

# Tackling Concurrency With STM

Mark Volkmann  
mark@ociweb.com  
10/22/09



## Two Flavors of Concurrency

- Divide and conquer
  - divide data into subsets and process it by running the same code on each subset concurrently
  - MapReduce solutions focus on this flavor
- Coexist
  - execute different code concurrently and allow it to safely access the same data
  - Transactional Memory (TM) focuses on this flavor



"Really? — my people always say *multiply* and conquer."



# Concurrency Options

- Locks

- popular in C, C++ and Java

- Actors

- popular in Erlang, Haskell and Scala

- Transactional Memory

- popular in Clojure and Haskell
- can be implemented in hardware or software



3

STM

## Locks

- Pros

- explicit control over when locks are acquired and released allows optimal solutions
- developer familiarity
- supported by many programming languages

- Cons

- which locks need to be acquired?
- what order to prevent deadlock?
- variables for which locks should be acquired can be accessed even when no locks or the wrong locks are acquired
- must remember to release locks in error recovery code
- correctly synchronized methods don't compose
- pessimistic approach reduces concurrency



4

STM

# Actors ...

- Software entities that execute as separate processes or threads
- Only use data passed to them via asynchronous messages
  - can retain data for use in subsequent processing
- When a message is received by an actor it can
  - create new actors
  - send messages to other actors
  - decide how it will handle subsequent messages



5

STM

# ... Actors

- Pros
  - since no memory is shared between actors, data access doesn't need synchronization
- Cons
  - some messages may be large since they are the only way to share data
  - no general mechanism for coordinating activities of multiple actors (i.e. transactions)
    - may want to send messages to multiple actors within a txn and guarantee that either all messages are processed successfully or all the actors involved rollback changes to their retained state



6

STM

# TM

- Provides ACID txn characteristics for memory
  - **atomic** - all changes commit or all changes roll back; changes appear to happen at a single moment in time
  - **consistent** - operate on a snapshot of memory using newest values at beginning of txn
  - **isolated** - changes are only visible to other threads after commit
  - **not durable** - changes are lost if software crashes or hardware fails
- Demarcating txns
  - `atomic { ... }` in literature, `(dosync ...)` in Clojure, `atomically` in Haskell
- Nested txns join the outermost txn - txns compose



# TM Retries

- If two txns overlap in time and they attempt to write the same memory then one of them will discard their changes and retry from their beginning
  - in some implementations, reads can also trigger retries
    - could retry txn A if txn B modifies memory that was read by txn A since it may have made decisions based on the value it read
    - reads only trigger a retry in Clojure when the history list of a Ref doesn't contain a value committed before the txn began (a fault) ... will make sense when history lists are described later



# TM Pros

- Familiar concept from database world
- Optimistic, not pessimistic
  - provides more opportunities for concurrency, especially for txns that only read data
- Easier to write correct code than using locks
  - don't have to determine which locks need to be acquired and order
  - only have to identify sections of code that require a consistent view of the data it reads and writes
- Implementations can guarantee ...
  - no deadlocks, livelocks or race conditions (see next slide)



# Concurrency Issues



- Deadlock
  - concurrent threads cannot proceed because they are each waiting on a resource for which another has acquired exclusive access
- Livelock
  - concurrent threads are performing work (not blocked), but cannot complete due to something other threads have done or not done
- Race condition
  - the outcome of a thread may be wrong due to the timing of changes to shared state made by other concurrent threads
- None of these can occur in Clojure STM



# TM Cons

- Potential for many retries resulting in wasted work
- Overhead imposed by txn bookkeeping
  - more detail on this later
- Need to avoid side-effects
  - such as I/O
  - since txns may retry any number of times
- Tool support is currently lacking
  - for learning which memory locations experienced write conflicts
  - for learning how often each txn retried and why



Clojure provides a solution using Agents.



# On Your Honor

- TM works best in programming languages that provide a special kind of mutable variable that can only be modified in a txn
  - Ref in Clojure
  - TVar in Haskell
- Otherwise developers are on their honor to use TM correctly
  - similar to developers being on their honor to use locks correctly



# Garbage Collection Analogy

- From Dan Grossman

- associate professor at University of Washington
- see paper “The Transactional Memory / Garbage Collection Analogy”
- listen to Software Engineering Radio podcast #68

- Can GC be replaced by TM in these statements?

