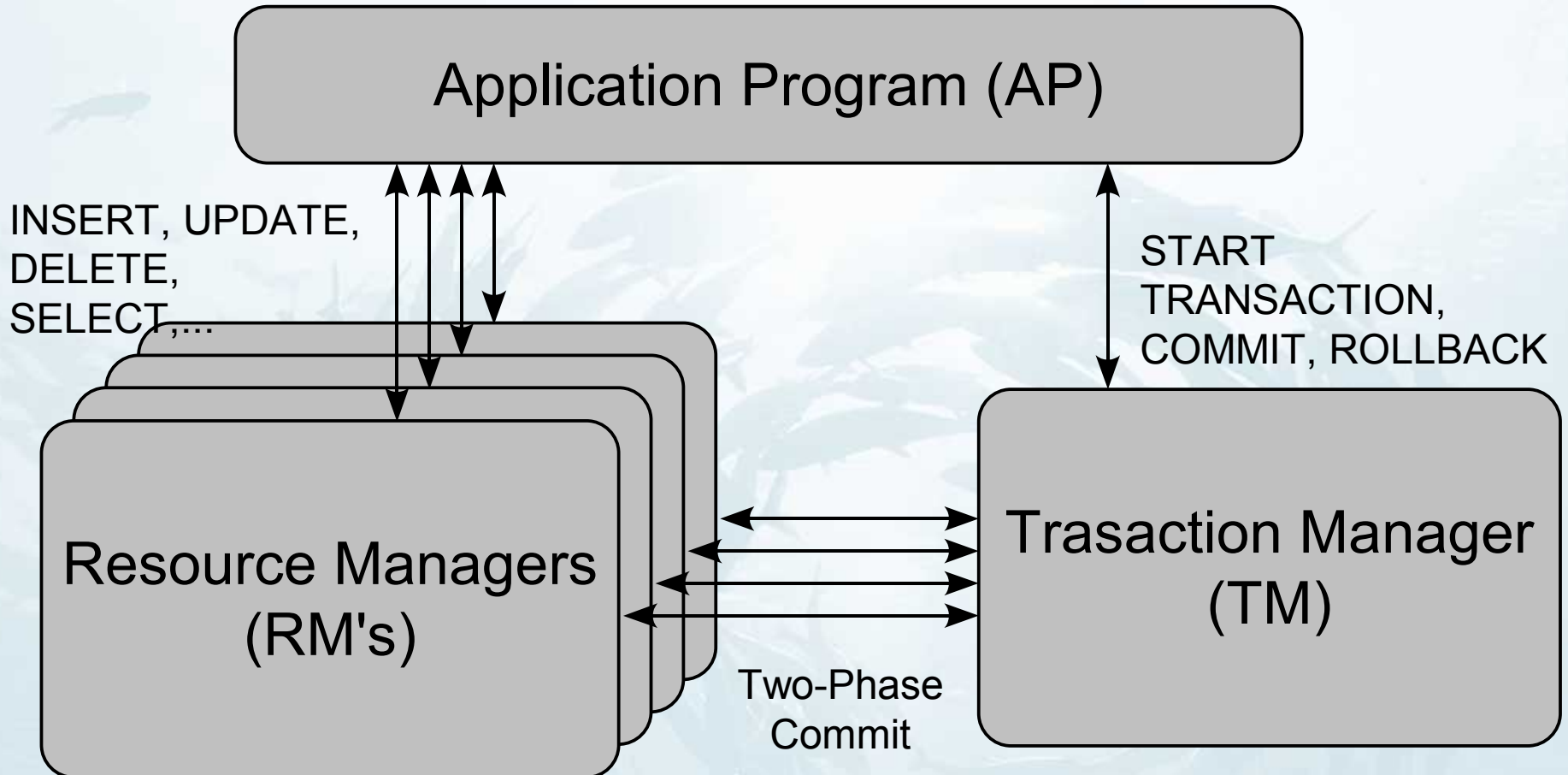
Two-Phase Commit

- Bank 1:
 - UPDATE account SET balance=balance – 100000
WHERE account_nr=12345678
 - **PREPARE**
- Bank 2:
 - UPDATE account SET balance=balance + 100000
WHERE account_nr=9876543
 - **COMMIT**
- Bank 1:
 - **COMMIT**

