Clojure Concurrency Constructs, Part Two

CSCI 5828: Foundations of Software Engineering
Lecture 18 — 10/22/2015

Goals

- Cover the material presented in Chapter 4, of our concurrency textbook
  - In particular, the material presented in Day 2 and 3 of that chapter

- New Concepts: Agents, refs and Software Transactional Memory
- Conclude with a few in-depth examples

Agents

- Agents are a type of mutable variable in Clojure
  - Designed for **asynchronous** updates
    - As opposed to atoms which provide **uncoordinated synchronous** updates and refs (discussed next) which provided **coordinated synchronous** updates
    - Useful for tasks that can proceed independently of each other with minimal coordination (update this value but I don't care when you do it)
- Like an atom, an agent encapsulates a reference to a single value
  - The value can be of any Clojure type
  - If you want to know what the value is, you `deref` the agent

Using agents

- Creating an agent: use the `agent` function
  - `(def counter (agent 0))`
- Update the value of an agent with the `send` function
  - `(send <agent> <function> <args>)`

- The `send` function takes an agent, a function, and that function's arguments
  - `send` returns immediately. At some point in the future, in a separate thread, the function and its arguments will be applied to the value of the agent. The new value of the agent will then be available via `deref`
- Examples
  - `(send counter inc)` and `(send counter + 200)`

Contention?

- What happens if multiple calls to send occur on the same agent?
  - The answer: the calls get serialized and are applied only once in order
    - No need to worry about race conditions, the value will eventually be the result of all calls to update the value
    - However, you can’t predict ahead of time what that order will be

Synchronization? (I)

- What happens if you send a long running function via send?
  - Answer: you can't predict when the new value will become available
    - `(defn wait-then-inc [i] (do (Thread/sleep 2000) (inc i)))`
    - `(def counter (agent 0))`
      - `(send counter wait-then-inc)`
        - after this call, value of counter is 0 for about 2 seconds
          - then it turns to 1
- Note: I couldn't get the book's anonymous function for performing "sleep then increment" to work, so I defined my own separate `wait-then-inc` function

Synchronization? (II)

- What happens if you want to wait until an agent's value has actually updated?
  - Use the await function to block the current thread until all sends have finished executing
    - `(send counter wait-then-inc)` ; returns immediately
      - `(await counter)` ; blocks until previous send has been applied
- The `send` function makes use of a common set of threads that Clojure allocates when it boots up a repl session
  - If one of your functions will take a long time to complete, you can use the `send-off` function to spin up a new thread, keeping the common thread pool available for use by other functions (such as the reducers library)

Error Handling

- Agents can have a validator associated with them to ensure that some property is always true of their value
  - `(def non-negative (agent 1 :validator (fn [new-val] (>= new-val 0))))`
- This call would create an agent with an initial value of 1 and checks to make sure that its value never goes negative
  - If a send causes the value to go negative, the agent enters an error state and will no longer accept send requests
- If an agent enters an error state, you can find out why with agent-error
  - `(agent-error non-negative)` ; returns the error object or nil
- You can then restart the agent, so it can accept new send requests
  - `(restart-agents non-negative 20)`

Updated counter example

- I transformed my counter example from Lecture 12 to use agents rather than atoms.
    - The program still works in that the final value of the counter is 40 and the log contains 40 entries, but you can see the asynchronous nature of agents in the log messages
    - ["Thread 2 updating atom: 0 to 1" "Thread 1 updating atom: 0 to 1" "Thread 2 updating atom: 0 to 1" "Thread 1 updating atom: 0 to 1" "Thread 2 updating atom: 0 to 1" "Thread 1 updating atom: 1 to 2" "Thread 1 updating atom: 1 to 2" "Thread 2 updating atom: 1 to 2" "Thread 1 updating atom: 1 to 2" "Thread 2 updating atom: 1 to 2" "Thread 1 updating atom: 1 to 2" "Thread 2 updating atom: 1 to 2" "Thread 1 updating atom: 1 to 2" "Thread 1 updating atom: 1 to 2" "Thread 2 updating atom: 1 to 2" "Thread 1 updating atom: 2 to 3" "Thread 2 updating atom: 2 to 3" "Thread 1 updating atom: 2 to 3" "Thread 1 updating atom: 2 to 3" "Thread 2 updating atom: 2 to 3" "Thread 1 updating atom: 2 to 3" "Thread 2 updating atom: 2 to 3" "Thread 1 updating atom: 2 to 3" "Thread 2 updating atom: 2 to 3" "Thread 1 updating atom: 2 to 3" "Thread 2 updating atom: 3 to 4" "Thread 1 updating atom: 3 to 4" "Thread 2 updating atom: 3 to 4" "Thread 1 updating atom: 3 to 4" "Thread 2 updating atom: 3 to 4" "Thread 1 updating atom: 3 to 4" "Thread 2 updating atom: 3 to 4" "Thread 1 updating atom: 3 to 4" "Thread 2 updating atom: 3 to 4" "Thread 1 updating atom: 3 to 4" "Thread 2 updating atom: 4 to 5" "Thread 2 updating atom: 4 to 5" "Thread 2 updating atom: 5 to 6”]
- Due to the asynchronous updates of both counter and log, the 40th log message saw only the fifth update of the counter variable!