- “Many used to think GC was too slow without hardware.”
- “Many used to think GC was about to take over, decades before it did.”
- “Many used to think we needed a back door for when GC was too approximate.”



# More on Concurrency Options

- Shared state

- locks provide manual management of shared state
- TM provides semi-automated management of shared state
- actors avoid shared state; Can it be avoided? Are transactions needed?

- Baseball analogy

- locks never attempt to steal a base since they might get caught - pessimistic
- TM does and just returns to the previous base and retries if caught - optimistic
- actors utilize a coach actor to send a message to the runner actor to request a steal attempt;  
the umpire actor sends a message to runner actor to indicate whether they are safe



# Persistent Data Structures

- Immutable data structures such as lists, vectors, sets and hash maps
- Can efficiently create new ones from existing ones
- New ones share memory with old ones
  - okay since they can't be modified
- Safe for concurrent access
- An opinion
  - saying a language is functional, but doesn't have persistent data structures is like saying a language is OO, but doesn't support polymorphism



# STM Implementations

- Part of language
  - Clojure, Haskell, Perl 6
- As a library
  - C, C++, C#, Common Lisp, Java, MUMPS, OCaml, Python, Scheme, Smalltalk
- Implementations vary greatly
  - difficult to make general statements about STM characteristics such as memory usage and performance





# Clojure STM Implementation

- The remaining slides describe the Clojure STM implementation, starting with concepts related to it
- Valuable even if you don't use Clojure
  - to understand how at least one STM implementation works under the covers
  - to enable reasoning about performance characteristics
  - to encourage implementations for other languages



# Transaction Creation

- `(dosync body)`
  - passed expressions that form the body of a txn



# Reference Types

- Mutable references to immutable objects
- Four kinds
  - Var
  - Atom
  - Agent
  - Ref

“coordinated” here means managed by txns

- Modification characteristics

	Uncoordinated	Coordinated
Synchronous	Var, Atom	Ref
Asynchronous	Agent	



# Vars

- Can have a root value that is shared by all threads
  - `(def name value)`
- Can have thread-specific values
  - `(binding [name value ...] body)`
- Often used for constants and configuration variables such as `*out*`
- Reads (dereferences) are atomic - `@name`
- Writes are also atomic
  - with `binding`

“atomic” here means that multiple threads can access them without synchronization





# Atoms

- Have a single value that is shared across threads

- `(def name (atom value))`

`def` isn't the only way to bind a reference type to a name; can pass as a function argument and use `let` and `binding`

- Reads (dereferences) are atomic - `@name`

- Writes are also atomic

- `(reset! name value)`

- `(compare-and-set! name current-value new-value)`

- `(swap! name update-function arg*)`

Function names that end with ! typically indicate that the function either modifies its arguments or has some other side effect.



# Agents

- Have a single value that is shared across threads

- `(def name (agent value))`

- Reads (dereferences) are atomic - `@name`

- Writes are asynchronous

- by sending a function (called an "action" in this context) to the Agent that is executed in a different thread

- action is passed the current value of the Agent and any additional, optional arguments

- return value of action becomes new value of the Agent, atomically

- only one action at a time is executed for a given Agent, which prevents concurrent updates to an Agent

- actions sent to Agents inside an STM txn are held until the txn commits

`(send name action arg*)`  
uses fixed size thread pool

`(send-off name action arg*)`  
uses variable size thread pool

useful for causing side effects after a txn



# Refs

- Have a single value that is shared across threads
  - `(def name (ref value))`
- Reads (dereferences) are atomic - `@name`
  - while reads are not required to be performed inside an STM txn, doing so provides access to a consistent snapshot of the set of Refs accessed inside the txn
- Writes must be performed inside an STM txn
- Refs are the only type managed by STM



# In-Txn and Committed Values

- In-txn values of Refs
  - maintained by each txn
  - only visible to code running in the txn
  - committed at end of txn if successful
  - cleared after each txn try
- Committed values
  - maintained by each Ref in a circular linked-list (`tvals` field)
  - each has a commit “timestamp” (`point` field in `TVa1` objects)
    - ordered long values obtained by calling `incrementAndGet` on an `AtomicLong`

in `java.util.concurrent.atomic` package



## Changing a Ref ...

- Three ways
  - all must be performed inside an STM txn
- `(ref-set ref new-value)`
- `(alter ref function arg*)`
  - invokes the function, passing it the current Ref value and the arguments, changes the Ref to refer to the function return value, and returns the new value

For both `ref-set` and `alter`, if another txn sets an in-txn value for the same Ref, one of the txns will retry. If another txn commits a change to the same Ref, this txn will retry.



