Firebird Conference 2005

JayBird: JCA/JDBC driver, part 2

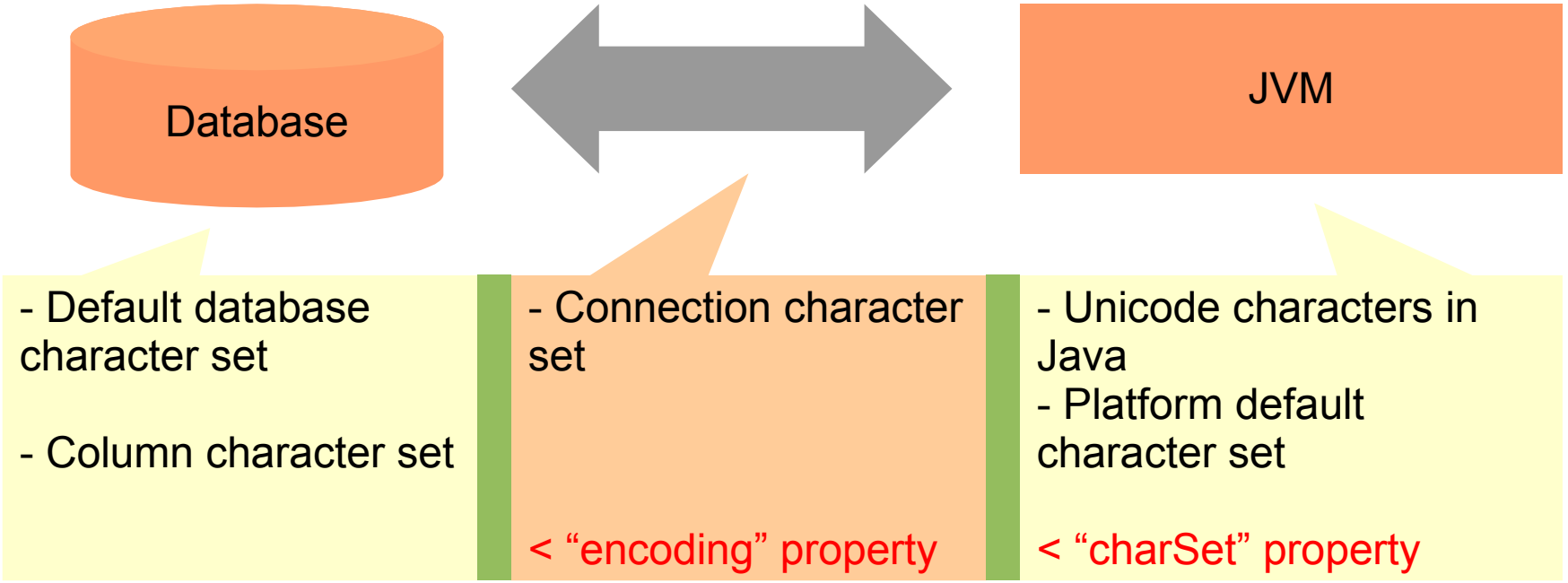
Roman Rokytskyy
Firebird Foundation

Agenda

- JayBird: introduction
  - History
  - Architecture
  - Connections
  - Statements
  - Result sets
  - Pooling
  - Embedded mode
  - Logging
  - Error handling
- JayBird: advanced
  - Character sets (brief)
  - BLOBs
  - Services API
  - Transactions

Character sets

Character sets



- “encoding” property
  - specifies in which charset client expects data to be send from the server
  - Allowed value can be found in Firebird documentation
- “charSet” property
  - specifies the charset to assume when converting byte[] into String and vice versa
  - Allowed values can be found in JVM documentation

Character sets

- NONE database charset
  - database server cannot interpret characters and sends the byte stream “as is”
  - JayBird cannot interpret the byte stream too, so it falls back to the platform default encoding
    - new String(someByteArray);
    - works on Windows quite well, but on Linux usually corrupts the data because of default “C” locale

Solution

- specify the “charSet” property to the encoding in which data is returned from the server

- UNICODE_FSS
  - database server can interpret characters
  - JayBird can interpret characters

Problems:

- no collations
  - will appear in UTF-8 in Firebird 2.0
- for Russian characters traffic increases approx. 2x
- JVM has to translate UTF-8 into internal representation anyway