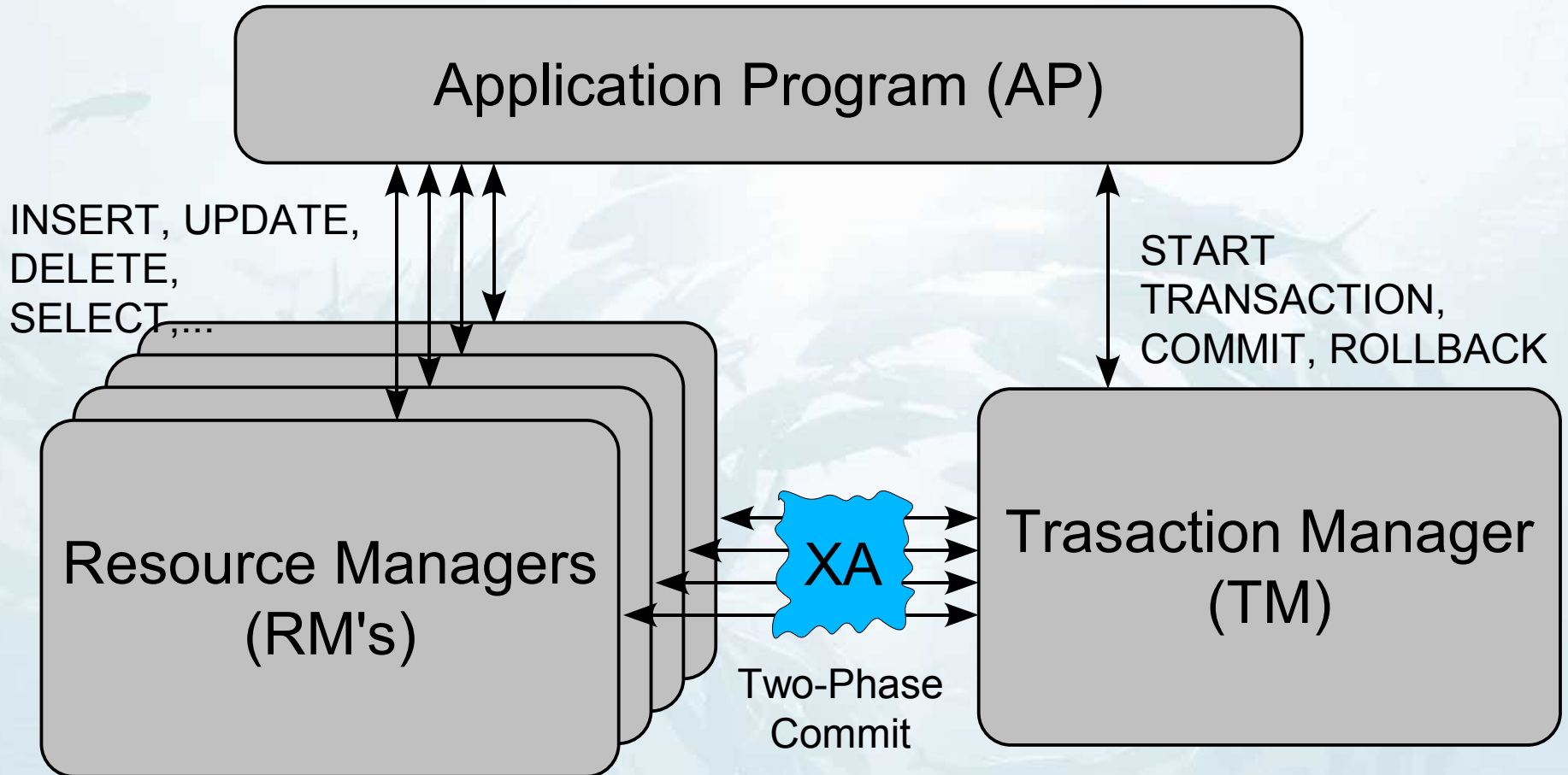
X/Open DTP Model

- An Application Program (AP)
 - defines transaction boundaries
 - specifies actions that constitute a transaction
- Resource Managers (RM's)
 - provide access to shared resources
- A Transaction Manager (TM)
 - assigns identifiers to transactions
 - takes responsibility for transaction completion and recovery

