

# Practical Concurrent and Parallel Programming 9

Riko Jacob  
IT University of Copenhagen

Friday 2018-11-02

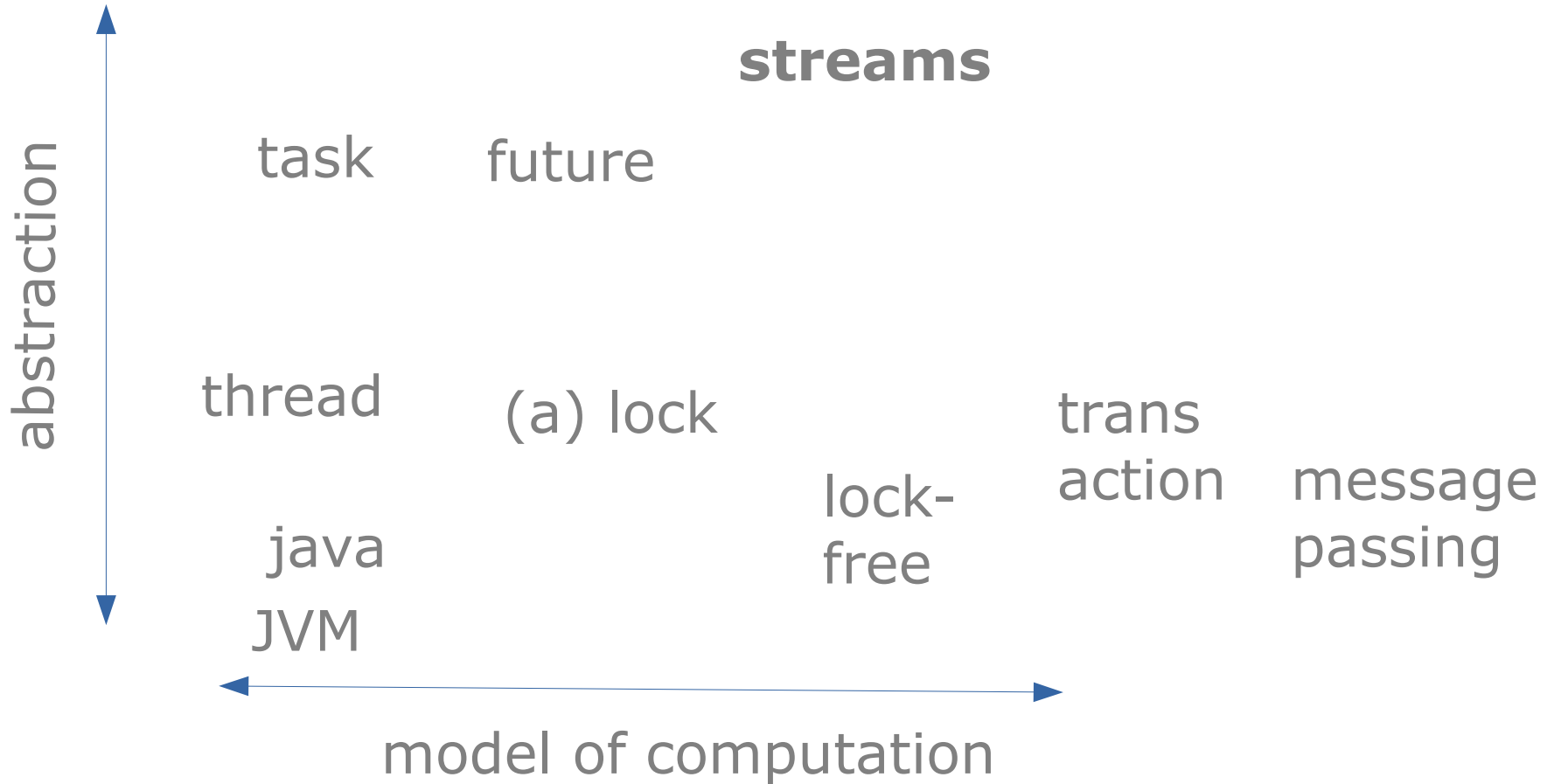
# Plan for today

- What's wrong with lock-based atomicity
- Transactional memory STM, Multiverse library
- A transactional bank account
- Transactional blocking queue
- Composing atomic operations
  - transfer from one queue to another
  - choose first available item from two queues
- Philosophical transactions
- Other languages with transactional memory
- Hardware support for transactional memory
- **NB: Course evaluation starting**

