

# Components & Transactions

*Marek Prochazka*

*Marek.Prochazka@inrialpes.fr*



# Transactional Components: Key Points

→ **Component participation in a transaction**

→ **Concurrency control**

→ **Transaction context propagation**

→ **Recovery**

→ **...?**

→ **Fractal:**

➤ **Interception of incoming invocations**

➤ **Use of various transactional controllers with well-defined interfaces**

- Commit Controller
- Concurrency Controller
- Transaction Propagation Controller
- ...

➤ **This is applied at any level of nesting**

# Component Participation in a Transaction

➔ **Key question: How components will take part in transactions?**

1. **Fractal-unaware transactions, transaction-unaware Fractal**
  - Object-level transactions (JTA, OTS)
2. **API-dependent Commit Controller**
  - Use one of the current standard APIs (JTA, OTS)
  - Define a new transactional API
3. **API-independent Commit Controller**
  - Being able to use existing standards

# API-Dependent Commit Controller

➔ Simple registration of component as a transaction participant

```
interface TwoPhaseCommitController {  
    boolean prepare( );  
    void commit( );  
    void abort( );  
}
```

```
interface SynchronizationController {  
    void before_completion( );  
    void after_completion(boolean committed);  
}
```

# API-Dependent Commit Controller

## → Pros:

- Easy to implement, straightforward solution

## → Cons:

- This gives the same functionality as OTS or JTA (API-dependent solution)
- Concurrency control on components?
  - Multiple commit controllers?
  - Collection interface of commit controller?
    - Which controller to register if there are many of them?
- Commit protocol and API are predefined

# API-Independent Commit Controller

```
interface CommitController {  
    void registerResources(BasicTransaction tx)  
}
```

## → Pros:

- Any API for transaction participant registration can be used
- Concurrency control in the register method

## → Cons:

- When to invoke the registerResources() method?
  - In the first visit of transaction?
  - Any time a method is invoked in a transaction? - Then it's just an interception method
- How to do concurrency control in the registerResources() method?
- It needs to be more elaborated

## → Component = unit of concurrency control

- EJB, CCM: components visited by a transaction are always locked by it
- CORBA: objects are not locked if a locking system is not used (Concurrency Control Service)

→ Denote component methods as READ or WRITE

→ An example:

```
<interface name="Account">  
  <method name="withdraw" mode="WRITE"/>  
  <method name="deposit" mode="WRITE" />  
  <method name="balance" mode="READ" />  
</interface>
```

```
<conflict_table name = RW>  
  <conflict req="READ" hold="READ" val="false" />  
  <conflict req="WRITE" hold="*" val="false" />  
</conflict_table>
```

→ Why not to use locking model with semantically rich operations?

# More Advanced Locking: An Example

→ Associate operations with “user-defined” lock modes

→ An example:

```
<interface name="Account">  
  <method name="withdraw" />  
  <method name="deposit" />  
  <method name="balance" />  
</interface>
```

```
<conflict_table name = AccountCTable>  
  <conflict req="balance" hold="*" val="false" />  
  <conflict req="deposit" hold="balance" val="false" />  
  <conflict req="deposit" hold="deposit" val="false" />  
  <conflict req="deposit" hold="withdraw" val="true" />  
  <conflict req="withdraw" hold="*" val="true" />  
</conflict_table>
```

## → Pros:

- Exploiting method semantics
- Bigger potential for sharing
- Easy to define (single conflict table per component type)
- Jotm2 Lock Manager implemented and ready to be used:
  - User-defined lock modes
  - User-defined conflict table
    - Non-symmetric
    - Non-transitive
  - Automatic lock mode conversion according to lock conversion lattice

## → Cons:

- ???