X/Open DTP Model



X/Open DTP Model



XA Interface: XID

- Transaction/Branch Identifier:
 - GTRID: Global Transaction ID — max. 64 bytes
 - Identifies global transaction — atomic unit of work
 - BQUAL: Branch Qualifier — max. 64 bytes
 - Identifies transaction branch within global transaction
 - Each branch participates separately in two-phase commit protocol
- XID's must be **globally unique**

XA Routines

ax_reg	Register an RM with a TM.
ax_unreg	Unregister an RM with a TM.
xa_close	Terminate the AP's use of an RM.
xa_commit	● Tell the RM to commit a transaction branch.
xa_complete	Test an asynchronous xa_ operation for completion.
xa_end	● Dissociate the thread from a transaction branch.
xa_forget	Permit the RM to discard its knowledge of a heuristically-completed transaction branch.
xa_open	Initialise an RM for use by an AP.
xa_prepare	● Ask the RM to prepare to commit a transaction branch.
xa_recover	● Get a list of XIDs the RM has prepared or heuristically completed.
xa_rollback	● Tell the RM to roll back a transaction branch.
xa_start	● Start or resume a transaction branch — associate an XID with future work that the thread requests of the RM.

Anatomy of the distributed transaction

-2. TM → RM1: xa_start

-1. TM → RM2: xa_start

0. the real work is here

1. TM → RM1: xa_prepare; RM1: record transaction

2. TM → RM2: xa_prepare; RM2: record transaction

3. TM: record commit decision

4. TM → RM1: xa_commit; RM1: commit

5. TM → RM2: xa_commit; RM2: commit

6. done!

Crash Recovery

- TM → RM1: xa_recover
- RM1 → TM: xid1, xid2, xid5, ...
- TM → RM2: xa_recover
- RM2 → TM: xid1, xid3, xid4, ...
- TM looks in the log: xid1, xid2, xid6 ...

Crash Recovery

- TM → RM1: xa_recover
- RM1 → TM: xid1, xid2, xid5, ...
- TM → RM2: xa_recover
- RM2 → TM: xid1, xid3, xid4, ...
- TM looks in the log: xid1, xid2, xid6 ...
- TM → RM1: xa_commit(xid1), xa_commit(xid2), xa_rollback(xid5)
- TM → RM2: xa_commit(xid1), xa_rollback(xid3), xa_rollback(xid4)

Anatomy of the distributed transaction

-2. TM → RM1: xa_start

-1. TM → RM2: xa_start

0. the real work is here

1. TM → RM1: xa_prepare; RM1: record transaction (**sync**)

2. TM → RM2: xa_prepare; RM2: record transaction (**sync**)

3. TM: record commit decision (**sync**)

4. TM → RM1: xa_commit; RM1: commit (**sync**)

5. TM → RM2: xa_commit; RM2: commit (**sync**)

6. done!

Optimizing 2PC: Lazy xa_start

- AP → TM: “start transaction”
 - TM: does nothing
- AP → RM1 (via TM): “do something”
 - TM → RM1: xa_start
 - TM1 → RM1: “do something”
- AP → RM2 (via TM): “do something”
 - TM → RM2: xa_start
 - TM1 → RM2: “do something”

Optimizing 2PC: Fallback to 1PC

-2. TM → RM1: xa_start

-1. TM → RM2: xa_start

0. the real work is here

1. TM → RM1: xa_prepare; RM1: record transaction (**sync**)

2. TM → RM2: xa_prepare; RM2: record transaction (**sync**)

3. TM: record commit decision (**sync**)

4. TM → RM1: xa_commit; RM1: commit (**sync**)

5. TM → RM2: xa_commit; RM2: commit (**sync**)

6. done!