BLOBs

BLOBs in JayBird

- java.sql.Blob
  - introduced in JDBC 2.0 with read-only access
  - JDBC 3.0 provides write access
    - setBytes(long, byte[], int, int)
    - setBinaryStream(long)
  - JDBC 4.0 will provide means to create BLOBs
- java.sql.Clob
  - “same” as BLOB, but for character streams
- org.firebirdsql.jdbc.FirebirdConnection
  - createBlob():java.sql.Blob
- org.firebirdsql.jdbc.FirebirdBlob
  - seek(long)
  - read(byte[], int, int)
  - readFully(byte[], int, int)

BLOBs: Theory

- Facts from the documentation
  - BLOBs are stored separately from the rest of the record
  - a unique BLOB identifier (8 bytes) is stored on the record page
  - when BLOB is created, it gets temporary ID which is made persistent on commit
  - almost infinite size
- Two BLOB types exist
  - segmented
    - has segment size (specified during table creation)
    - does not support “seek”
  - stream
    - just a stream of bytes
    - supports “seek” operation
- API
  - iscCreateBlob
  - iscOpenBlob
  - iscPutSegment
    - up to 64k segment can be written
  - iscGetSegment
    - up to 64k segment can be read
  - iscSeekBlob
  - iscCloseBlob
  - iscBlobInfo
    - type of the BLOB
    - number of segments in the BLOB
    - max. size of the BLOB segment
    - length of the BLOB in bytes

BLOBs: Reality

- Segments
  - the segment size specified during table creation for “segmented” BLOB is ignored
  - the BLOB reading happens in chunks as they were written
    - the blobBufferSize matters for good performance, especially when writing a BLOB, the smaller chuncks are, the more loops will be done in Firebird and in JayBird
    - the blobBufferSize should correspond the database page size
    - the socketBufferSize should match the blobBufferSize for better performance

- “seek” operation
  - can address only BLOBs of max 2 GB size, since the seek offset is a signed 32-bit integer
  - supported only on reading, on writing it is silently ignored

- BLOB size

  max_size = page_size ^3 / 16

  1k page size – max 64 MB

  4k page size – max 4 GB

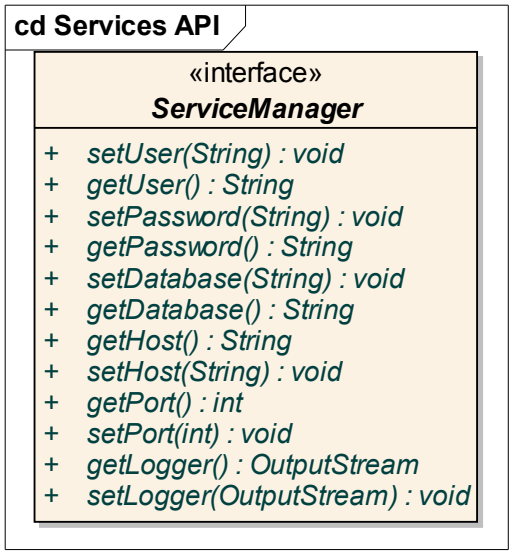
Warning

- gbak for Firebird 1.0.x (and possibly Firebird 1.5.0) had a bug that prevented backing up the database containing stream BLOBs

But if BLOBs are used in a right way, they are fast enough to compete with the file system

