Atomic RMI 2: Distributed Transactions for Java

Paweł T. Wojciechowski and Konrad Siek
Poznań University of Technology
{pawel.t.wojciechowski,konrad.siek}@cs.put.edu.pl

30 X 2016

http://dsg.cs.put.poznan.pl
Transactional memory

Concurrency control is notoriously difficult:

- interaction between unrelated threads
- additional structural code
- deadlocks, livelocks, priority inversion

```java
synchronized{aLock} {
    synchronized{bLock} {
        a = b;
    }
    b = b + 1;
}
```
Transactional memory

Concurrency control is notoriously difficult:

- interaction between unrelated threads
- additional structural code
- deadlocks, livelocks, priority inversion

```java
synchronized{aLock} {
  synchronized{bLock} {
    a = b;
  }
  b = b + 1;
}
```

```java
transaction.start();
a = b;
b = b + 1;
transaction.commit();
```

Transactional memory (TM):

- ease of use on top
- efficient concurrency control under the hood
Transaction abstraction

Transaction:
\[ T_i = [ \text{op}_1, \text{op}_2, \ldots, \text{op}_n ] \]
\[ \text{op}_1 = \text{start}_i \]
\[ \text{op}_i = r_i(x) \rightarrow v \mid w_i(x) v \rightarrow \text{ok} \mid \ldots \]
\[ \text{op}_n = \text{try}C_i \rightarrow C \mid \text{try}C_i \rightarrow A \mid \text{try}A_i \rightarrow A \mid \]
\[ r_i(x) \rightarrow A \mid w_i(x) v \rightarrow A \mid \ldots \]

Execution:

\[ \begin{array}{cccccc}
T_i & \text{start}_i & r_i(x) \rightarrow v & w_i(x) u & \rightarrow \text{ok} & \text{try}C_i \rightarrow C
\end{array} \]
Transaction abstraction

Transaction:

\[ T_i = [ \text{op}_1, \text{op}_2, \ldots, \text{op}_n ] \]
\[ \text{op}_1 = \text{start}_i \]
\[ \text{op}_i = r_i(x) \rightarrow v \mid w_i(x) \rightarrow \text{ok} \mid \ldots \]
\[ \text{op}_n = \text{tryC}_i \rightarrow C \mid \text{tryC}_i \rightarrow A \mid \text{tryA}_i \rightarrow A \mid \]
\[ r_i(x) \rightarrow A \mid w_i(x) \rightarrow A \mid \ldots \]

Execution:

\[ T_i \quad \text{start}_i \quad r_i(x) \rightarrow v \quad w_i(x)u \quad \rightarrow \text{ok} \quad \text{tryC}_i \rightarrow C \]

Conflict resolution (optimistic TM, increment of \( x \)):

\[ T_i \quad \text{start}_i \quad r_i(x) \rightarrow 0 \quad w_i(x)1 \rightarrow \text{ok} \quad \text{tryC}_i \rightarrow C \]

\[ T_j \quad \text{start}_j \quad r_j(x) \rightarrow 0 \quad w_j(x)1 \rightarrow A \]
Transaction abstraction

Transaction:

\[ T_i = [\text{op}_1, \text{op}_2, \ldots, \text{op}_n] \]

\[
\begin{align*}
\text{op}_1 & = \text{start}_i \\
\text{op}_i & = r_i(x) \rightarrow v \mid w_i(x) v \rightarrow \text{ok} \mid \ldots \\
\text{op}_n & = \text{tryC}_i \rightarrow C \mid \text{tryC}_i \rightarrow A \mid \text{tryA}_i \rightarrow A \mid r_i(x) \rightarrow A \mid w_i(x) v \rightarrow A \mid \ldots
\end{align*}
\]

Execution:

\[ T_i \quad \text{start}_i \quad r_i(x) \rightarrow v \quad w_i(x) u \quad \rightarrow \text{ok} \quad \text{tryC}_i \rightarrow C \]

(instantaneous) (delayed)