**correctness**

**measure**

**streams**



# Transactional memory

- Based on transactions, as in databases
- Transactions are composable
  - unlike lock-based concurrency control
- Easy to implement blocking
  - no `wait` and `notifyAll` or semaphore trickery
- Easy to implement blocking choice
  - eg. get first item from any of two blocking queues
- Typically *optimistic*
  - automatically very scalable read-parallelism
  - unlike *pessimistic* locks
- No deadlocks and usually no livelocks
- Fine grained 'locking' expressed

# Transactions

- Know from databases since 1981 (Jim Gray)
- Proposed for programming languages 1986
  - (In a functional programming conference)
- Became popular again around 2004
  - due to Harris, Marlow, Peyton-Jones, Herlihy
  - Haskell, Clojure, Scala, ... and Java Multiverse
- A transaction must be
  - **A**tomic: if one part fails, the entire transaction fails
  - **C**onsistent: maps a valid state to a valid state
  - **I**solated: A transaction does not see the effect of any other transaction while running
  - (But *not* **D**urable, as in databases)

# Current State STM

- Activity peaked some time ago
- Important concept:
  - Connection to databases
  - Implemented in hardware
  - Perhaps comes back again
- I am not aware of a current actively developed java library (there perhaps is scalaSTM), hence little support, no community, sometimes tricky to get things to work

# Difficulties with lock-based atomicity

- Transfer money from account `ac1` to `ac2`
  - No help that each account operation is atomic
  - Can lock both, but then there is deadlock risk
- Transfer an item from queue `bq1` to `bq2`
  - No help that each queue operation is atomic
  - Locking both, nobody can put and take; deadlock
- Get an item from either queue `bq1` or `bq2`
  - (when both queues are blocking)
  - Should block if both empty
  - But just calling `bq1.take()` may block forever even if there is an available item in `bq2`

# Transactions make this trivial

- Transfer amount from account ac1 to ac2:

Pseudo-code

```
atomic {  
    ac1.deposit(-amount);  
    ac2.deposit(+amount);  
}
```

- Transfer one item from queue bq1 to bq2:

```
atomic {  
    T item = bq1.take();  
    bq2.put(item);  
}
```

- Take item from queue bq1 if any, else bq2:

```
atomic {  
    return bq1.take();  
} orElse {  
    return bq2.take();  
}
```



# Transactional account

## Pseudo-code

```
class Account {  
    private long balance = 0;  
    public void deposit(final long amount) {  
        atomic {  
            balance += amount;  
        }  
    }  
    public long get() {  
        atomic {  
            return balance;  
        }  
    }  
    public void transfer(Account that, final long amount) {  
        final Account thisAccount = this, thatAccount = that;  
        atomic {  
            thisAccount.deposit(-amount);  
            thatAccount.deposit(+amount);  
        }  
    }  
} }
```

Composite transaction  
without deadlock risk

# Transactional memory in Java

- Multiverse Java library 0.7 from April 2012
  - Seems comprehensive and well-implemented
  - Little documentation apart from API docs
  - ... and those API docs are quite cryptic
- A transaction must be wrapped in
  - `new Runnable() { ... }` if returning nothing
  - `new Callable<T>() { ... }` if returning a T value
  - or just a lambda `() -> { ... }` in either case
- Runs on unmodified JVM
  - Thus is often slower than locks/volatile/CAS/...
- To compile and run (or use `myj.sh`):

```
$ javac -cp ~/lib/multiverse-core-0.7.0.jar TestAccounts.java
$ java -cp ~/lib/multiverse-core-0.7.0.jar:. TestAccounts
```