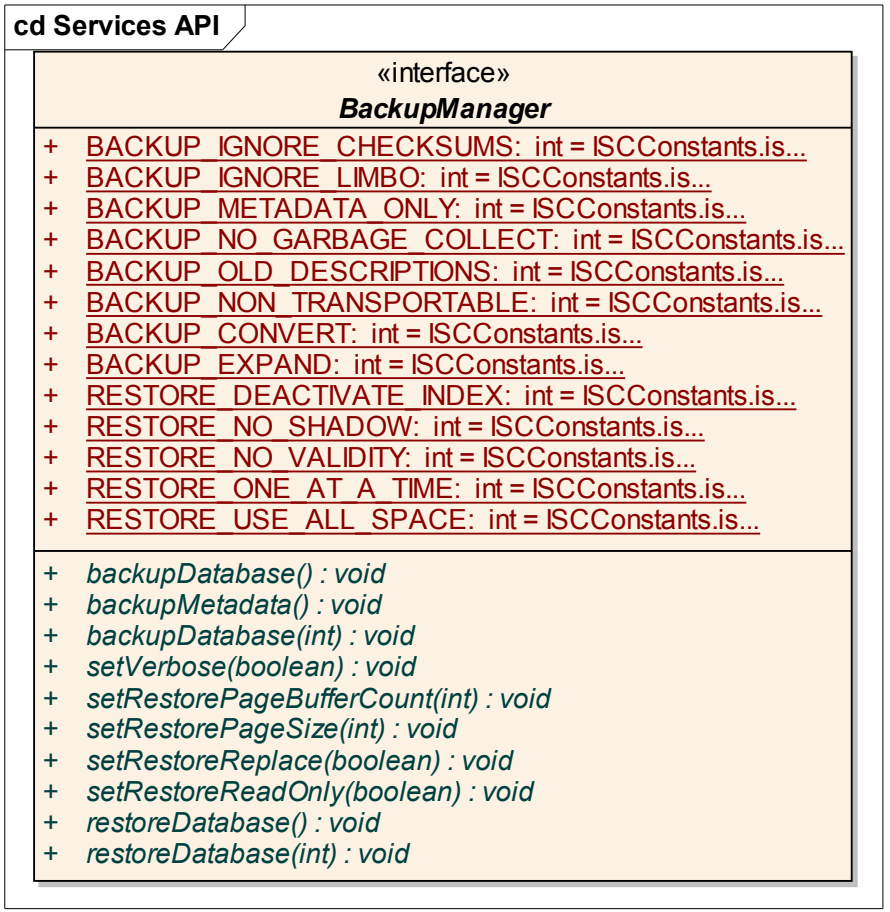
Services API

Services API



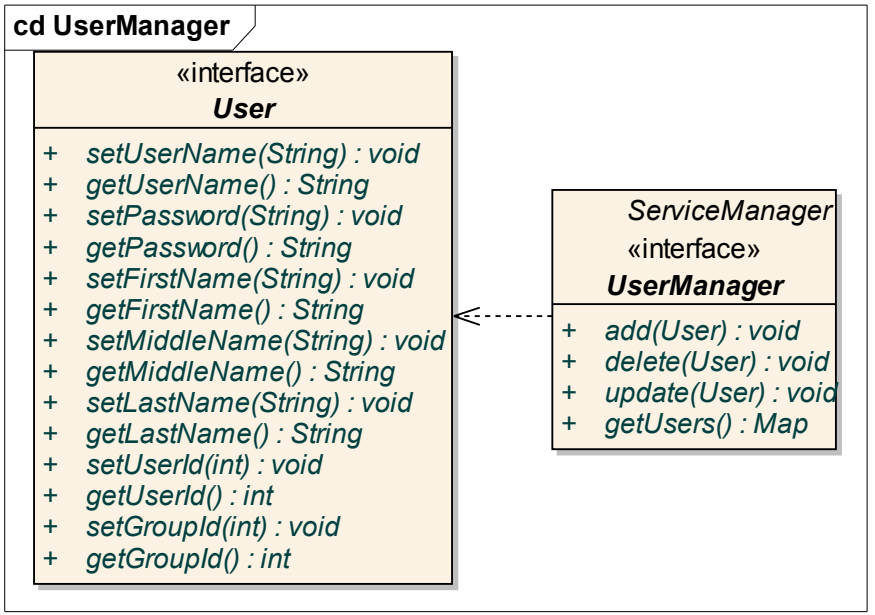
- Server-wide connection
  - host
  - port
  - user name
  - password
  - database
- Service queuing
- Functional groups
  - Backup/Restore
    - backup database/metadata only
    - restore / replace existing
    - change page size during restore
  - User management
    - add
    - modify
    - delete
  - Database maintenance
    - forced writes
    - database dialect
    - shutdown/bring online
    - validate/mark corrupt records
    - sweep
    - shadow management
    - in-limbo transaction management
  - Statistics
    - header statistics
    - table statistics
    - index statistics
    - system table statistics

