Design Approaches for Concurrent Systems

CSCI 5828: Foundations of Software Engineering
Lecture 08 — 02/09/2012

Goals

- Discuss material in Chapter 3 of our concurrency textbook
    - Design Approaches for Concurrent Systems
        - Dealing with State
        - Shared Mutable Design
        - Isolated Mutable Design
        - Purely Immutable Design
            - Persistent/Immutable Data Structures
    - Selecting an approach

Dealing with State (I)

- In any concurrent program, you have three choices when dealing with state
  - shared mutability
  - isolated mutability
  - pure immutability
- The book describes each approach using a simple example
  - Imagine a room with a whiteboard and a group of people
  - The task is to get the people to tally up the total number of years the people have been working in industry
  - In each case, let's imagine that the whiteboard is an instance variable and each person in the room is a separate thread

Dealing with State (II)

- Shared mutability is the most familiar for programmers
  - We have a set of objects that have methods and instance variables
  - We have a set of threads that are accessing these objects in some fashion
  - As a result, we have the potential for two or more threads to be accessing the same object at the same time and potentially updating the same instance variable; this can lead to interference
- Example
  - Someone writes “0” on the whiteboard and then asks everyone to come up and update the total with their years of experience
    - People jump up and… form a line and update the value one at a time
    - Analogous to a synchronized increment() method; no true concurrency

Dealing with State (III)

- Isolated mutability is less familiar but straightforward to understand
    - For each mutable variable in a concurrent program, you design the program such that only one thread has access to it
    - As a result, the behavior of that variable matches what we would expect in a single threaded program, even though multiple threads may be running
- Previous Examples
    - The “total” variable in the Concurrent portfolio calculator is an example of isolated mutability
    - The widgets in the ConsiderateWindow example from Lecture 4 are examples of isolated mutability; they only ever get updated by the GUI thread
- Current Example: each member texts their years of experience to one person who updates the total as the texts arrive; highly concurrent; queue does serialization

Dealing with State (IV)

- Pure immutability is even less familiar to most developers and the hardest to understand
  - The basic idea is that you design your program such that a variable **can be assigned at most one value** and **that value never changes**
    - It then **doesn't matter** if that variable is accessible to multiple threads
    - It's value will never change on any of them. It is read only
- For those developers comfortable with shared mutability and isolated mutability, this seems like an impossible goal
  - How can you write a program (at least an interesting program) that does not change the state of at least one variable?!

Immutable Values (I)

- Most programmers are familiar with immutable values
  - String foo = “Software Engineering”
  - String bar = foo;
- foo and bar point at a value (a string) that will never change.
  - foo = foo + “ is cool!”
- Foo now points at a **new** string; the previous value did not change
  - Instead, the original string was copied and the copy was combined with the “ is cool!” part to create a new string value (which itself will never change)
    - You can’t change the value; if you call foo.replace(), it returns a NEW string

Immutable Values (II)

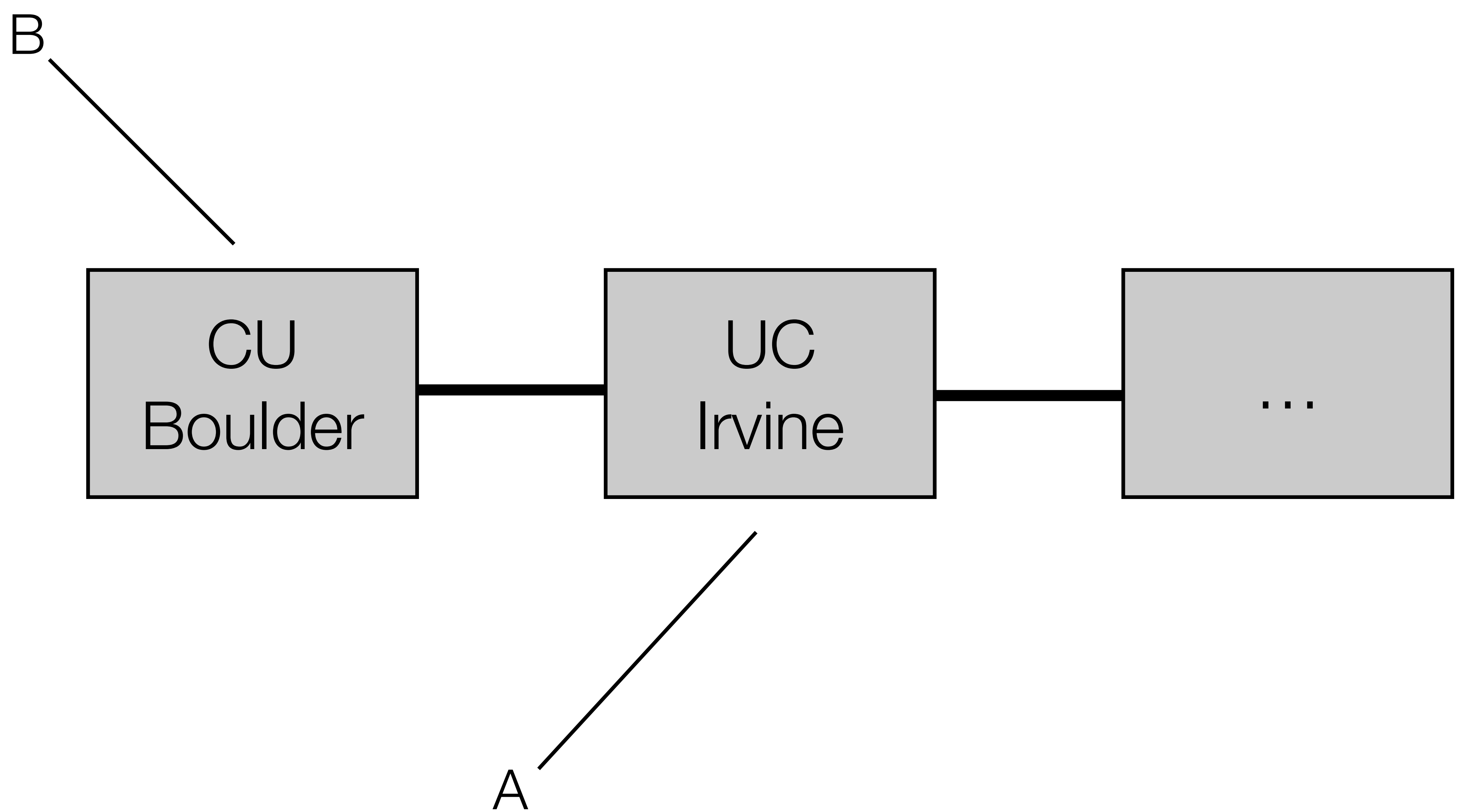
- Consider extending this idea to other types of values
  - Imagine you created a Car class and designed it such that each value is immutable;
    - Car a = new Car(“blue”, 4, “Chevy”, “Volt”); #blue, Chevy Volt (4 wheels)
    - Car b = a.setColor(“red”); #red, Chevy Volt (4 wheels)
  - If Car’s values were immutable, then setColor() would return a completely new instance of Car that copied all of the values from a except for “blue”, set the color to “red”, and returned the copy
  - The value pointed at by a would still exist and that particular value would never change; likewise the value of b would never change
    - the word above that makes developers cringe is “copy”