Refs and Software Transactional Memory (STM)

- The final type of mutable variable in Clojure is known as a ref
- Like atoms and agents, a ref encapsulates a reference to a single value
  - The value can be of any Clojure type
- To create a ref: (def counter (ref 0))
- To reference the value: (deref counter) or @counter
- Unlike atoms and agents, changes to a set of refs can happen in an atomic fashion => either all changes are completed successfully or all fail
  - In order to do that, changes to one or more refs **MUST** happen inside of a transaction that is managed by Clojure's software transactional memory

STM Transactions

- STM transactions are atomic, consistent, and isolated
  - **Atomic**: from the point of view of other transactions, all the changes made in a transaction take place or none of them do
  - **Consistent**: a transaction can have a validator associated with it; if validation fails, all updates are rolled back
  - **Isolated**: Multiple transactions can run concurrently; the effect of these transactions however can not be distinguished from running each of them sequentially
- These three properties are the first three of the famous ACID properties supported by most relational databases; the missing property, *durability*, is missing because these transactions happen in memory; they are not (automatically) persisted to disk. Changes made in a transaction can be lost if a power failure or crash occurs

Updating refs

- If you try to update a ref outside of a transaction, it will fail
- `(ref-set counter 42)` => IllegalStateException No transaction running
- `(alter counter inc)` => IllegalStateException No transaction running

- To create a transaction, you need to use `dosync`.
  - `(dosync (ref-set counter 42))` => @ref == 42
  - `(dosync (alter counter inc))` => @ref == 43

One last update to the counter example (!)

- I updated the counter example to use refs and STM
  - The program works as expected and THIS TIME because we're using transactions, the log's output finally matches our expectations
  - ["Thread 2 updating atom: 0 to 1" "Thread 1 updating atom: 1 to 2" "Thread 2 updating atom: 2 to 3" "Thread 1 updating atom: 3 to 4" "Thread 2 updating atom: 4 to 5" "Thread 2 updating atom: 5 to 6" "Thread 1 updating atom: 6 to 7" "Thread 2 updating atom: 7 to 8" "Thread 1 updating atom: 8 to 9" "Thread 2 updating atom: 9 to 10" "Thread 1 updating atom: 10 to 11" "Thread 2 updating atom: 11 to 12" "Thread 1 updating atom: 12 to 13" "Thread 2 updating atom: 13 to 14" "Thread 1 updating atom: 14 to 15" "Thread 2 updating atom: 15 to 16" "Thread 1 updating atom: 16 to 17" "Thread 2 updating atom: 17 to 18" "Thread 1 updating atom: 18 to 19" "Thread 2 updating atom: 19 to 20" "Thread 1 updating atom: 20 to 21" "Thread 2 updating atom: 21 to 22" "Thread 1 updating atom: 22 to 23" "Thread 2 updating atom: 23 to 24" "Thread 1 updating atom: 24 to 25" "Thread 2 updating atom: 25 to 26" "Thread 1 updating atom: 26 to 27" "Thread 2 updating atom: 27 to 28" "Thread 1 updating atom: 28 to 29" "Thread 2 updating atom: 29 to 30" "Thread 1 updating atom: 30 to 31" "Thread 2 updating atom: 31 to 32" "Thread 1 updating atom: 32 to 33" "Thread 2 updating atom: 33 to 34" "Thread 1 updating atom: 34 to 35" "Thread 2 updating atom: 35 to 36" "Thread 1 updating atom: 36 to 37" "Thread 2 updating atom: 37 to 38" "Thread 1 updating atom: 38 to 39" "Thread 1 updating atom: 39 to 40"]
- The updates to both log and counter are finally coordinated!

Book's Example: Transfer Money

- The book has an example that demonstrates how STM can detect a conflict between two transactions running in parallel
  - If it does, it will allow one transaction to commit and rollback any attempted changes by the other; it then runs the second transaction again
    - All of this is handled automatically by Clojure and STM at run-time!