BackupManager




```
FBManager fbManager = createFBManager();

fbManager.setServer(DB_SERVER_URL);
fbManager.setPort(DB_SERVER_PORT);
fbManager.start();

fbManager.setForceCreate(true);
fbManager.createDatabase(
    getDatabasePath(), DB_USER, DB_PASSWORD);


BackupManager backupManager =
    new FBBackupManager();

backupManager.setHost(DB_SERVER_URL);
backupManager.setUser(DB_USER);
backupManager.setPassword(DB_PASSWORD);

backupManager.setDatabase(getDatabasePath());
backupManager.setBackupPath(getBackupPath());

backupManager.setLogger(System.out);
backupManager.setVerbose(true);

backupManager.backupDatabase();

fbManager.dropDatabase(
    getDatabasePath(), DB_USER, DB_PASSWORD);

tryConnectAndFailIfSucceedded();

backupManager.restoreDatabase();

tryConnectA();
```

UserManager




```
// Initialize the UserManager.
UserManager userManager =
    new FBUserManager();

userManager.setHost(DB_SERVER_URL);
userManager.setUser(DB_USER);
userManager.setPassword(DB_PASSWORD);

// Add a user.
User user1 = new FBUser();
user1.setUserName("TESTUSER123");
user1.setPassword("tes123");
user1.setFirstName("First Name");
user1.setMiddleName("Middle Name");
user1.setLastName("Last Name");
user1.setUserId(222);
user1.setGroupId(222);

userManager.add(user1);

// Check to make sure the user was added.
User user2 = (User)
    userManager.getUsers().get(
        user1.getUserName());

// User2 should be correct and not null
assertTrue("User 2 should not be null.",
    user2 != null);
assertTrue("user1 should equal user2",
    user1.equals(user2));

// update the password
user1.setPassword("123test");
userManager.update(user1);

// delete user
userManager.delete(user1);
```

MaintenanceManager


```
MaintenanceManager maintenanceManager =
  new FBMaintenanceManager();

maintenanceManager.setHost(DB_SERVER_URL);
maintenanceManager.setUser(DB_USER);
maintenanceManager.setPassword(DB_PASSWORD);
maintenanceManager.setDatabase(DB_PATH);
maintenanceManager.setLogger(System.out);

int[] trid =
  maintenanceManager.getLimboTransactions();

// there should be no in-limbo transactions
assertEquals(0, trid.length);

createLimboTransaction(3);

// now there should be three in-libo tx
trid =
  maintenanceManager.getLimboTransactions();

// rollback one
maintenanceManager.rollbackTransaction(
  trid[0]);

// now there should be 2 tx
trid =
  maintenanceManager.getLimboTransactions();
assertEquals(2, trid.length);
```

Transactions

Transactions: functions

- JDBC level
  - one transaction per connection
  - no explicit transaction start
  - commit or rollback
- JCA level
  - LocalTransaction
    - begin()
    - commit()
    - rollback()
  - XAResource

- Firebird transactions
  - iscStartTransaction(...)
  - iscPrepareTransaction(...)
  - iscCommitTransaction(...)
  - iscRollbackTransaction(...)
  - iscTransactionInformation(...)
  - iscReconnectTransaction(...)

Hack from Borland

- iscCommitRetaining(...)
- iscRollbackRetaining(...)

Warning

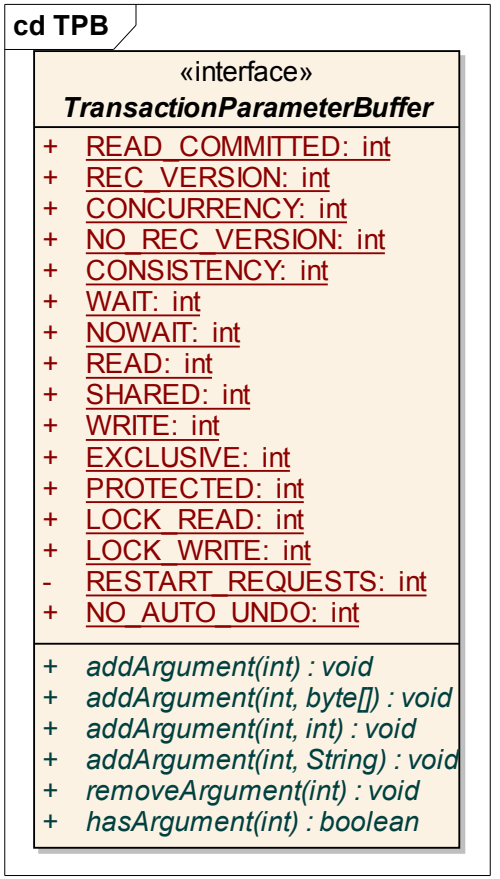
- starting multiple transactions is not supported!