# Transactional account, Multiverse

```
class Account {
    private final TxnLong balance = newTxnLong(0);
    public void deposit(final long amount) {
        atomic(() -> balance.set(balance.get() + amount));
    }

    public long get() {
        return atomic(() -> balance.get());
    }

    public void transfer(Account that, final long amount) {
        final Account thisAccount = this, thatAccount = that;
        atomic(() -> {
            thisAccount.deposit(-amount);
            thatAccount.deposit(+amount);
        });
    }
}
```

Composite transaction  
without deadlock risk

# Consistent reads

- Auditor computes balance sum during transfer

```
long sum = atomic(() -> account1.get() + account2.get());  
System.out.println(sum);
```

- Must read both balances in same transaction
  - Does not work to use a transaction for each reading
- Should print the sum only outside transaction
  - After the transaction committed
  - Otherwise risk of printing multiple times...

# How do transactions work?

- A transaction txn typically keeps
  - Read Set: all variables read by the transaction
  - Write Set: *local copy* of variables it has updated
- When trying to commit, check that
  - no variable in Read Set or Write Set has been updated by another transaction
  - if OK, write Write Set to global memory, *commit*
  - otherwise, discard Write Set and *restart* txn again
- So the Runnable may be called many times!
- How long to wait before trying again?
  - Exponential backoff: wait `rnd.nextInt(2)`, `rnd.nextInt(4)`, `rnd.nextInt(8)`, ...
  - Should prevent transactions from colliding forever

# Nested transactions

- By default, an `atomic` within an `atomic` reuses the outer transaction: So if the inner fails, the outer one fails too
- Several other possibilities, see `org.multiverse.api.PropagationLevel`
  - Default is `PropagationLevel.Requires`: if there is a transaction already, use that; else create one

# Multiverse transactional references

- Only transactional variables are tracked
  - TxnRef<T>, a transactional reference to a T value
  - TxnInteger, a transactional `int`
  - TxnLong, a transactional `long`
  - TxnBoolean, a transactional `boolean`
  - TxnDouble, a transactional `double`
- Methods, used in a transaction, inside `atomic`
  - `get()`, to read the reference
  - `set(value)`, to write the reference
- Several other methods, eg
  - `getAndLock(lockMode)`, for more pessimism
  - `await(v)`, block until value is `v`

# Plan for today

- What's wrong with lock-based atomicity
- Transactional memory STM, Multiverse library
- A transactional bank account
- **Transactional blocking queue**
- **Composing atomic operations**
  - transfer from one queue to another
  - choose first available item from two queues
- Philosophical transactions
- Other languages with transactional memory
- Hardware support for transactional memory



# Lock-based bounded queue (wk 8)

```
class SemaphoreBoundedQueue <T> implements BoundedQueue<T> {
    private final Semaphore availableItems, availableSpaces;
    private final T[] items;
    private int tail = 0, head = 0;

    public void put(T item) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(item);
        availableItems.release();
    }

    private synchronized void doInsert(T item) {
        items[tail] = item;
        tail = (tail + 1) % items.length;
    }

    public T take() throws InterruptedException { ... }
    ...
}
```

Use semaphore to block until room for new item

Use lock for atomicity

# Transactional blocking queue

```
class StmBoundedQueue<T> implements BoundedQueue<T> {  
    private int availableItems, availableSpaces;  
    private final T[] items;  
    private int head = 0, tail = 0;  
  
    public void put(T item) {           // at tail  
        atomic {  
            if (availableSpaces == 0)  
                retry();  
            else {  
                availableSpaces--;  
                items[tail] = item;  
                tail = (tail + 1) % items.length;  
                availableItems++;  
            }  
        }  
    }  
  
    public T take() {  
        ... availableSpaces++; ...  
    }  
}
```