Conflict resolution (optimistic TM, increment of \(x\)):

\[ T_i \quad \text{start}_i \quad r_i(x) \rightarrow 0 \quad w_i(x) 1 \rightarrow \text{ok} \quad \text{tryC}_i \rightarrow C \]

\[ T_j \quad \text{start}_j \quad r_j(x) \rightarrow 0 \quad w_j(x) 1 \rightarrow A \quad \text{start}_{j'} \quad r_{j'}(x) \rightarrow 1 \quad w_{j'}(x) 2 \rightarrow \text{ok} \quad \text{tryC}_{j'} \rightarrow C \]
Problems with optimistic TM

Optimistic TM relies on aborts:

- low performance in high contention

![Diagram of optimistic TM with aborts](image-url)
Problems with optimistic TM

Optimistic TM relies on aborts:

- **low performance in high contention**

  ![Diagram of optimistic TM]

  

  - problems with irrevocable operations:
    - do not operate on shared data
    - have visible side effects
    - effects cannot be withdrawn (must be compensated)
    - examples: network communication, locks, system calls, I/O
Optimistic TM:

- run simultaneously in case there are no conflicts
- abort and retry if there are conflicts
Pessimistic TM

Optimistic TM: Pessimistic TM:

- run simultaneously in case there are no conflicts
- abort and retry if there are conflicts
Pessimistic TM

Optimistic TM: Pessimistic TM:
- defer execution to prevent conflict
- abort and retry if there are conflicts
Optimistic TM: Pessimistic TM:

- defer execution to prevent conflict
- avoid (most) forced aborts

- less waste of CPU (more waiting)
- performs better in high contention
- easy handling of irrevocable operations
Atomic RMI 2

A Java framework implementing distributed pessimistic TM
Implements the Optimized Supremum Versioning Algorithm

- completely distributed
- early release
- irrevocable operations
- rollback support
- fault tolerance

Backend: Java RMI
Atomic RMI 2 architecture
interface Resource extends Remote {
    @Access(Mode.READ)
    int get() throws RemoteException;

    @Access(Mode.WRITE)
    void set(int value) throws RemoteException;

    @Access(Mode.ANY)
    void increment() throws RemoteException;
}

class ResourceImpl implements Resource extends TransactionalUnicastRemoteObject {
    private int value = 0;

    void set(int value) {
        this.value = value;
    }

    int get() {
        return this.value;
    }

    void increment() {
        this.value += 1;
    }
}

class Server {
    public static void main(String[] args) throws Exception {
        Registry registry = LocateRegistry.createRegistry(9001);
        registry.bind("x", new ResourceImpl());
        registry.bind("y", new ResourceImpl());
    }
}
Registry registry = LocateRegistry.getRegistry(9001);

Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

int xv = x.get();
int yv = y.get();
x.set(xv + 2);
y.set(yv + 2);

transaction.commit();
Transaction example (Transactional)

Registry registry = LocateRegistry.getRegistry(9001);

Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start(
    new Transactional() {
        void atomic (Transaction transaction) {
            int xv = x.get();
            int yv = y.get();
            x.set(xv + 2);
            y.set(yv + 2);
        }
    }
);
OptSVA: basic versioning

$T_i$ starts:
- atomically get the next free version ticket for each object

$T_i$ executes a method on $x$:
- wait until $T_i$’s ticket matches $x$’s version counter
- execute the method

$T_i$ commits:
- wait until all transactions with lower versions for $x, y, z$ commit
- release each object by incrementing version counter
Transaction execution: basic versioning

\[ T_i \]

\[ \text{start}_i \]

\[ r_i(x) \rightarrow 0 \]

\[ r_i(y) \rightarrow 0 \]

\[ w_i(x) \rightarrow ok \]

\[ w_i(y) \rightarrow ok \]

\[ \text{tryC}_i \rightarrow C \]

get version for \( x \): 0

get version for \( y \): 0

wait until counter at \( x \) is set to 0

wait until counter at \( y \) is set to 0

release \( x, y \):

set counter at \( x \) to 1

set counter at \( y \) to 1
Transaction execution: basic versioning

\[
\begin{align*}
T_i & \quad \text{start}_i \quad r_i(x) \rightarrow 0 \quad r_i(y) \rightarrow 0 \quad w_i(x) \rightarrow \text{ok} \quad w_i(y) \rightarrow \text{ok} \quad \text{try}C_i \rightarrow C \\
& \quad \text{get version for } x: 0 \quad \text{wait until counter at } x \text{ is set to } 0 \quad \text{wait until counter at } y \text{ is set to } 0 \quad \text{release } x, y: \text{set counter at } x \text{ to } 1 \quad \text{set counter at } y \text{ to } 1
\end{align*}
\]