## ... Changing a Ref

- `(commute ref function arg*)`
  - similar to `alter`
  - use when the order of changes across concurrent txns doesn't matter and use of in-txn values won't produce incorrect results
  - the txn will not retry if another txn has modified the Ref since the current txn try began
  - during txn commit, all commute functions invoked in the txn are called again using latest committed Ref values
  - example uses
    - computing min, max and average of values in a collection
    - adding objects to a collection



# Validators

- `(set-validator! ref function)`
- Function is invoked when the value of a given reference type is about to be modified
- Passed the proposed new value of the reference type
- Reject the change by returning false or throwing any kind of exception



# Watch Functions

- `(add-watch ref key function)`
- Function is invoked if the reference type may have changed
  - passed the key, the reference type, its old value and its new value
- A reference type can have any number of watch functions
  - each must be added with a unique key





# Watcher Agents

- `(add-watcher ref send-type agent function)`
- Function is “sent” to the Agent  
in a different thread  
if the reference type may have changed
  - passed current value of Agent and the reference type
- *send-type* determines the thread pool used  
to obtain the thread in which the function runs
  - `:send` for fixed or `:send-off` for variable size thread pool



# Clojure STM Design and Implementation

- Design is based on
  - multi-version concurrency control
  - snapshot isolation
- Mostly implemented in Java now
  - but work to re-implement in Clojure is underway



# Multi-Version Concurrency Control

- Uses timestamps or increasing txn ids
- Maintains multiple versions of objects with commit “timestamps”
- Txns read the most recent versions of objects that were committed before the txn start
- When attempting a write, if another txn has modified the object since the current txn started then the txn discards its changes and retries
- Reads are never blocked

Clojure uses increasing ids for txn starts, try starts and commits. They are obtained by calling the `incrementAndGet` method on a Java `AtomicLong`. This uses a compare and swap (CAS) approach.



# Snapshot Isolation

- Txns appear to operate on a snapshot of memory taken at the start of the txn
- At the end of the txn, changes are committed only if no other txns have committed changes to the values to be committed since the txn began







# Write Skew

- Occurs when

- concurrent txns read common sets of data, make changes to different data within that set, and there are constraints on the data

avoided by STM implementations that use "read tracking"; a performance tradeoff

- Example

- data is the number of dogs and number of cats owned by a family
- constraint is the sum of dogs and cats that can be owned by a family  $\leq 3$
- John and his wife Mary own one dog and one cat
- John adopts a new dog while Mary simultaneously adopts a new cat (2 txns)
- both txns read the number of dogs and cats they own, but they modify different data
- neither txn violates the constraint, so both succeed which results in a constraint violation



# Clojure Solution To Write Skew



- `(ensure ref)`
- Prevents other txns from modifying the Ref
  - calling txn can modify the Ref unless another txn has also called **ensure** on it
- Must be called inside a txn
- For previous example
  - both txns should **ensure** the Ref they don't plan to modify since its value, together with that of the Ref being modified, is used in a constraint



## Main STM Classes ...

- **LockingTransaction**

- one of these objects per thread (**ThreadLocal**)
  - doesn't make sense to have more than one active txn per thread
  - avoids having to recreate bookkeeping collections referred to by **LockingTransaction** fields every time a new txn starts
- **dosync** macro calls **sync** macro which calls **LockingTransaction runInTransaction** static method