Atomic  
action

Use `retry()`  
to block

# Real code, using Multiverse library

```
class StmBoundedQueue<T> implements BoundedQueue<T> {
    private final TxnInteger availableItems, availableSpaces;
    private final TxnRef<T>[] items;
    private final TxnInteger head, tail;

    public void put(T item) { // at tail
        atomic(() -> {
            if (availableSpaces.get() == 0)
                retry();
            else {
                availableSpaces.set(availableSpaces.get() + 1);
                items[tail.get()].set(item);
                tail.set((tail.get() + 1) % items.length);
                availableItems.set(availableItems.get() + 1);
            }
        });
    }

    public T take() {
        ... availableSpaces.set(...); ...
    }
}
```

Atomic  
action

Use `retry()`  
to block

# How does blocking work?

- When a transaction executes `retry()` ...
  - The Read Set says what variables have been read
  - No point in restarting the transaction until one of these variables have been updated by other thread
- Hence NOT a busy-wait loop
  - but automatic version of `wait` and `notifyAll`
  - or automatic version of `acquire` on Semaphore
- Often works out of the box, idiot-proof
- Must distinguish:
  - *restart* of transaction because could not commit
    - exponential backoff, random sleep before restart
  - an explicit `retry()` request for blocking
    - waits passively in a queue for Read Set to change

# Atomic transfer between queues

```
static <T> void transferFromTo(BoundedQueue<T> from,
                              BoundedQueue<T> to)
{
    atomic(() -> {
        T item = from.take();
        to.put(item);
    });
}
```

stm/TestStmQueues.java

- A direct translation from the pseudo-code
- Can hardly be wrong

# Blocking until some item available


```
private static <T> T takeOne(BoundedQueue<T> bq1,
                             BoundedQueue<T> bq2)
throws Exception {
    return atomic( () -> {
        if( !bq1.isEmpty() ) return bq1.take();
        if( !bq2.isEmpty() ) return bq2.take();
        retry();
        return null; // unreachable
    } );
}
```

stm/TestStmQueues.java

- Specific to BoundedQueues:
  - test if take will succeed
- Can hardly be wrong

# Blocking until some item available

```
static <T> T takeOne(BoundedQueue<T> bq1,  
                   BoundedQueue<T> bq2) throws Exception  
{  
    return atomic(() -> bq1.take(),  
                 () -> bq2.take());  
}
```



The diagram shows two callables, `bq1.take()` and `bq2.take()`, highlighted with red boxes. A red callout bubble labeled "Do this" points to `bq1.take()`, and another red callout bubble labeled "or else that" points to `bq2.take()`.

- If `bq1.take()` fails, try instead `bq2.take()`
- Implemented using general `myOrElse` method
  - taking as arguments two Callables
- Does not quite work like this in multiverse

# Implementing method myOrElse

```
static <T> T myOrElse(Callable<T> either, Callable<T> orelse)
    throws Exception
{
    return atomic(() -> {
        try {
            return either.call();
        } catch (org.multiverse.api.exceptions.RetryError retry) {
            return orelse.call();
        }
    });
}
```

stm/TestStmQueues.java

- Exposes Multiverse's internal machinery
  - `retry()` is implemented by throwing an exception
- Hand-made implementation
  - Because Multiverse's `OrElseBlock` seems faulty... (unit/integration test seems to fail)



# Plan for today

- What's wrong with lock-based atomicity
- Transactional memory STM, Multiverse library
- A transactional bank account
- Transactional blocking queue
- Composing atomic operations
  - transfer from one queue to another
  - choose first available item from two queues
- **Philosophical transactions**
- Other languages with transactional memory
- Hardware support for transactional memory

