

Concurrent Programming

Actors, SALSA, Coordination Abstractions

Carlos Varela
Rensselaer Polytechnic Institute

March 31, 2016

Advantages of concurrent programs

- **Reactive programming**
 - User can interact with applications while tasks are running, e.g., stopping the transfer of a big file in a web browser.
- **Availability of services**
 - Long-running tasks need not delay short-running ones, e.g., a web server can serve an entry page while at the same time processing a complex query.
- **Parallelism**
 - Complex programs can make better use of multiple resources in new multi-core processor architectures, SMPs, LANs or WANs, e.g., scientific/engineering applications, simulations, games, etc.
- **Controllability**
 - Tasks requiring certain preconditions can suspend and wait until the preconditions hold, then resume execution transparently.

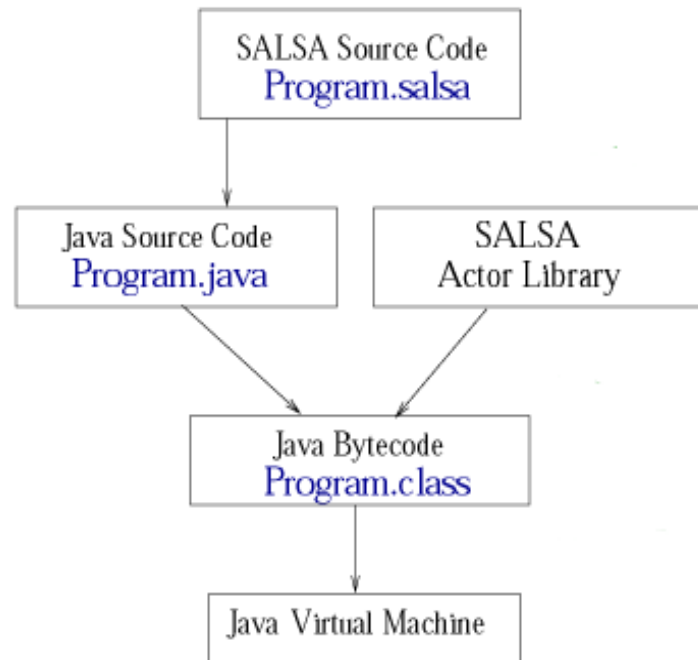
Disadvantages of concurrent programs

- **Safety**
 - « *Nothing bad ever happens* »
 - Concurrent tasks should not corrupt consistent state of program
- **Liveness**
 - « *Anything ever happens at all* »
 - Tasks should not suspend and indefinitely wait for each other (deadlock).
- **Non-determinism**
 - Mastering exponential number of interleavings due to different schedules.
- **Resource consumption**
 - Threads can be expensive. Overhead of scheduling, context-switching, and synchronization.
 - Concurrent programs can run *slower* than their sequential counterparts even with multiple CPUs!

Overview of concurrent programming

- There are four basic approaches:
 - **Sequential programming** (no concurrency)
 - **Declarative concurrency** (streams in a functional language)
 - **Message passing** with active objects (Erlang, SALSA)
 - **Atomic actions** on shared state (Java)
- The atomic action approach is the *most difficult*, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
 - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, else message passing if you can get away with it.

SALSA and Java



- SALSA source files are compiled into Java source files before being compiled into Java byte code.
- SALSA programs may take full advantage of the Java API.

Hello World Example

```
module examples.helloworld;

behavior HelloWorld {

    void act( String[] args ) {

        standardOutput <- print( "Hello" ) @
        standardOutput <- println( "World!" );

    }

}
```

Hello World Example

- The `act (String[] args)` message handler is similar to the `main (...)` method in Java and is used to bootstrap SALSA programs.
- When a SALSA program is executed, an actor of the given behavior is created and an `act (args)` message is sent to this actor with any given command-line arguments.
- References to `standardOutput`, `standardInput` and `standardError` actors are available to all SALSA actors.

SALSA Support for Actors

- Programmers define *behaviors* for actors.
- Messages are sent asynchronously.
- State is modeled as encapsulated objects/primitive types.
- Messages are modeled as potential method invocations.
- Continuation primitives are used for coordination.