Transactions: isolation levels

- JDBC isolation levels
  - NONE
  - READ UNCOMMITTED
    - dirty reads, non-repeatable reads and phantom reads can occur
  - READ COMMITTED
    - dirty reads are prevented
    - non-repeatable reads and phantom reads can occur
  - REPEATABLE READ
    - dirty reads and non-repeatable reads are prevented
    - phantom reads can occur
  - SERIALIZABLE
    - dirty reads, non-repeatable reads and phantom reads are prevented
- Firebird isolation levels
  - READ COMMITTED REC_VER
    - dirty reads are prevented
    - non-repeatable reads and phantom reads can occur
  - READ COMMITTED NO_REC_VER
    - same as above
    - conflict when uncommitted version is found
  - CONCURRENCY
    - dirty reads, non-repeatable reads and phantom reads are prevented
  - CONSISTENCY
    - table reservation possibility
    - truly serial execution of transaction
- Other parameters
  - READ or WRITE
  - WAIT or NOWAIT

Transaction parameters

- FirebirdConnection
  - createTransactionParameterBuffer()
  - getTransactionParameters()
  - setTransactionParameters(TPB)
  - setTransactionParameters(int, TPB)
- FBManagedConnection
  - setTransactionParameters(TPB)
  - setTransactionParameters(int, TPB)
- FirebirdDriver
  - newConnectionProperties()
  - connect(FirebirdConnectionProperties)
- FBManagedConnectionFactory
  - setTransactionParameters(int, TPB)
  - setTpbMapping(String)



Using TransactionParameterBuffer

Changing the WAIT/NOWAIT parameter for the mapping of READ COMMITTED isol.

```
FirebirdConnection connection = (FirebirdConnection)...

TransactionParameterBuffer tpb =
    connection.getTransactionParameters(Connection.TRANSACTION_READ_COMMITTED);

tpb.removeArgument(TransactionParameterBuffer.WAIT);
tpb.addArgument(TransactionParameterBuffer.NOWAIT);

connection.setTransactionParameters(Connection.TRANSACTION_READ_COMMITTED, tpb);
connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);

connection.setAutoCommit(false);
```

Defining new mapping for the next transaction

```
TransactionParameterBuffer anotherTpb = connection.createTransactionParameterBuffer();

anotherTpb.addArgument(TransactionParameterBuffer.CONSISTENCY);
anotherTpb.addArgument(TransactionParameterBuffer.WRITE);
anotherTpb.addArgument(TransactionParameterBuffer.NOWAIT);

anotherTpb.addArgument(TransactionParameterBuffer.PROTECTED);
anotherTpb.addArgument(TransactionParameterBuffer.LOCK_WRITE, "TEST_LOCK");

connection.setTransactionParameters(anotherTpb);
```

XA Transactions

- X/Open Distributed Transaction Processing specification
  - Applications
  - Resource managers
  - Transaction manager