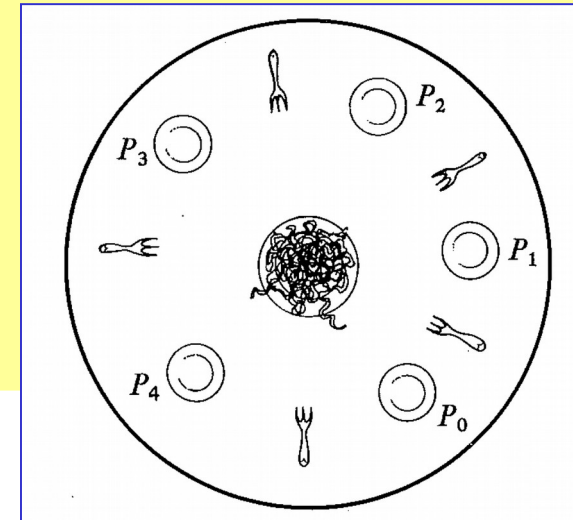
# Philosophical Transactions

```

class Philosopher implements Runnable {
    private final Fork[] forks;
    private final int place;
    public void run() {
        while (true) {
            int left = place, right = (place+1) % forks.length;
            synchronized (forks[left]) {
                synchronized (forks[right]) {
                    System.out.print(place + " "); // Eat
                }
            }
            try { Thread.sleep(10); } // Think
            catch (InterruptedException exn) { }
        }
    }
}

```

Exclusive  
use of forks



- Lock-based philosopher (wk 8)
  - Likely to deadlock in this version

# TxnBooleans as Forks A

```
class Philosopher implements Runnable {
    private final TxnBoolean[] forks;
    private final int place;
    public void run() {
        while (true) {
            final int left = place, right = (place+1) % forks.length;
            atomic(() -> {
                if (!forks[left].get() && !forks[right].get()) {
                    forks[left].set(true);
                    forks[right].set(true);
                } else
                    retry();
            });
            System.out.printf("%d ", place); // Eat
            atomic(() -> {
                forks[left].set(false);
                forks[right].set(false);
            });
            try { Thread.sleep(10); } // Think
            catch (InterruptedException exn) { }
        }
    }
}
```

Exclusive  
use of forks

Release  
forks

# TxnBooleans as Forks B

```
class Philosopher implements Runnable {
    private final TxnBoolean[] forks;
    private final int place;
    public void run() {
        while (true) {
            final int left = place, right = (place+1) % forks.length;
            atomic(() -> {
                forks[left].await(false);
                forks[left].set(true);
                forks[right].await(false);
                forks[right].set(true);
            });
            System.out.printf("%d ", place); // Eat
            atomic(() -> {
                forks[left].set(false);
                forks[right].set(false);
            });
            try { Thread.sleep(10); } // Think
            catch (InterruptedException exn) { }
        }
    }
}
```

Exclusive  
use of forks

Release  
forks

# Transaction subtleties

- What is wrong with this Philosopher?
  - Variant of B that “eats” inside the transaction

```
public void run() {  
    while (true) {  
        final int left = place, right = (place+1) % forks.length;  
        atomic(() -> {  
            forks[left].await(false);  
            forks[left].set(true);  
            forks[right].await(false);  
            forks[right].set(true);  
            System.out.printf("%d ", place); // 1  
            forks[left].set(false);  
            forks[right].set(false);  
        });  
        try { Thread.sleep(10); } // Th  
        catch (InterruptedException exn) { }  
    }  
}
```

**BAD**

Transaction has its own view of the world until commit

Other transactions may have taken all the forks!

# Optimism and multiple universes

- A transaction has its own copy of data (forks)
- At commit, it checks that data it used is valid
  - if so, writes the updated data to common memory
  - otherwise throws away the data, and restarts
- Each transaction works in its own “universe”
  - until it successfully commits
- This allows higher concurrency
  - especially when write conflicts are rare
  - but means that a Philosopher cannot know it has exclusive use of a fork until transaction commit
- Transactions + optimism = multiple universes
- No I/O or other side effects in transactions!