# More Advanced Locking: Concurrency Controller

```
interface ConcurrencyController {  
    boolean beforeInvocation(BasicTransaction tx,  
                            String methodName);  
}
```

## → Semantic Lock Concurrency Controller

- An implementation base on the Jotm2 Lock Manager

## → Other Implementations possible

## → Cons:

- beforeInvocation() looks like a very generic interception method
- This approach may seem too much based on the jotm2 locking

# Concurrency Control in General

- **Concurrency control = scheduling operations**
- **Conservative schedulers**
  - More delaying operations
  - Extreme case - serial execution: delay operation of all but one transaction *tx*, when *tx* finishes select another *tx*
- **Aggressive schedulers**
  - More scheduling operations immediately, fewer delaying, often have to reject operations by transaction abort
  - Extreme case - Optimistic CC: no delaying and certifying at the time of commit
- **Various implementations**
  - Lock-based (most common)
  - Timestamp-based
  - Serialization Graph Testing
- **Multiversion concurrency control**

➔ Typically, scheduler is contacted by TM when an operation is executed

➔ Scheduler can:

- Schedule operation
- Reject operation (tx aborted)
- Delay operation

➔ The criteria upon which the scheduler decides are implementation- and scheduler-type- dependent

# Generic Concurrency Controller

## → Should not be related to locking

```
interface GenericConcurrencyController {  
    boolean beforeInvocation(BasicTransaction tx,  
                            String operation)  
}
```

## → beforeInvocation() can:

- Delay operation invocation (temporarily suspend the current thread)
- Reject operation by returning false (and e.g. rolling back the transaction)
- Schedule operation (by returning true)

## → Cons:

- beforeInvocation() is again very generic interception method (transaction-aware)
  - It can also modify transaction context (context propagation) or register new transaction participants
- **Scheduler often needs “global” information**
  - Need for communication among distributed schedulers

## ➔ Transaction context interrogation and modification when intercepting transactional invocation

- Client transaction have must/must not be present
- Client transaction can be suspended
- Client transaction can be propagated
- Controller-managed transaction can be invoked
- Specific relationships between transactions can be established
  - E.g. parent-child relationship

# Transaction Propagation: A Simple Solution

## → Predefined set of policies:

```
interface TxPropagationController {
    final static int TX_NEVER ...
    final static int TX_NOT_SUPPORTED ...
    final static int TX_REQUIRED ...
    ...
    final static int TX_SUPPORTS_INTERROGABLE
    final static int TX_EXPLICIT ...

    int getPropagationPolicy(String methodName);
}
```

## → Each implementation defines propagation policy value for every method name

## → Cons:

- This is not a generic solution
- Everything has to be implemented by the interceptor, not by controller implementations
- No way to define new propagation policies

# Transaction Propagation: Another Solution

➔ **Interceptor where any transaction propagation policy can be defined**

```
interface TransactionPropagationController {  
    boolean beforeInvocation(BasicTransaction tx,  
                            String operation)  
}
```

➔ **Cons:**

- The same work can be also done in the `ConcurrencyController.beforeInvocation( )` method
- Again, this is not specific interface but a generic invocation interception

## → When to specify transactional features?

### ➤ During deployment

- E.g. EJB deployment descriptor

### ➤ When coding the component

- Only the component author can specify concurrency control or transaction propagation policies (as well as recovery code)

### ➤ ???

## → How to specify

### ➤ Attribute values

### ➤ Another formalism

### ➤ Hardcoded in Commit Controller, Concurrency Controller, and TxPropagationController implementations

### ➤ ???

## → Select the most suitable solution

- Cooperation with JOTDF (LIFL)

## → Prototype

- Concurrency Controller based on jotm2 semantic locking

## → Recovery

- No clear idea yet

## → Transactional reconfiguration

- Use of transaction manager for reconfiguration
  - Proof of the concept
- Non-transactional and transactional applications

## ➔ Transactions in component – several different areas

- Component participation in transactions
- Concurrency control
- Transaction context propagation
- (Recovery)

## ➔ Looking for something between two extreme approaches:

1. Just an interception method, all code implemented in controller
    - Difficult to find a generic interface
  2. Transactional behavior predefined, almost all implemented in interceptor
- Related to definition of jotm2 core services