Reference Cell Example

```
module examples.cell;

behavior Cell {
    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() { return content; }

    void set(Object newContent) {
        content = newContent;
    }
}
```

Actor Creation

- To create an actor:

```
TravelAgent a = new TravelAgent();
```

Message Sending

- To create an actor:

```
TravelAgent a = new TravelAgent();
```

- To send a message:

```
a <- book( flight );
```

Causal order

- In a sequential program all execution states are totally ordered
- In a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program as a whole is partially ordered

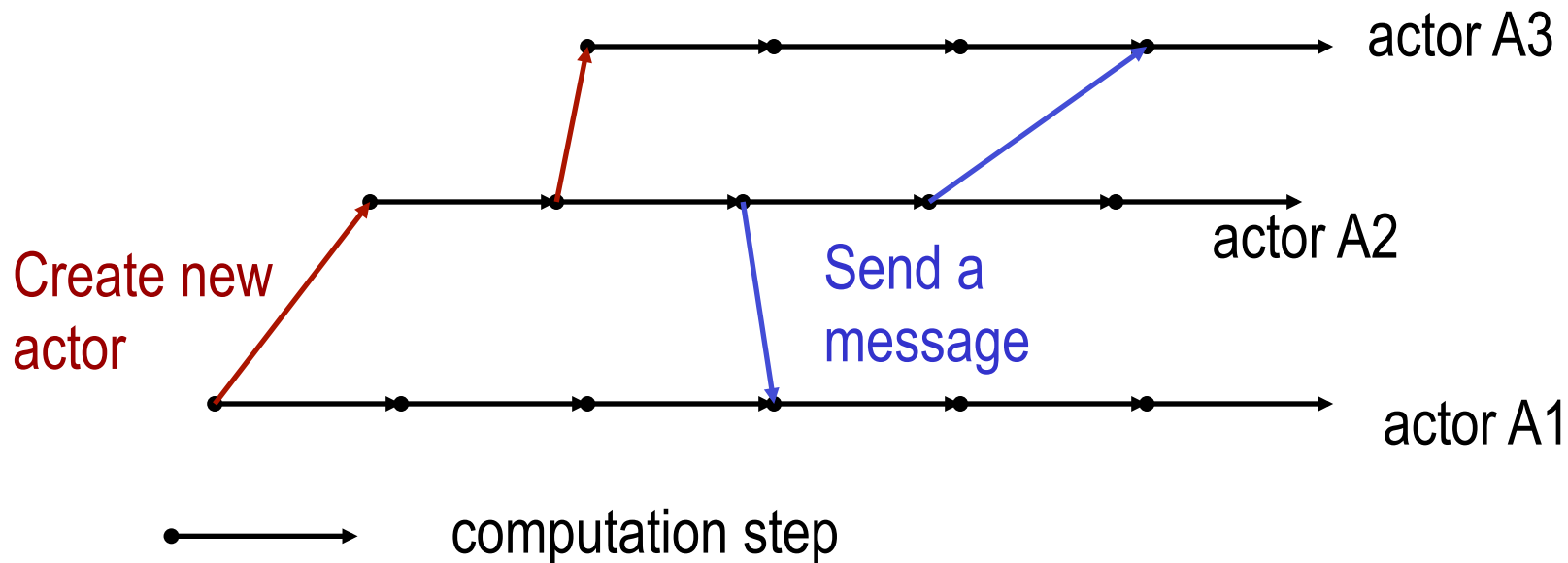
Total order

- In a sequential program all execution states are totally ordered



Causal order in the actor model

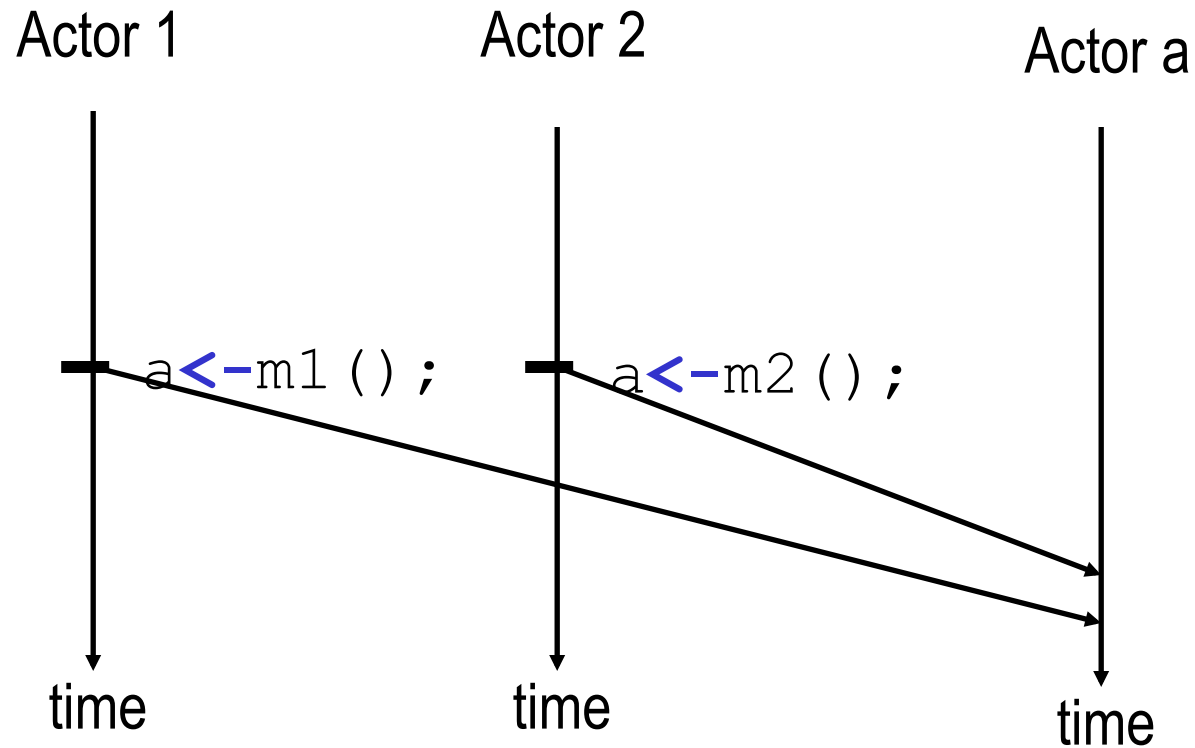
- In a concurrent program all execution states of a given actor are totally ordered
- The execution state of the concurrent program is partially ordered



Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is asynchronous message passing
 - Messages can arrive or be processed in an order different from the sending order.

Example of nondeterminism



Actor a can receive messages `m1 ()` and `m2 ()` in any order.

Coordination Primitives

- SALSA provides three main coordination constructs:
 - **Token-passing continuations**
 - To synchronize concurrent activities
 - To notify completion of message processing
 - Named tokens enable arbitrary synchronization (data-flow)
 - **Join blocks**
 - Used for barrier synchronization for multiple concurrent activities
 - To obtain results from otherwise independent concurrent processes
 - **First-class continuations**
 - To delegate producing a result to a third-party actor

Token Passing Continuations

- Ensures that each message in the continuation expression is sent after the previous message has been **processed**. It also enables the use of a message handler return value as an argument for a later message (through the token keyword).

– Example:

```
a1 <- m1 () @  
a2 <- m2 ( token );
```

Send m1 to a1 asking a1 to forward the result of processing m1 to a2 (as the argument of message m2).

