# CS3110 Spring 2017 Lecture 21: Distributed Computing with Functional Processes

Robert Constable

| | Date for | Due Date |
|---|---|---|
| PS6 | Out on April 24 | May 8 (day of last lecture) |

## 1 Introduction

In the next two lectures, we will look at computing with *asynchronous functional processes* in OCaml. PS6 that just came out will be on this topic. The Async package is a feature of OCaml that Jane Street advocated for. Yaron Minsky from that company often comes to visit. He is also a co-author on the text book we often cite.

At Cornell the systems group, the formal methods, and PL groups have worked closely with each other for over twenty years. The material we will cover is based on results of this shared interest. Many of our joint results use functional programming concepts because proof assistants are especially well suited to reasoning about *asynchronous functional processes*. We came to see from this work why Brouwer's notion of free choice sequences makes sense in modern computer science. In Lecture 20 we saw free choice sequences in action. We will see them again here.

The area of *distributed computing* is extremely broad and deep. It has taken many years to isolate the basic concepts, such as *causal order*, *distributed state machines*, *virtual synchrony*, logical clocks, fault tolerance, Byzantine fault tolerance, etc. We will focus on the problem of achieving *consensus decisions* among a group of processes computing together asynchronously. This topic is relevant to modern cloud computing and fundamental "systems theory."

In particular, we will use a simple but realistic consensus protocol, called 2/3 *consensus* to illustrate the concepts. This protocol is the heart of PS6.

We will briefly discuss a logic for reasoning about protocols called *event logic* (or the *logic of events*). This language arose as the formal methods researchers tried to make logically precise some of the new ideas being explored by the systems group. In this realm, the notion of co-inductive (also called co-recursive) types is quite natural and fundamental. Event logic is also becoming important in cyber physical systems (CPS).

## 2 Leader election in a ring

The leader election algorithm is quite easy to understand, specify precisely, and verify. Each process sends its unique identifier (uid) to its clockwise neighbor. Each process compares the uid received with its own uid. If the received uid is larger, then the process passes it along. Otherwise it "drops the uid," that is, it does not send it along. The one and only process that receives its own uid back knows because of the ring structure of the communication links that its uid is the largest, otherwise it would have been dropped by a process with a higher uid. When a processes receives back its own uid, it know that it must have been the largest uid, and the process declares itself the leader and sends this information around the ring. (Of course every process can know this on its own by keeping the list of uids it sees before being informed). An easy correctness proof is sketched below in these notes.

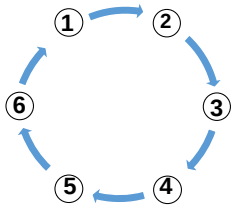
# Specification for Leader Election in a Ring

Leader Election

In a Ring R of Processes with Unique Identifiers (uid's)

Specification

Let R be a non-empty list of locations linked in a ring




Let $n(i) = dst(out(i))$, the next location

Let $p(i) = n^{-1}(i)$, the predecessor location

Let $d(i,j) = \mu k \geq 1.\ n^k(i) = j$, the distance from i to j

Note $i \neq p(j) \Rightarrow d(i,p(j)) = d(i,j)-1$.

3

# Specification, continued

$$\text{Leader}(R,es) == \exists\, ldr: R.\ (\exists e@ldr.\ kind(e)=leader)\ \&\ (\forall i:R.\ \forall e@i.\ kind(e)=leader \Rightarrow i=ldr)$$

Theorem $\forall$ R:List(Loc). Ring(R)

$\exists$ D:Dsys(R). Feasible(D) &

$\forall$es: ES. Consistent(D,es). Leader(R,es)

4

# Realizing Leader Election

Theorem ∀R:List(Loc) . Ring(R)
∃D:Dsys(R) . Feasible(D) .
∀es:Consistent(D, es) . (LE(R, es) ⇒ Leader(R, es))

Proof: Let $m = max\{uid(i) \mid i \in R\}$, then $ldr = uid^{-1}(m)$.
We prove that $ldr = uid^{-1}(m)$ using three simple lemmas.

5

# Lemmas

Lemma 1. ∀i : R. ∃e @ i. kind(e) = rcv (in(i) , <vote, ldr>)
By induction on distance of i to ldr.

Lemma 2. ∀i, j : R. ∀e @ i. kind(e) = rcv (in(i) , <vote, j>) .
(j = ldr ∨ d(ldr, j) < d(ldr, i))
By induction on causal order of rcv events.

Lemma 3. ∀i : R. ∀e' @ i. (kind(e') = leader ⇒ i = ldr)

If kind(e') = leader, then by property 5, ∃v @ i.rcv (in(i) , <vote, uid(i)>) .
Hence, by Lemma 2 i = ldr ∨ (d(ldr, i) < d(ldr, i))
but the right disjunct is impossible.

Finally, from property 4, it is enough to know
∃e.kind(e) = rcv (in(ldr) , <vote, uid(ldr)>)
which follows from Lemma 1.

QED

6

# Voting Approach to Consensus

Suppose a group G of n processes, Pi, decide by voting. If each Pi collects all n votes into a list L, and applies some deterministic function f(L), such as majority value or maximum value, etc., then consensus is trivial in one step, and the value is known at each process in the first round – possibly at very different times.

The problem is much harder because of possible failures.