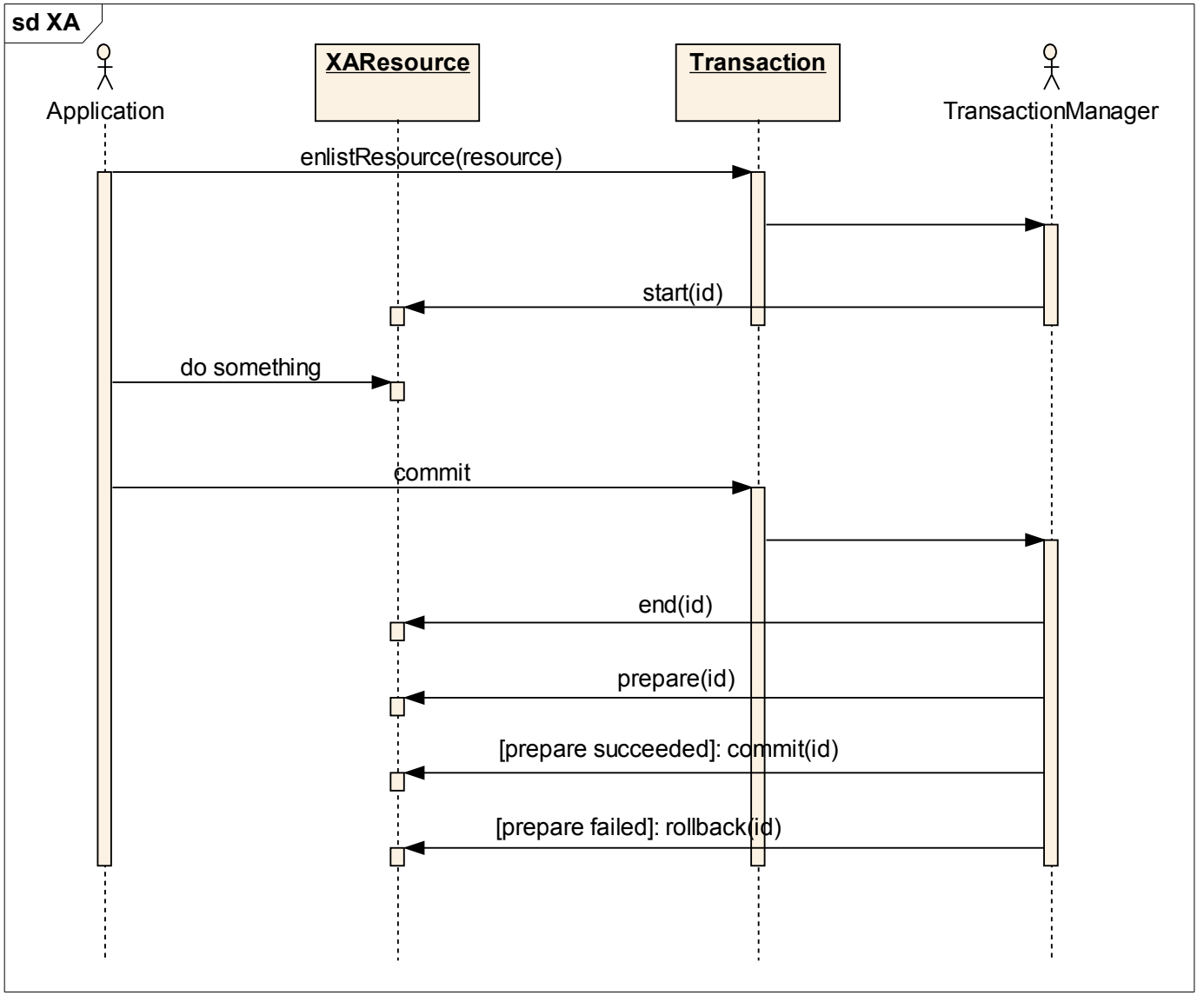
Distributed transaction
  global transaction
    Work performed by a single resource manager can be successfully completed only if work done by other resource managers can complete successfully
  transaction branch
    branch is associated with a request to each resource manager involved in a distributed transaction

- Thread of control
  - “an operating system process: an address space and single thread of control that executes in within that address space, and its required system resources”
  - Java Transaction API maps thread of control concept to all Java threads that are given access to the resource manager
  - A tightly-coupled threads are threads that share resources and are treated by resource manager as single entity
    - In particular this means that resource manager must guarantee absence of deadlocks in a transaction branch
  - A loosely-coupled relationship does not require such guarantee, and two threads can be treated as if they were in separate global transactions that are completed atomically

Main entities of JTA specification

- Xid
  - getBranchQualifier():byte[]
  - getFormatId():int
  - getGlobalTransactionId():byte[]
- XAResource interface
  - start(Xid, int)
  - end(Xid, flags)
  - prepare(Xid)
  - commit(Xid, boolean)
  - rollback(Xid)
  - isSameRM(XAResource)
  - recover(int):Xid[]
  - forget(Xid)
- TransactionManager
  - getTransaction():Transaction
- Transaction
  - enlistResource(XAResource)
  - delistResource(XAResource)