- **Ref**

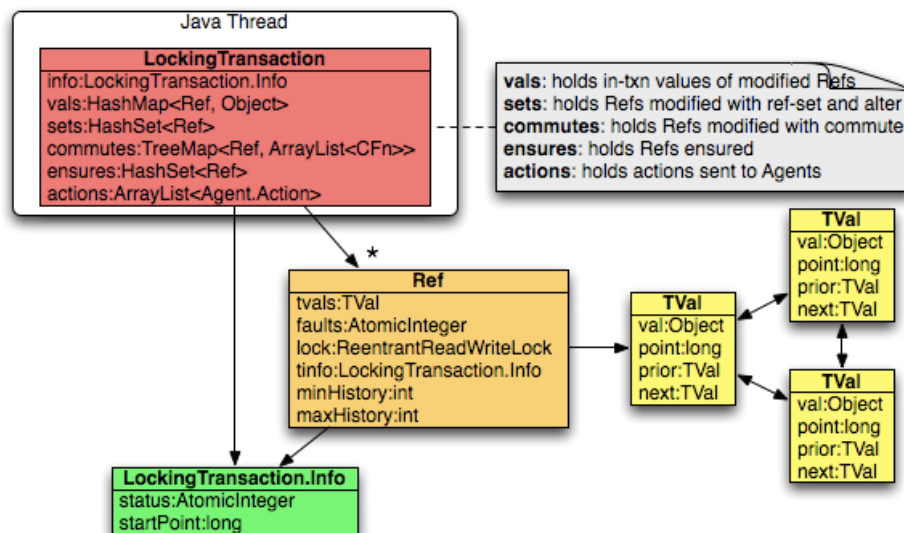
- one of these objects per mutable reference to be managed by STM

- **LockingTransaction.Info**

- part of lock-free strategy to mark Refs as having an uncommitted change



## ... Main STM Classes



# Transaction Status

- **RUNNING**

- executing code in the body

- **COMMITTING**

- finished executing code in the body; committing changes to Refs, if any

- **RETRY**

- will attempt a retry, but hasn't started the next try yet

- **KILLED**

- has been "barged" by another txn (explained later)

- **COMMITTED**

- finished committing changes to Refs, if any; transaction completed (even if read-only)

stored in `status` field of `LockingTransaction.Info` objects which are referred to from the `info` field of `LockingTransaction` objects and the `tinfo` field of `Ref` objects



# Ref History List

- Each Ref maintains a list of recently committed values

- using `TVa1` objects
- length is controlled by `minHistory` and `maxHistory`
  - default to 0 and 10, but can be customized for each Ref

- When a change to a Ref is committed

- a new node is added to its history list if
  - history list length < `minHistory` OR
  - a fault (described on next slide) has occurred since the last commit of the Ref and history list length < `maxHistory`
- otherwise the oldest node is modified to become the newest node

With `minHistory` set to zero, the history list of each Ref grows according to how the Ref is actually used. If a Ref never has a fault, its history list never needs to grow.



# Faults

- A fault occur when

- there is an attempt to read a Ref in a txn AND
- there is no in-txn value in the txn AND
- all values in the history list for the Ref were committed after the txn started

means it hasn't been modified in the txn

- Faults cause

- a txn to retry
- the history chain for the Ref to grow, unless **maxHistory** has been reached



# Write Conflicts

- A write conflict occurs when

- txn A attempts to modify a Ref
- txn B has already modified the same Ref, but hasn't yet committed the change
  - it has an in-txn value

- When this occurs

- txn A will attempt to barge txn B (see next slide)



# Barging

- Determines whether txn A should be allowed to continue while txn B retries
- Three conditions must be met
  - txn A must have been running for at least 1/100th of a second (**BARGE\_WAIT\_NANOS**)
    - if it just started then it may as well be the one to retry
  - txn A started before txn B (favors older txns)
  - txn B has a status of RUNNING and can be changed to KILLED
    - won't interrupt a txn that is in the process of committing
- Otherwise txn A retries and txn B continues



# What Causes a Retry?

- A Ref is read in a txn and one of the following is true
  - another txn barged the current one (status isn't RUNNING)
  - the Ref has no in-txn value and fault occurs
 

the next try will have a new start time, so will look for newer Ref values
- **ref-set** or **alter** is called on a Ref and one of the following is true
  - can't set the **info** field of the Ref to indicate that it was modified by the current txn because a write lock cannot be obtained for the Ref because another thread holds a read or write lock for it
  - another txn changed the Ref, but hasn't committed yet and an attempt to barge it fails
- **ref-set**, **alter**, **commute** or **ensure** is called on a Ref and another txn barged the current one (status isn't RUNNING)
- During commit, a txn is preparing to rerun a commute function on a Ref, but another txn made an in-txn change to the same Ref and barging it fails



# Retries

- When a txn retires
  - all read and write locks currently held are released
  - all in-txn values of modified Refs are discarded
  - many collections associated with the txn are cleared
    - `ensures`, `notify`, `actions`, `vals`, `sets` and `commutes`
  - the txn status is changed to RETRY
  - execution continues at the beginning of the txn body
- Limited number of retries
  - 10,000 (`RETRY_LIMIT`)
  - seems arbitrary and cannot be configured