# Fault Tolerance

Replication is used to ensure system availability in the presence of faults. Suppose that we assume that up to f processes in a group G of n might fail, then how do the processes reach consensus?

The TwoThirds method of consensus is to take n = 3f +1 and collect only 2f+1 votes on each round, assuming that f processes might have failed.

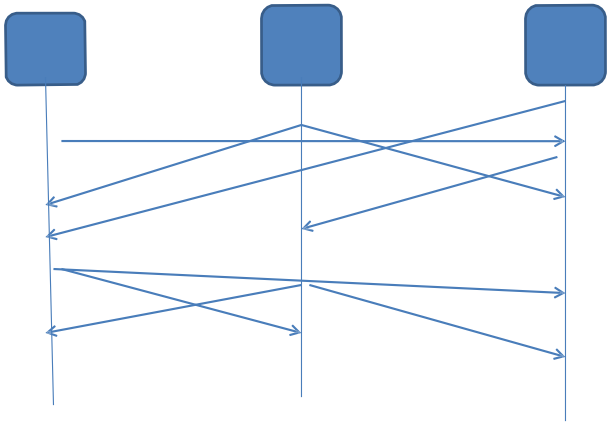
## 3 Protocols and Events

Here is our plan. We will illustrate the ideas using *consensus protocols*. These are critical to the *asynchronous distributed computing* used in "the cloud," especially at Google, Microsoft, Amazon, the FAA, and the French ATC system. Research in this area has led to very deep foundational results such as FLP, distributed state machines, virtual synchrony, Byzantine fault tolerance, and more.

The topic has also led to new insights about *computation beyond Turing-computability*. The key idea is that the communication infrastructure adds an element of *unpredictable choice* to computation. This notion was considered by mathematicians, especially Brouwer, before it became central in CS. So there is this certain *conceptually fundamental character* to these ideas that goes beyond technology, just as the idea of computability goes beyond any particular formalism for it – Turing machines, the $\lambda$-calculus, general recursive functions, etc.

### Requirements of Consensus Task

Use asynchronous message passing to decide on a value.



# Logical Properties of Consensus

P1: If all inputs are unanimous with value v, then any decision must have value v.

All v:T. ( If All e:E(Input). Input(e) = v  then
All e:E(Decide). Decide(e) = v)

Input and Decide are event classes that effectively partition the events and assign values to them. The events are points in abstract space/time at which "information flows." More about this just below.

# Logical Properties continued

P2: All decided values are input values.

All e:E(Decide). Exists e':E(Input).
e' < e & Decide(e) = Input(e')

We can see that P2 will imply P1, so we take P2 as part of the requirements.

## Further Requirements for Consensus

The key safety property of consensus is that all decisions agree.

P3: Any two decisions have the same value.

This is called agreement.

All e1,e2: E(Decide). Decide(e1) = Decide(e2).

## Liveness

If f processes eventually fail, then our design will work because if f have all failed by round r, then at round r+1, all alive processes will see the same 2f+1 values in the list L, and thus they will all vote for v' = f(L), so in round r+2 the values will be unanimous which will trigger a decide event.

# Example for f = 1, n = 4

Here is a sample of voting in the case T = {0,1}.

| 0 | 0 | 1 | 1 | inputs |
|---|---|---|---|---|
| 0 01_ | 001_ | 001_ | _011 | collected votes |
| 0 | 0 | 0 | 1 | next vote |

---

| 000_ | 00_1 | 0_01 | _001 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |

where f is majority voting, first vote is input, round numbers omitted.

# Pseudo-code for 2/3 Consensus

**Begin** r:Nat, decided_i, vote_i: Bool,
r = 0, decided_i = false, vi = input to Pi; vote_i = vi

**Until** decided_i **do**:

1. r := r+1
2. **Broadcast** vote <r,vote_i> to group G
3. **Collect** 2f+1 round r votes in list L
4. vote_i := majority(L)
5. **If** unanimous(L) **then** decided_i := true

**End**

# Safety Example

We can see in the f = 1 example that once a process Pi receives 2/3 unanimous values, say 0, it is not possible for another process to over turn the majority decision.

Indeed this is a general property of a 2/3 majority, the remaining 1/3 cannot overturn it even if they band together on every vote.

# Safety Continued

- In the general case when voting is not by majority but using f(L) and the type of values is discrete, we know that if any process Pi sees unanimous value v in L, then any other process Pj seeing a unanimous value v' will
- see the same value, i.e. v = v' because the two lists, Li and Lj at round r must share a value, that is they intersect.

# Executing Systems of Processes

The environment chooses which messages will be delivered. A run of a system is an unbounded sequence of pairs <sys,choice>.

From a run of a system, we can build event structures with locations and causal order.

# Event Orderings over Runs

An event ordering of a run R is a collection of events E, a function loc giving the location of the event, a well founded causal order < on events, and info, the information conveyed by an event: <E, loc, <, info>

The events are pairs <x,n> at which location x receives a message at step n of the run.

# Constructive FLP

•Theorem (Fischer/Lynch/Paterson, 1985): Given any deterministic non-blocking consensus procedure P with two or more processes tolerating a single failure, P allows non-terminating executions.

# 4 Message sequence diagrams and the logic of events

Protocol designers use Message Sequence Diagrams to think about their task. They are used to illustrate both the specification of the task and its solution as a distributed protocol. We will examine these diagrams in the next lecture.