- To see this in action, consider having to transfer money between two bank accounts
  - We only want this transaction to succeed if both accounts are updated correctly!

Bank Account Example (I)

- Assume that accounts are modeled as refs
- A function to perform the transfer then looks like this
  - `(defn transfer [from to amount]`
    - `(dosync`
      - `(alter from - amount)`
      - `(alter to + amount)))`
- This code subtracts amount from the source account and adds it to the destination account
  - These operations occur within a transaction created by `dosync`.

Bank Account Example (II)

- A typical use scenario
  - `(def checking (ref 1000)) => 1000`
  - `(def savings (ref 2000)) => 2000`
  - `(transfer savings checking 100) => 1100`
  - `@checking => 1100`
  - `@savings => 1900`

- If we had lots of transfers occurring at once, STM may have to retry several of them that make conflicting changes with other transactions

- To detect this, we are going to make use of a different transfer function and two mutable variables: one agent and one atom

Bank Account Example (III)

- The new set-up
  - `(def attempts (atom 0))`
  - `(def transfers (agent 0))`
  - `(defn transfer [from to amount]`
    - `(dosync`
      - `(swap! attempts inc)`
      - `(send transfers inc)`
      - `(alter from - amount)`
      - `(alter to + amount)))`
- This code has one function to increment an atom inside the transaction and one function to increment an agent
  - It turns out that the exclamation point at the end of `swap!` is there to remind developers that this function is **NOT safe to use within a transaction** — it generates a side effect that will build up over time as a transaction is retried; `send`, it turns out, is "transaction aware"

Bank Account Example (IV)

- To stress test this new version of transfer, we'll do the following
  - `(defn stress-thread [from to iterations amount]`
    - `(Thread. #(dotimes [_ iterations] (transfer from to amount))))`
  - This function does the following
    - Creates a new Java thread and passes an anonymous function to run
    - That function calls `dotimes` performing a transfer for a particular amount, a specified number of times
    - The _ indicates that we do not need to use a variable name from the iteration within the body of `dotimes`.

Bank Account Example (V)

- Our “main” routine looks like this

```
(def checking (ref 10000))
(def savings (ref 20000))
(defn -main [& args]
  (println "Before: Checking =" @checking " Savings =" @savings
  (let [t1 (stress-thread checking savings 100 100)
        t2 (stress-thread savings checking 200 100)]
    (.start t1)
    (.start t2)
    (.join t1)
    (.join t2))
  (await transfers)
  (println "Attempts: " @attempts)
  (println "Transfers: " @transfers)
  (println "After: Checking =" @checking " Savings =" @savings))
```

- Initialize the bank accounts, start two threads to create 300 transfers, wait for them to finish, wait for our agent to update, print results

Bank Account Example (VI)

- The results?
  - Before: Checking = 10000 Savings = 20000
  - Attempts: 733
  - Transfers: 300
  - After: Checking = 20000 Savings = 10000
- The transfers were basically designed to swap the balances between the accounts; 10K moved from checking to savings; 20K went the other way
  - The agent recorded 733 attempts to perform transfers; 300 of them are all that were needed; 433 were the results of retries
  - The update to the agent however is “transaction aware”; it was only sent when a transaction completed successfully. Hence it’s final value was 300

The Clojure Way

- We've now seen all the ways in which Clojure can help you design concurrent systems
  - immutable values and persistent data structures
  - functions
  - loops via recursion or recur
  - lazy sequences
  - map, reduce, filter, etc.
  - reducers library
  - atoms, agents, refs
  - future, promise, deliver
  - Java's own support for threads
- Our book now delves into a more in-depth example

Dining Philosophers (Strike Back)

- The book looks at some implementations of the dining philosophers problem
  - The first uses software transactional memory
    - similar to the Java-based solution with condition variables
  - The second uses atoms without relying on software transactional memory

- Let's look at the STM-based version
  - First, we need five philosophers, each will be a ref storing either :thinking or :eating. They start in the :thinking state
- `(def philosophers (into [] (repeatedly 5 #(ref :thinking))))`
  - This creates a vector with five refs; each ref has the value :thinking

Philosophers STM (I)

- Thinking and Eating are simulated with calls to sleep
  - `(defn think [] (Thread/sleep (rand 1000)))`
  - `(defn eat [] (Thread/sleep (rand 1000)))`
- At run-time there will be a thread associated with each philosopher
- That thread will update the value of its associated ref from :thinking to :eating and vice versa
  - It can only update itself from :thinking to :eating when it is sure that the two philosophers on either side of it are :thinking

Philosophers STM (II)