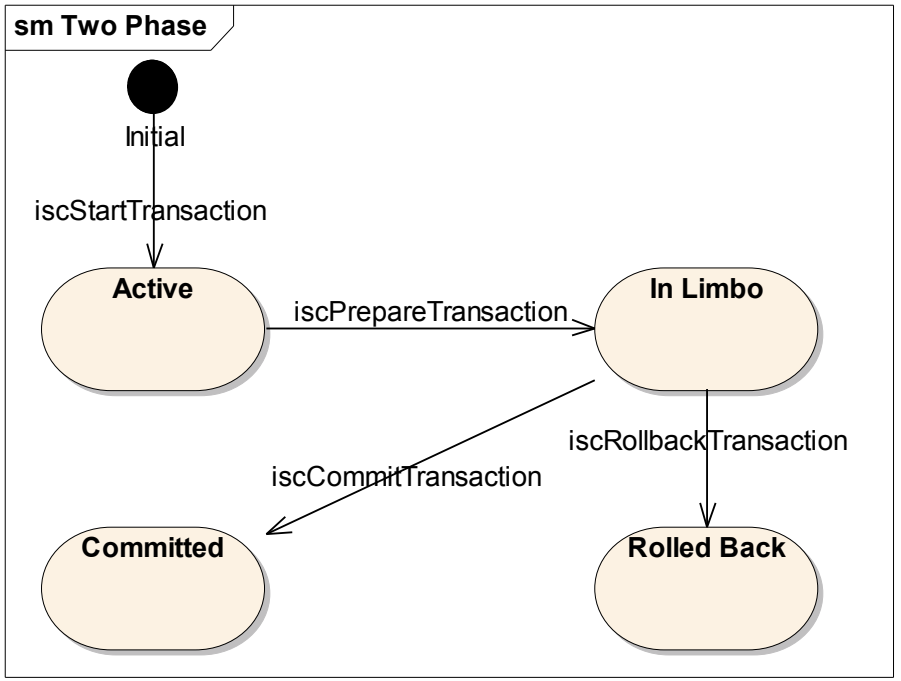
XA Transaction dynamics



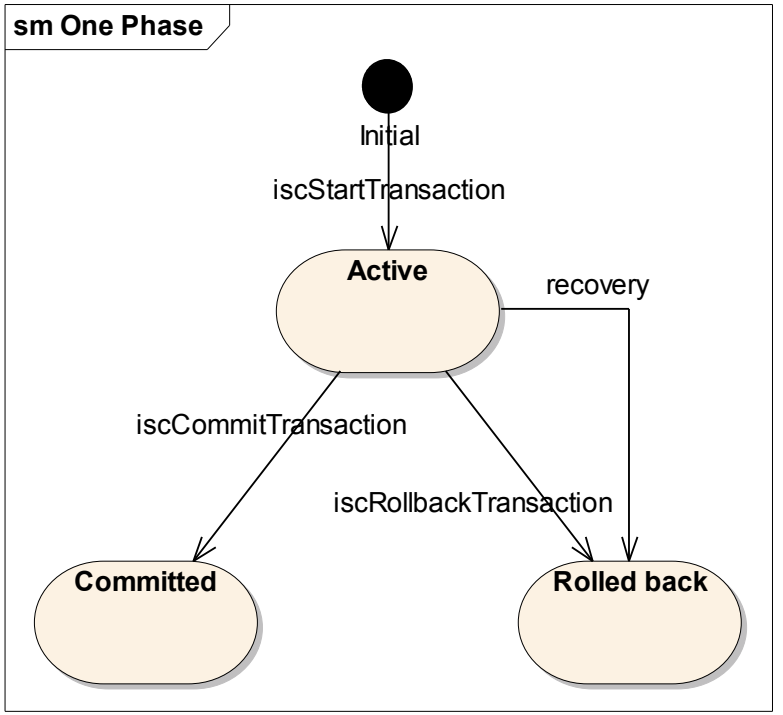
XA Transactions, JayBird and Firebird

- Firebird
  - internal transaction ID
  - transaction handle
  - In-Limbo transactions

XAResource transaction management



LocalTransaction transaction mgmt.



JayBird
- mapping between transaction handle and Xid
- bookkeeping during prepare phase
- recovery of in-limbo transactions

Transactions - Conclusions

- MGA Architecture
  - Virtually one isolation level
  - Somehow artificial mapping to ANSI levels
- TransactionParameterBuffer
  - mapping between ANSI isolation and Firebird isolation
  - additional features
    - no record versions for detecting concurrent updates
    - no-wait to get immediate lock conflict notification
    - table reservation with different lock modes
  - TPB mapping per connection or just for the next transaction
- XA Transactions
  - Powerful mechanism
  - Fully JCA and JTA compliant
  - Overhead of the two-phase commit
  - Built-in recovery for in-limbo transactions
  - Better use LocalTransaction where possible

Questions?