\[
\begin{align*}
T_j & \quad \text{start}_j \quad r_j(x) \\
& \quad \text{get version for } x: 1 \quad \text{wait until counter at } x \text{ is set to } 1 \quad \text{get version for } y: 1
\end{align*}
\]
Transaction execution: basic versioning

Transaction $T_i$:
- Start transaction $T_i$ ($start_i$)
- Get version for $x$: 0 (get version for $x$)
- Wait until counter at $x$ is set to 0 (wait until counter at $x$ is set to 0)
- Set counter at $x$ to 1 (release $x$)
- Get version for $y$: 0 (get version for $y$)
- Wait until counter at $y$ is set to 0 (wait until counter at $y$ is set to 0)
- Set counter at $y$ to 1 (release $y$)
- Try to commit ($tryC_i$)

Transaction $T_j$:
- Start transaction $T_j$ ($start_j$)
- Get version for $x$: 1 (get version for $x$)
- Wait until counter at $x$ is set to 1 (wait until counter at $x$ is set to 1)

Transaction $T_k$:
- Start transaction $T_k$ ($start_k$)
- Get version for $z$: 0 (get version for $z$)
- Wait until counter at $z$ is set to 0 (wait until counter at $z$ is set to 0)
- Set counter at $z$ to 1 (release $z$)
- Try to commit ($tryC_k$)

Try to commit ($tryC_i$) or ($tryC_k$):
- Release $x$, $y$: set counter at $x$ to 1
- Release $z$: set counter at $z$ to 1
Transaction example: upper bounds

Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"), 2);
Resource y = transaction.accesses(registry.lookup("y"), 2);

transaction.start();

int xv = x.get();
int yv = y.get();
x.set(xv + 2);
y.set(yv + 2);

transaction.commit();
OptSVA: early release

$T_i$ starts:
- atomically get the next unclaimed version ticket for each object

$T_i$ executes a method on $x$:
- wait until $T_i$’s ticket matches $x$’s version counter
- execute the method
- if execution counter reached declared upper bound, release $x$ by incrementing its version counter

$T_i$ commits:
- wait until all transactions with lower versions for $x, y, z$ commit
- release each object by incrementing its version counter
  (if necessary)
Transaction execution: early release

\[ T_i \]

\[ \text{start}_i \quad r_i(x) \rightarrow 0 \quad r_i(y) \rightarrow 0 \quad w_i(x) 2 \rightarrow \text{ok} \quad w_i(y) 2 \rightarrow \text{ok} \quad \text{try}C_i \rightarrow C \]

get version for \( x \): 0
get version for \( y \): 0
UB for \( x \): 2
UB for \( y \): 2

UB for \( x \) reached
release \( x \):
set counter at \( y \) to 1

UB for \( y \) reached
release \( y \):
set counter at \( y \) to 1
Transaction execution: early release

$T_i$
- $\text{start}_i$
- $r_i(x) \rightarrow 0$
- $r_i(y) \rightarrow 0$
- $w_i(x) \rightarrow \text{ok}$
- $w_i(y) \rightarrow \text{ok}$
- $\text{try}C_i \rightarrow C$

get version for $x$: 0
get version for $y$: 0
UB for $x$: 2
UB for $y$: 2

$T_j$
- $\text{start}_j$
- $r_j(x)$
- $\rightarrow 2$
- $r_j(y) \rightarrow 2$

get version for $x$: 1 wait until counter
get version for $y$: 1 at $x$ is set to 1

wait until counter at $y$ is set to 1
Deriving upper bounds

Upper bounds can be derived by static analysis (precompiler)
Supplemented by manual early release
Transaction example: manual early release

```java
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

int xv = x.get();
int yv = y.get();

if (xv < 10)
    x.set(xv + 2);
else
    transaction.release(x);

y.set(yv + 2);

transaction.commit();
```
Transaction example: manual abort

```java
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"), 2);
Resource y = transaction.accesses(registry.lookup("y"), 2);

transaction.start();

int xv = x.get();
int yv = y.get();

if (xv < 10)
    x.set(xv + 2);
else
    transaction.abort();

y.set(yv + 2);

transaction.commit();
```
OptSVA: abort support