- The philosopher thread looks like this

```
- (defn philosopher-thread [n]
  - (Thread.
    - #(let [philosopher (philosophers n)
      - left (philosophers (mod (- n 1) 5))
      - right (philosophers (mod (+ n 1) 5))]
      - (while true
        - (think)
        - (when (claim-chopsticks philosopher left right)
          - (eat)
          - (release-chopsticks philosopher)))))))
```

- This thread takes an id which allows the philosopher to locate its ref and the refs next to it; It then loops forever, thinking and attempting to eat. It can only eat if it can claim its chopsticks

Philosophers STM (III)

- To release our chopsticks, we simply update our ref to :thinking
  - This has to happen in a transaction
  - `(defn release-chopsticks [philosopher]`
    - `(dosync (ref-set philosopher :thinking)))`
- To claim our chopsticks, we check (in a transaction) the status of our neighbors and update to eating if they are thinking
  - `(defn claim-chopsticks [philosopher left right]`
    - `(dosync`
      - `(when (and (= @left :thinking) (= @right :thinking))`
        - `(ref-set philosopher :eating))))`
- Looks good! Let's see what happens

Philosophers STM (IV)

- ...
- Philosopher 4 is eating.
- Philosopher 3 is eating.
- ...
- Philosopher 3 is thinking.
- Philosopher 4 is thinking.
- ...
- WHOOPS! Two adjacent philosophers were allowed to eat together!
  - (The horror!) What went wrong?

Reads Inside of a Transaction

- Our current claim-chopsticks simply reads the values of left and right
  - **(= @left :thinking) (= @right :thinking)**
- Since it doesn't attempt to change these values, Clojure's software transactional memory is able to allow other transactions running concurrently to change them
  - As a result, we can be looking at stale data, when we decide to eat
- To tell the STM that "reads" have to be consistent as well as "writes" we must use the function ensure; if we access the value of a ref with ensure, then the STM will make sure that no other transaction changes that value when this transaction reads that value; the relevant portion of claim-chopsticks is now:
  - **(and (= (ensure left) :thinking) (= (ensure right) :thinking))**
- Now, if we run the program, it will run forever without error

Dining Philosophers without STM

- Let's compare the previous implementation with one that makes use of atoms
  - We can no longer rely on transactions to coordinate changes across multiple refs
    - as a result, we'll have to do more work to keep things consistent
- The first change we need to make is to use a single atom to hold a vector of values (either :eating or :thinking) rather than a vector of refs
  - `(def philosophers (atom (into [] (repeat 5 :thinking))))`
- The reason for this is that atoms can only guarantee the synchronous update of a single value
  - we can't coordinate changes to multiple atoms, so we don't even try

Philosophers Atoms (I)

- The value in our atom captures the state of all philosophers at once
  - [:eating :thinking :eating :thinking :thinking]
- `release-chopsticks` is now updated to `release-chopsticks!` to change an entry in this vector from :eating to :thinking
- ```
  (defn release-chopsticks! [philosopher]
    (swap! philosophers assoc philosopher :thinking))
  ```
- We've seen `assoc` used with maps. With vectors, they take an index (in this case the symbol philosopher) plus the new value and returns a new vector with the updated value
- swap! is used to actually update the atom to the new state
  ```

Philosophers Atoms (II)

- The philosophers-thread remains exactly the same except for one minor change in which we refer to philosophers by index into the philosophers vector rather than by ref
  - See your book for details
- The last change involves claim-chopsticks!
  - (defn claim-chopsticks! [philosopher left right]
    - (swap! philosophers
      - (fn [ps]
        - (if (and (= (ps left) :thinking) (= (ps right) :thinking))
          - (assoc ps philosopher :eating)
          - ps)))
    - (= (@philosophers philosopher) :eating))

- We call swap! and switch our value to :eating as long as our left/right are thinking; if not, we return the current value, so that the atom is unchanged

Philosophers Atoms (III)

- The results?
  - It runs fine with no deadlock and
    - at least two philosophers can eat at any given time
- So, what's the difference between this approach and the STM approach?
  - Nothing
  - Both solve the problem with similar clarity
    - It comes down to style
      - If you like managing multiple refs via transactions, use STM
      - If you like having all mutable state in a single compound data structure, use atoms

Summary

- Clojure provides a wide range of tools to help with the design of concurrent software systems
  - Once you embrace the functional approach and get comfortable with the syntax, you have
    - a variety of “concurrency-aware” mutable variables
      - agents, atoms, refs
    - each with various concurrency guarantees
- You can also make use of
  - the reducers library, for in-memory, compute-bound tasks
  - futures, promises, and deliver for inter-thread communication