# Lazy vs. Eager

- **Lazy** commit strategy:
  - Keep everything in transaction's universe until commit
  - Conflict resolution at commit time
  - Keep redo log of what should be redone on retry
- **Eager** commit strategy:
  - Commit changes upon making them
  - Detect conflicts as transaction proceeds
  - Conflict resolution happens at multiple places
  - Keep an undo log of things that need to be reverted on conflict

# Lazy vs. Eager

- **Lazy:**
  - Rollback is faster (just drop local data)
  - Slower commits (commits everything at once!)
  - Memory not inconsistent on crashes
- **Eager:**
  - Rollback is slower
  - Conflicts detected earlier
  - Memory may be inconsistent on crashes



# Optimistic Concurrency and Game Theory

- View transactions as competing entities
- Transactions have knowledge of system
- E.g. long-running transactions get priority
- Why should we keep transactions short?
- Conversely, we also want fairness
- Paper by Eidenbenz and Wattenhofer
- Conclusion: Any deterministic policy can be gamed/exploited
- Optimistic, **cooperative** concurrency (next week)

# Pessimistic Concurrency and Game Theory

- Same principle applies to pessimistic concurrency
- Why should I let go of a lock?
- Keep holding on to object associated with lock
- Security concern: Locking on **this**
- Any code with a reference to your object can block everyone else
- Use a private lock object instead

# Hints and warnings

- Transactions should be short
  - When a long transaction finally tries to commit, it is likely to have been undermined by a short one
  - ... and must abort, and a lot of work is wasted
  - ... and it restarts, so this happens again and again
- For example, concurrent hash map
  - short: `put`, `putIfAbsent`, `remove`
  - long: `reallocateBuckets` – not clear it will ever succeed when others `put` at the same time
- Some STM implementations avoid aborting the transaction that has done most work
  - Many design tradeoffs

# Some languages with transactions

- Haskell – in GHC implementation
  - TVar T, similar to TxnRef<T>, TxnInteger, ...
- Scala – ScalaSTM, on Java platform
  - Ref[T], similar to TxnRef<T>, TxnInteger, ...
- Clojure – on Java platform
  - (ref x), similar to TxnRef<T>, TxnInteger, ...
- C, C++ – future standards proposals
- Java – via Multiverse library
  - Creator Peter Ventjeer is on ScalaSTM team too
- Java – DeuceSTM, other research prototypes
- And probably many more ...

# Transactional memory in perspective

- Works best in a mostly immutable context
  - eg functional programming: Haskell, Clojure, Scala
- Mixes badly with side effects, input-output
- Requires transactional (immutable) collection classes and so on
- Some loss of performance in software-only TM
- Still unclear how to best implement it
- Some think it will remain a toy, Cascaval 2008
  - ... **but** they use C/C++, too much mutable data
- Multicore hardware support would help
  - can be added to cache coherence (MESI) protocols

# Hardware support for transactions

- Eg Intel TSX for Haswell CPUs, since 2013
  - New XBEGIN, XEND, XABORT instructions
  - <https://software.intel.com/sites/default/files/m/9/2/3/41604>
- Could be used by future JVMs, .NET/CLI, ...
- Uses core's cache for transaction's updates
- Extend cache coherence protocol (MESI, wk 7)
  - Messages say when another core writes data
  - On commit, write cached updates back to RAM
  - On abort, invalidate cache, do not write to RAM
- Limitations:
  - Limited cache size, ...

# This week

- Reading
  - Herlihy and Shavit sections 18.1-18.2
  - Harris et al: *Composable memory transactions*
  - Cascaval et al: *STM, Why is it only a research toy*
  - Eidenbenz and Wattenhofer: Good programming in transactional memory Game theory meets multicore architecture
- Exercises
  - Show you can use transactional memory to implement histogram and concurrent hashmap
- Read before next week
  - Goetz et al chapter 15 (lock free data structures)