\( T_i \) executes a method on \( x \):
- wait until \( T_i \)'s ticket matches \( x \)'s version counter
- if any declared object is invalidated: force abort
- if first operation on \( x \): make backup copy
- execute the method
- if reached declared upper bound for \( x \): release \( x \)

\( T_i \) commits:
- wait until all transactions with lower versions for \( x, y, z \) finish
- if any declared object is invalidated: force abort
- release each object (if necessary)

\( T_i \) aborts:
- wait until all transactions with lower versions for \( x, y, z \) finish
- invalidate modified objects and revert them from backup
- release each object (if necessary)
Transaction execution: abort

$T_i$

- $start_i$
- $r_i(x) \rightarrow 0$
- $r_i(y) \rightarrow 0$
- $w_i(x)2 \rightarrow ok$
- $tryA_i \rightarrow A$

make backup of $x$
make backup of $y$
release $x$

invalidate $x, y$
restore $x, y$ from backup
release $y$
Transaction execution: cascading abort

$T_i$  
$\text{start}_i$  
$r_i(x) \rightarrow 0$  
make backup of $x$  
$r_i(y) \rightarrow 0$  
make backup of $y$  
$w_i(x) \rightarrow 2 \rightarrow \text{ok}$  
release $x$  
$\text{tryA}_i \rightarrow A$  
invalidate $x, y$  
restore $x, y$ from backup  
release $y$

$T_j$  
$\text{start}_j$  
$r_j(x)$  
wait for $x$  
$\rightarrow 2$  
check if $x, y$ are invalidated  
make backup of $x$  
$r_j(y) \rightarrow A$  
wait until $y$ is released  
check if $x, y$ are invalidated  
force abort
Transaction example: prevent cascading aborts

Transaction transaction = new Transaction(true); // reluctant transaction

Resource x = transaction.accesses(registry.lookup("x"), 2);
Resource y = transaction.accesses(registry.lookup("y"), 2);

...
OptSVA: reluctant transactions

**Reluctant** $T_i$ executes a method on $x$:
- wait until all transactions with lower versions for $x$ finish
- if any declared object is invalidated: force abort
- if first operation on $x$: make backup copy
- execute the method
- if reached declared upper bound for $x$: release $x$

$T_i$ commits:
- wait until all transactions with lower versions for $x, y, z$ finish
- if any declared object is invalidated: force abort
- release each object (if necessary)

$T_i$ aborts:
- wait until all transactions with lower versions for $x, y, z$ finish
- invalidate modified objects and revert them from backup
- release each object (if necessary)
Transaction execution: prevented cascading aborts

$T_i$ $\text{start}_i \quad r_i(x) \to 0 \quad r_i(y) \to 0 \quad w_i(x) 2 \to \text{ok} \quad \text{try}A_i \to A$

- make backup of $x$
- make backup of $y$
- release $x$
- invalidate $x, y$
- restore $x, y$ from backup
- release $y$

$T_j$ $\text{start}_j \quad r_j(x)$

- reluctantly wait for $x$

$\to 0$
Example: a transaction treating objects as read-only

Transaction transaction = new Transaction();

Resource x = transaction.reads(registry.lookup("x"), 1);
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

int xv = x.get();
y.set(xv + 2);
System.out.println("new value: " + y.get());

transaction.commit();
OptSVA: a transaction treating objects as read-only

$T_i$ starts:
- (atomically) get the next unclaimed version ticket for each object
- cache all read-only objects in parallel
- once object $x$ is cached, release $x$

$T_i$ executes a read method on read-only object $x$:
- wait until $x$ object finished caching
- if any declared object is invalidated: force abort
- if first operation on $x$: make backup copy
- execute the method

$T_i$ commits:
- wait until all transactions with lower versions for $x, y, z$ commit
- if any declared object is invalidated: force abort
- increment version counter for each object (if necessary)
Transaction execution: read-only objects