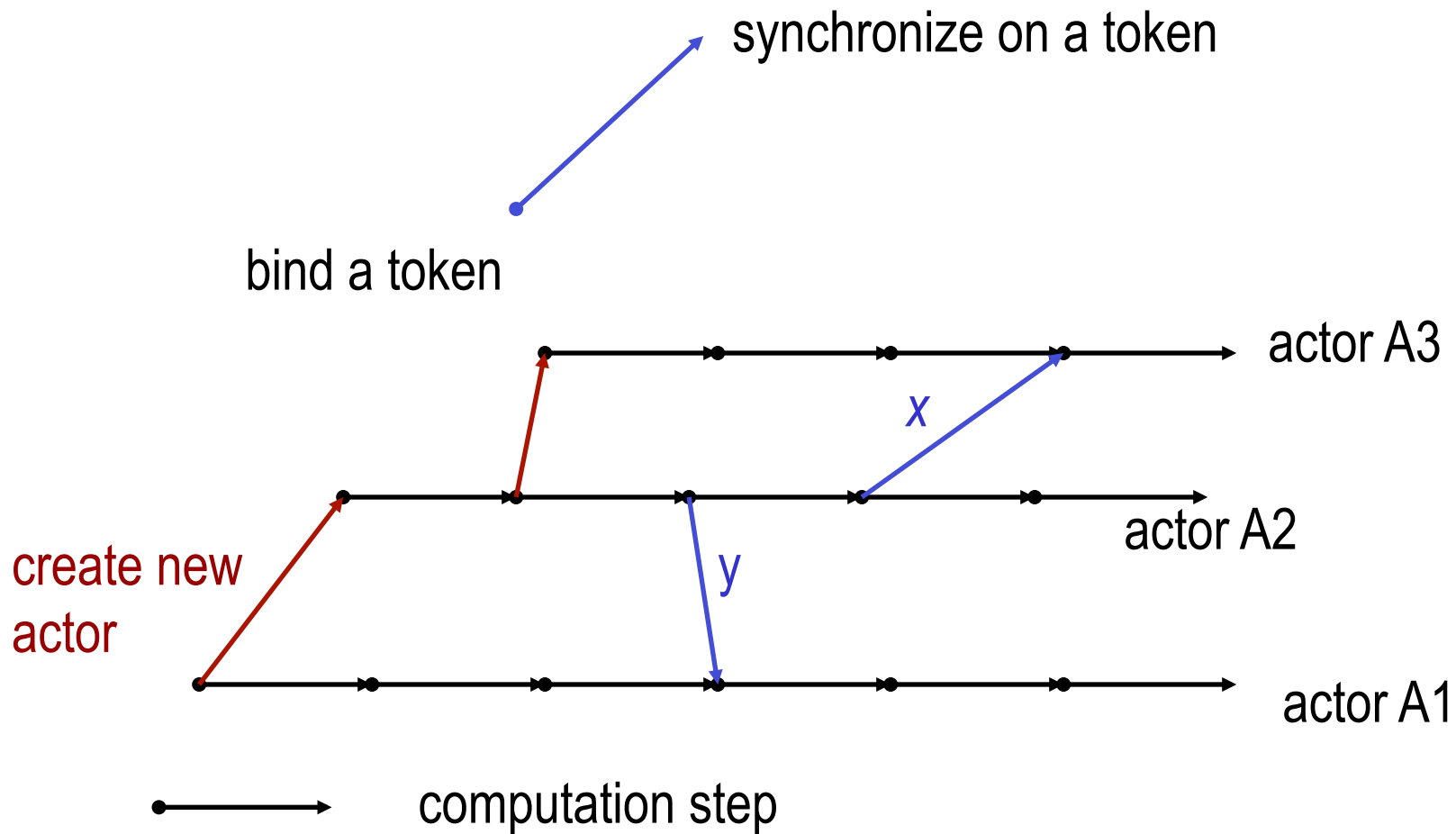
Named Tokens

- Tokens can be named to enable more loosely-coupled synchronization
 - Example:

```
token t1 = a1 <- m1 ();  
token t2 = a2 <- m2 ();  
token t3 = a3 <- m3 ( t1 );  
token t4 = a4 <- m4 ( t2 );  
a <- m ( t1 , t2 , t3 , t4 );
```

Sending $m(\dots)$ to a will be delayed until messages $m1()$. . . $m4()$ have been processed. $m1()$ can proceed concurrently with $m2()$.

Causal order in the actor model



Cell Tester Example

```
module examples.cell;

behavior CellTester {

    void act( String[] args ) {

        Cell c = new Cell("Hello");
        standardOutput <- print( "Initial Value:" ) @
        c <- get() @
        standardOutput <- println( token ) @
        c <- set("World") @
        standardOutput <- print( "New Value:" ) @
        c <- get() @
        standardOutput <- println( token );

    }
}
```

Join Blocks

- Provide a mechanism for synchronizing the processing of a set of messages.
- Set of results is sent along as a *token* containing an array of results.
 - Example:

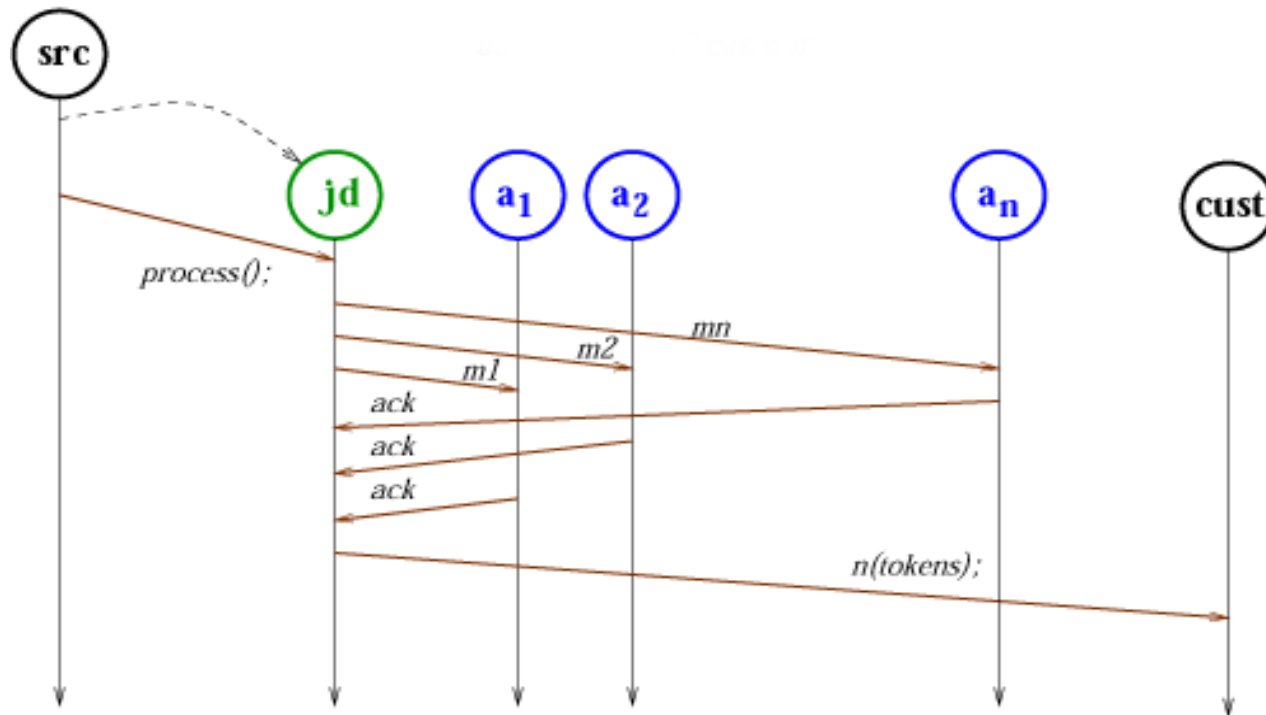
```
Actor[] actors = { searcher0, searcher1,
                  searcher2, searcher3 };

join {
  for (int i=0; i < actors.length; i++){
    actors[i] <- find( phrase );
  }
} @ resultActor <- output( token );
```

Send the find(phrase) message to each actor in actors[] then after all have completed send the result to resultActor as the argument of an output(...) message.

Example: Acknowledged Multicast

```
join{ a1 <- m1 (); a2 <- m2 (); ... an <- mn (); } @  
cust <- n(token);
```



Lines of Code Comparison

	Java	Foundry	SALSA
Acknowledged Multicast	168	100	31

First Class Continuations

- Enable actors to delegate computation to a third party independently of the processing context.
- For example:

```
int m (...) {  
    b <- n (...) @ currentContinuation;  
}
```

Ask (delegate) actor b to respond to this message m on behalf of current actor ($self$) by processing its own message n .

Delegate Example

```
module examples.fibonacci;

behavior Calculator {

  int fib(int n) {
    Fibonacci f = new Fibonacci(n);
    f <- compute() @ currentContinuation;
  }
  int add(int n1, int n2) {return n1+n2;}

  void act(String args[]) {
    fib(15) @ standardOutput <- println(token);
    fib(5) @ add(token,3) @
    standardOutput <- println(token);
  }
}
```

Fibonacci Example

```
module examples.fibonacci;

behavior Fibonacci {
  int n;

  Fibonacci(int n)          { this.n = n; }

  int add(int x, int y) { return x + y; }

  int compute() {
    if (n == 0)          return 0;
    else if (n <= 2)    return 1;
    else {
      Fibonacci fib1 = new Fibonacci(n-1);
      Fibonacci fib2 = new Fibonacci(n-2);
      token x = fib1<-compute();
      token y = fib2<-compute();
      add(x,y) @ currentContinuation;
    }
  }

  void act(String args[]) {
    n = Integer.parseInt(args[0]);
    compute() @ standardOutput<-println(token);
  }
}
```