Optimizing 2PC: Fallback to 1PC

-2. TM → RM1: xa_start

-1. TM → RM2: xa_start

0. the real work is here

1. TM → RM1: xa_prepare; RM1: record transaction (**sync**)

2. TM → RM2: xa_prepare; RM2: record transaction (sync)

3. TM: record commit decision (**sync**)

4. TM → RM1: xa_commit; RM1: commit (**sync**)

5. TM → RM2: xa_commit; RM2: commit (sync)

6. done!

Optimizing 2PC: Fallback to 1PC

-2. TM → RM1: xa_start

0. the real work is here

1. TM → RM1: xa_prepare; RM1: record transaction (sync)

3. TM: record commit decision (sync)

4. TM → RM1: xa_commit; RM1: commit (**sync**)

6. done!

Commit Grouping

- My home PC: 1,000 sync/sec
- For TM it means: no more than 1,000 commits/sec
- If commits are coming at rate 5,000/sec, then while the first will be syncing, 5 new will come.
- Let's log them in one I/O block and sync in one do.
- Throughput with grouping: $1,000 * \text{xids_per_io_block}$
- For MySQL as TM it means 680,000 commits/sec
- But don't take this number literally ! 😊

OS-specific tricks

- TM has to `fsync()` but it does not have to `write()`
- By mapping the complete log into the memory with `mmap()`, one can sync twice as fast
- My home PC:
 - `write()+fsync()` → ~ 300–500 syncs/sec
 - `mmap()+fsync()` → ~ 1000 syncs/sec
- `fdatasync()` instead of `fsync()`

XA inside MySQL

- MySQL has many storage engines
- You can use more than one storage engine in one transaction
- Therefore MySQL needs 2PC even **internally** to guarantee consistency
- MySQL acts as a TM, with storage engines being RM's. MySQL is responsible for recovery logging
- It is implemented in 5.1
- All mentioned optimizations are, of course fully implemented, and more are to come...

MySQL XA Statements

- XA START *xid* [JOIN | RESUME]
- XA END *xid* [SUSPEND [FOR MIGRATE]]
- XA PREPARE *xid*
- XA COMMIT *xid* [ONE PHASE]
- XA ROLLBACK *xid*
- XA RECOVER

2PC Example

1> XA START "foobar10029i3";

2> XA START "foobar10029i3";

1> SELECT * FROM stats;

2> INSERT INTO aggr VALUES (...);

1> DELETE FROM stats;

1> XA PREPARE "foobar10029i3";

2> XA COMMIT "foobar10029i3";

1> XA COMMIT "foobar10029i3";

XA Primer with JTA

- The “2PC example” was too simple
- Does not follow X/Open DTP Model
- No TM to perform recovery logging — no recovery in case of crash
- Using MySQL XA statements one can implement complete TM relatively easy
- When using *Java Transaction API* one usually has TM he can use

JDBC Example

(assuming you're inside a J2EE container, or using standalone JTA, and have configured your datasource as "com.mysql.jdbc.jdbc2.optional.MysqlXADataSource")

```
//  
// Transaction Managers are usually bound into JNDI when using JTA  
// UserTransactions  
//  
  
Context ictx = new InitialContext();  
  
//  
// Lookup the TM in JNDI which gives us the UserTransaction  
//  
  
UserTransaction utx = (UserTransaction)ictx.lookup("UserTransaction");  
  
//  
// Begin the user transaction (if you were inside something that  
// was container-managed txn's, such as an EJB, you wouldn't need to  
// use user transactions at all, the container would handle it for you  
//
```

JDBC Example Continued

```
//  
// Get a connection from an XADataSource...  
//  
// This assumes that the XADataSource has been  
// registered with your TM, which is the case with a J2EE appserver...  
// If you're using JTA standalone, this is done in some non-portable  
// (but usually declarative) way...  
//
```

```
Connection conn = myXADataSource.getConnection();
```

```
//  
// Do JDBC-like stuff with this connection  
//  
// Reference any other XA-compliant resources (datasource, jms, EJB)  
// as well, and the TM takes care of enlisting them automagically  
//
```

```
conn.close(); // doesn't actually close the connection, the TM holds  
// on to this until the global tx commits or rolls back.
```

```
utx.commit(); // TM takes care of two-phase here for us
```

Questions ?