$T_k$

release $x$

$T_i$

start caching $x$

wait for $x$

cache $x$

release $x$

start $i$

$r_i(x) ightarrow 0$

wait until $x$

is cached

$w_i(y) ightarrow ok$

$r_i(y) ightarrow 2$

try $C_i ightarrow C$
Transaction execution: read-only objects

\[ T_k \]
- release \( x \)

\[ T_i \]
- \( \text{start}_i \)
  - start caching \( x \)
  - wait for \( x \)
  - cache \( x \)
  - release \( x \)

\[ r_i(x) \rightarrow 0 \]
- wait until \( x \) is cached

\[ r_i(y) \rightarrow 0 \]
- \( w_i(y) \rightarrow \text{ok} \)

\[ r_i(y) \rightarrow 2 \]
- \( r_j(y) \rightarrow 2 \)
- \( \text{tryC}_i \rightarrow C \)

\[ T_j \]
- \( \text{start}_j \)
  - \( r_j(x) \rightarrow 0 \)
  - \( r_j(y) \rightarrow 2 \)
Transaction example: write optimizations

Transaction transaction = new Transaction();

Resource x = transaction.reads(registry.lookup("x"), 1);
Resource y = transaction.accesses(registry.lookup("y"),
1 /*write*/, 1 /*read*/);

transaction.start();

int xv = x.get();
y.set(xv + 2);
System.out.println("new value: " + y.get());

transaction.commit();
OptSVA: first write

\( T_i \) executes a write method on \( x \):

- if first operation of any kind on \( x \): create log
- execute the method on log (if available)

\( T_i \) executes other methods on \( x \):

- wait until \( T_i \)'s ticket matches \( x \)'s version counter
- if log for \( x \) has operations: apply log to \( x \) and discard the log
- execute the method

\( T_i \) commits:

- wait until all transactions with lower versions for \( x, y, z \) commit
- if any declared object is invalidated: force abort
- apply log to \( x \) (if necessary)
- increment version counter for each object (if necessary)
OptSVA: first write, last write

$T_i$ executes a write method on $x$:
- if first operation of any kind on $x$: create log
- execute the method on log (if available)
- if last write on $x$:
  - if log is empty: release $x$
  - otherwise: wait for $x$, apply log, cache $x$, release $x$ (in parallel)

$T_i$ executes other methods on $x$:
- wait until $T_i$’s ticket matches $x$’s version counter
- if log for $x$ has operations: apply log to $x$ and discard the log
- execute the method

$T_i$ commits:
- wait until all transactions with lower versions for $x, y, z$ commit
- if any declared object is invalidated: force abort
- apply log to $x$ (if necessary)
- increment version counter for each object (if necessary)
Transaction execution: write operations

$T_i$ 
- start caching $x$
- wait for $x$
- cache $x$
- release $x$

$T_k$ 
- release $x$
- release $y$

$w_i(y) \rightarrow ok$
- create log
- execute on log
- wait for $y$
- cache $y$
- release $y$

$r_i(x) \rightarrow 0$
- wait until $x$
- is cached
- ok
- create log
- execute on log
- wait for $y$
- cache $y$
- release $y$

$tryC_i \rightarrow C$
Transaction execution: write operations

\[ T_k \]
- release \( x \)

\[ T_i \]
- \( \text{start}_i \)
  - start caching \( x \)
  - wait for \( x \)
  - cache \( x \)
  - release \( x \)
- \( r_i(x) \rightarrow 0 \)
- wait until \( x \) is cached

\[ w_i(y) \rightarrow \text{ok} \]
- create log
- execute on log
- wait for \( y \)
- apply log to \( y \)
- cache \( y \)
- release \( y \)
- \( r_i(y) \rightarrow 2 \)

\[ T_j \]
- \( \text{start}_j \)
  - \( r_j(x) \rightarrow 0 \)

\[ \text{try} C_i \rightarrow C \]
- \( r_j(y) \rightarrow 2 \)

Transactions for Actors?