# Locks in STM ...

in `java.util.concurrent.locks` package

- ReentrantReadWriteLock
  - each instance manages the read and write locks for a single Ref
  - any number of concurrent txns can hold a read lock for a Ref  
OR one txn can hold the write lock for a Ref
  - locks are only held briefly, not for the duration of a txn
    - unless `ensure` is called on a Ref in which case a read lock is held until the Ref is modified in the txn or the txn ends



## ... Locks in STM ...

- Lock-free strategy
  - the `txinfo` field in `Ref` objects is set to a `LockingTransaction.Info` object to mark them as having an in-txn value for a given txn
  - alternative to having `LockingTransaction` objects lock `Ref` objects for the duration of a transaction



## ... Locks in STM

- Read locks are acquired to
  - read (dereference) a `Ref` in a txn that has no in-txn value - released when finished
  - ensure a `Ref` - not released until txn modifies the `Ref` or commits
  - commute a `Ref` - only held until newest value is copied into map of in-txn values; released before commute function is executed
- Write locks are acquired to
  - mark a `Ref` as having an in-txn value in a specific txn - released after marking
  - commit changes to `Refs` - released when commit completes



# STM Overhead

- Despite overhead introduced by STM, it can still be faster than using locks because it is optimistic instead of pessimistic
  - more opportunities for concurrency instead of blocking
- Specific sources of overhead incurred during reads and writes of Refs are described next



# STM Ref Read Overhead

- read = dereference
- Unless a Ref was modified in a txn, giving it an in-txn value, walk history list to find newest value before txn started
  - if no other txn committed a change to the Ref since the current one started, the first value in the chain is used and the lookup is fast







# STM Ref Write Overhead ...

- Verify that a txn is running
  - if not, throw `IllegalStateException`
- Verify no commute
  - once `commute` has been called on a Ref, `ref-set` and `alter` cannot be called on it within the same txn
  - since `commute` functions are called a second time during the commit, calls to `ref-set` and `alter` after `commute` wouldn't have a lasting affect



# ... STM Ref Write Overhead ...

- On the first write of a Ref within the txn
  - add the Ref to the set of Refs modified by the txn
    - used during commit
  - mark the Ref as having an in-txn value for the txn
    - if `ensure` was called on the Ref earlier in the txn, release the read lock for the Ref
    - attempt to acquire a write lock for the Ref and retry if unsuccessful
    - if another txn has committed a change to the Ref since the current txn began, retry
    - if another txn made an in-txn change to the Ref since the current txn began, attempt to barge it and retry if unsuccessful
    - mark Ref as being "locked" by current txn (sets its `tinfo` field to refer to current txn)
    - release the write lock for the Ref





## ... STM Ref Write Overhead

- On each write of a Ref,  
including the first in a txn
  - add new value to the map of in-txn values for the txn
    - subsequent reads of the Ref inside the txn  
get the value from this map instead of the history chain



## STM Commit Overhead ...

- Change txn status from **RUNNING** to **COMMITTING**
- Rerun **commute** functions called in txn
  - must acquire a write lock for each commuted Ref
- Acquire write locks for all Refs modified in txn  
so there can be no readers
- Validate
  - call all the validate functions registered on modified Refs  
and retry if any disapprove of a change
- Update history list for each modified Ref
  - add new node or modify an existing one (explained on slide 38)
- For each modified Ref that has at least one registered watcher,  
create a **Notify** object that describes the change





## ... STM Commit Overhead

- Change txn status from **COMMITTING** to **COMMITTED**
- Release all write locks acquired previously
- Release all read locks still held from calls to **ensure**
- Clear contents of several txn collection fields in preparation for next use
  - **ensures**, **vals**, **sets** and **commutes**
- If commit was successful, notify all registered Ref watchers using data in **Notify** objects
- Dispatch all actions sent to Agents in txn
- Clear contents of more txn collection fields (**notify** and **actions**)
- Return value of last expression in txn body



## Conclusions

- STM makes writing correct concurrent code much easier than using locks!
- Clojure isn't the only programming language with STM support
- For more details on Clojure, see <http://ociweb.com/mark/clojure/>
- For more details on STM and the Clojure implementation, see <http://ociweb.com/mark/stm/>