Fibonacci Example 2

```
module examples.fibonacci2;

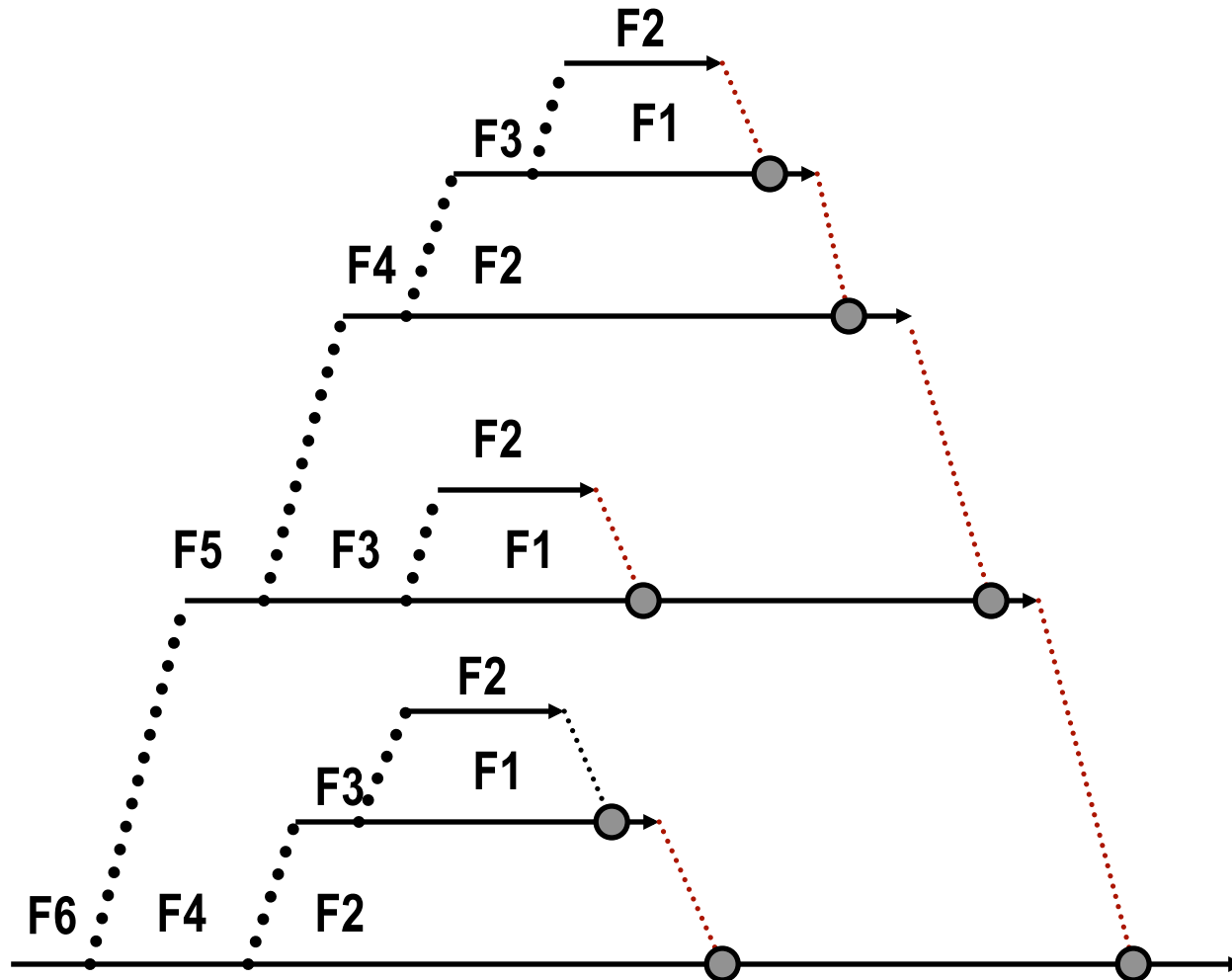
behavior Fibonacci {

  int add(int x, int y) { return x + y; }

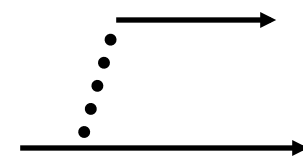
  int compute(int n) {
    if (n == 0)      return 0;
    else if (n <= 2) return 1;
    else {
      Fibonacci fib = new Fibonacci();
      token x = fib <- compute(n-1);
      compute(n-2) @ add(x,token) @ currentContinuation;
    }
  }

  void act(String args[]) {
    int n = Integer.parseInt(args[0]);
    compute(n) @ standardOutput<-println(token);
  }
}
```

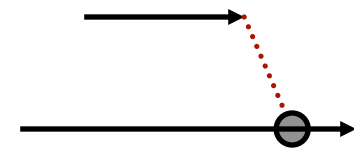
Execution of salsa Fibonacci 6



Create new actor



Synchronize on result



Non-blocked actor



Exercises

1. How would you implement the join continuation linguistic abstraction in terms of message passing?
2. Download and execute the `CellTester.salsa` example.
3. Write a solution to the Flavius Josephus problem in SALSA. A description of the problem is at Concepts Techniques and Models of Computer Programming textbook Section 7.8.3 (page 558).