Actors: $a_1, a_2, \ldots$

Transaction:

\[ T_i = \left[ \text{op}_1, \text{op}_2, \ldots, \text{op}_n \right] \]

\[ \text{op}_1 = \text{start}_i \]

\[ \text{op}_i = \text{send}(a_j)[r_i(x)] \rightarrow \text{ok} | \text{recv}[p] \rightarrow v | \]

\[ \text{send}(a_j)[w_i(x)v] \rightarrow \text{ok} | \ldots \]

\[ \text{op}_n = \text{tryC}_i \rightarrow C | \text{tryC}_i \rightarrow A | \text{tryA}_i \rightarrow A | \]

\[ \text{send}(a_j)[r_i(x)] \rightarrow A | \text{recv}[p] \rightarrow A | \]

\[ \text{send}(a_j)[w_i(x)v] \rightarrow A | \ldots \]
Transactions for Actors?

Actors: $a_1, a_2, ...$

Transaction:

$T_i = \begin{bmatrix} op_1, & op_2, & ..., & op_n \end{bmatrix}$

$op_1 = start_i$

$op_i = send(a_j)[r_i(x)] \rightarrow ok \mid recv[p] \rightarrow v \mid$

$send(a_j)[w_i(x)v] \rightarrow ok \mid ...$

$op_n = tryC_i \rightarrow C \mid tryC_i \rightarrow A \mid tryA_i \rightarrow A \mid$

$send(a_j)[r_i(x)] \rightarrow A \mid recv[p] \rightarrow A \mid$

$send(a_j)[w_i(x)v] \rightarrow A \mid ...$

Pros and cons:

- allow for consistent behavior on multiple nodes
- introduce dependencies between asynchronous messages
TM safety property primer

**Serializability:**

The outcome of all committed transactions is equivalent to the outcome of some serial execution of these transactions.

**Real-time order:**

Transactions executing one after another cannot be re-arranged to justify their correctness.

**Opacity:**

- Serializability and real-time order
- Transactions only view the effects of committed transactions

**Last-use opacity:**

- Serializability and real-time order
- Committed transactions only view the effects of committed transactions, but
- Committed and uncommitted transactions only view the effects of the final modifications in transactions
Atomic RMI 2 (OptSVA) properties

- Serializable and real-time order
- If transactions don’t invoke manual aborts:
  - opaque from programmers’ point of view
  - irrevocable operations always correct
- Otherwise:
  - last-use opaque
  - irrevocable operations in reluctant transactions always correct
Evaluation

Frameworks:
- Atomic RMI (SVA)
- Atomic RMI 2 (OptSVA)
- Fine grained locking (variants of 2PL):
  - exclusion locks
  - R/W locks
  - single global lock
- HyFlow2 (TFA) – optimistic distributed TM

Environment:
- $10 \times 2 \times$ quad-core Intel Xeon L3260 (2.83 GHz), 4 GB RAM
- OpenSUSE 13.1
- JRE (64 bit): Oracle 1.8.0_05-b13, Hotspot 25.5-b02

Benchmark:
- Distributed version of EigenBench
Throughput

Short transactions, 5 objects per node:

- 80% reads
- 50% reads
- 20% reads

Short transactions, 10 objects per node:

- 80% reads
- 50% reads
- 20% reads
Throughput

Long transactions, 10 objects per node

![Throughput Graphs](image)

Scalability

![Scalability Graphs](image)

36 / 39
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

for (i = 0; i < n; i++) {
    x.increment();
    y.increment();
}

transaction.release(x);
transaction.release(y);

// local operations

transaction.commit();
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"), n);
Resource y = transaction.accesses(registry.lookup("y"), n);

transaction.start();

for (i = 0; i < n; i++) {
    x.increment(); // x released before calling y
    y.increment();
}

// local operations

transaction.commit();
Transaction transaction = new Transaction();

Resource[] resources = new Resource[n];
resources[0] = transaction.accesses(registry.lookup("r1"), 2);
resources[1] = transaction.accesses(registry.lookup("r2"), 2);
// ...
resources[n] = transaction.accesses(registry.lookup("rn"), 2);

transaction.start();

for (i = 0; i < n; i++) {
    if (resources[i].get() == 0) {
        resources[i].set(1);
        break;
    } else
        transaction.release(resources[i]); // released with no delay
}

transaction.commit();