Immutable Values (III)

- Do I really want to make an entire copy of a car each time I change something about it?
  - Maybe, but imagine an immutable linked list that is 10,000 elements long
  - If I add a new element to the front of the list, do I really want to copy all 10,000 elements to a new list?
  - List a = < code to create large list >
  - List b = a.insert(0, “CU Boulder”);
- If you don’t do this efficiently, after this simple operation, you’ll have two lists in memory with 20,001 elements and 20,000 of those elements will be duplicated
  - Not good

Immutable Values (IV)

- To address this problem of inefficiency, i.e.,
    - duplication of elements as lots of copies are made of an immutable value
- computer scientists developed the idea of persistent data structures
    - a fancy way of indicating that the data structure is used to represent lots of immutable values but it shares as much structure between the “copies” as possible
- If our linked list was implemented as a persistent data structure, then List a and List b would both be treated as separate lists that are immutable BUT in memory we would have allocated only 10,001 items
    - b would point to the head of the list; a would point to the second element of the list


B
CU
Boulder
UC
Irvine
...
A

Persistent Data Structures

- The book discusses how persistent tries can be used to represent numerous immutable values of various types of other data structures including trees, maps (hashtables), and lists
  - Such data structures can hold one million elements in a tree structure that is only four levels deep; any particular element can be accessed very quickly
- This is important because it means that you can efficiently use a persistent trie to model common data structures but make use of the pure immutability approach to design
  - More importantly model multiple immutable values of a common data structure over time with maximal sharing of common structure
    - This ensures that your immutable system is as efficient as possible

More on Pure Immutability

- Note: It's not just that you have immutable values but also your variable assignments are immutable
  - This means, you can't do this
    - String foo = "cat";
    - foo = "dog";
- Even though foo points at two different immutable objects, this would violate immutability
- In a pure immutable design, foo is not allowed to shift from "cat" to "dog" after it has been assigned
  - Instead, you must use function composition to ensure that new immutable values are constructed from previous immutable values and that the result is only stored one (or not at all)

Example

- For our example of totaling years of experience,
    - we ask the people in the room to form a chain
    - the first person in the chain hands their years of experience to the next person in the chain
    - all others take the years of experience given to them, adds that number to their own, and passes the total to the next person in the chain
    - int yearsOfExperience = firstPerson.getExperience();
    - where public void getExperience() consists of
        - return numYearsOfExperience + self.nextPerson().getExperience();

Selecting an Approach

- Most problems associated with concurrency go away if you design for isolated mutability or pure immutability
  - This aspects makes them more desirable approaches over shared mutability
- Shared mutability is the most difficult design approach to adopt given the high number of problems associated with it and the complexity of synchronization methods
  - It is however the easiest to encounter since all you need to do is write a single threaded program and then add multiple threads to it (!)
- We have seen some examples of isolated mutability so far (with more on the way) and we will see in future chapters examples of designing for pure immutability

Summary

- We have reviewed the three design approaches available for designing concurrent software systems
  - Shared Mutable Design
    - Easiest to create, hardest to debug
  - Isolated Mutable Design
    - Easiest “safe” approach to understand
    - java.util.concurrent has classes that encourage this style
  - Purely Immutable Design
    - The hardest to understand and implement but the safest style of all
    - requires a new style of programming based on the use of immutable values, persistent data structures, recursive structures and functional composition

Coming Up Next

- Lecture 9: Behavior-Driven Development and Cucumber
- Lecture 10: Agile Project Inception