Distributed Computing

José Orlando Pereira

http://di.uminho.pt/~jop

2013/2014
Asynchronous system model

- Avoid using a global time reference
- Assume no bounds on:
  - clock drift
  - processing time
  - message passing time
- Why?
Example: Leader election

- Select a unique leader in a distributed system
- Useful for:
  - Coordination
  - Efficiency
  - ...

Network
Example: Properties

- No two processes disagree about the leader
- Every process will select a leader
Example: Simple algorithm (FloodMax)

- Each process trying to be the leader sends its network address to all others.
- Each process considers the process with the highest address to be the leader.
Example: Approach

- Start with a synchronous reliable fully connected network
- Relax the system model:
  - Unbounded message loss
  - Large/unknown graph diameter
  - Dynamic graph
- Example: Leader election
Example: Leader election

- **Trivial**
  - Static known participants

- **Possible**
  - Synchronous Reliable clique
  - Synchronous Bounded unreliable clique

- **Impossible (eventually)**
  - Synchronous Reliable dynamic

- **Impossible**
  - Disconnected

- **Asynchronous Reliable clique**
  - Can loose all messages
  - Possible
  - Possible (eventually)

- Other configurations:
  - Synchronous Reliable static
  - Synchronous Reliable connected
  - Unknown diameter
  - Bounded unreliable clique

© 2007-2013 José Orlando Pereira
HASLab / U.Minho & INESC TEC
Summary

- Asynchrony subsumes:
  - Heterogeneity
  - Dynamics
  - Uncertainty

- Much simpler than handling them explicitly

- Often considered an Universal model:
  - Widely applicable solutions
Goals

- How do we make sure that algorithms are correct?
- Why are algorithms correct?
Sample computation

- An alarm clock program:

```plaintext
main:
    cnt:=3
while cnt>0:
    sleep 1s
    cnt := cnt-1
    ring
```
Observation

- Select model variables and periodically observe the system:
Abstraction

- Choose observation that allows reasoning on the desired properties:
Consider all possible sequences of chosen atomic actions:
Safety properties

- Nothing bad ever happens:

OK! \( \text{vcnt:=3} \)  \( \text{vcnt:=3} \)

OK! \( \text{vcnt:=2} \)

OK! \( \text{vcnt:=1} \)

OK! \( \text{vcnt:=0 END} \)

\( \text{vcnt:=3} \)

\( \text{vcnt:=4} \)

\( \text{vcnt:=2} \)

\( \text{vcnt:=1} \)

\( \text{vcnt:=0 END} \)

\( \text{vcnt:=3} \)

\( \text{vcnt:=2} \)

\( \text{vcnt:=1} \)

\( \text{vcnt:=1} \)

\( \text{vcnt:=1} \) ...

...
Liveness properties

- Something good eventually\(^(*)\) happens:

\[(*)\text{ eventually} = \text{inevitavelmente} \neq \text{eventuallye}\]
Specification

- **Specification is a set of allowable behaviors:**

\[ S = \{ \text{vcnt:=3, timeout}, \text{vcnt:=2, timeout}, \text{vcnt:=1, timeout}, \text{vcnt:=0, ring} \} \]
Goal 1: Is it correct?

- Is there a convenient representation for specification sets?
  - Compact
  - Practical

- How to prove safety and liveness properties?
Specifications and automata

- Specification is a set of allowable behaviors:
  \[ S = \{ \text{vcnt:=3} \text{timeout} \rightarrow \text{vcnt:=2 \ timeout} \rightarrow \text{vcnt:=1 \ timeout} \rightarrow \text{vcnt:=0 \ END} \} \]

- An automaton provides a compact and practical representation
An I/O automaton $A$ has five components:

- $\text{sig}(A)$, a triplet $S$ of disjoint sets of actions:
  - $\text{in}(S)$, the input actions
  - $\text{out}(S)$, the output actions
  - $\text{int}(S)$, the internal actions
- $\text{states}(A)$, a (possibly infinite) set of states
- $\text{start}(A)$, a non-empty subset of $\text{states}(A)$
- $\text{trans}(A)$, a subset of $\text{states}(A) \times \text{acts}(\text{sig}(A)) \times \text{states}(A)$
- $\text{tasks}(A)$, a partition of $\text{local}(\text{sig}(A))$
Transitions

- A action is **enabled** in state $s$ if there is some $\pi, s'$ such that $(s, \pi, s') \in \text{trans}(A)$
- Input transitions are required to be enabled in all reachable states of $A$
- A state in which only input transitions are enabled is said to be quiescent
Signature and State

- **Input:**
  - none

- **Internal:**
  - Timeout

- **Output:**
  - Ring

- **States:**
  - `vcnt`, integer, initially 3
  - `END`, boolean, initially false
Transitions

- **Timeout:**
  - Pre-condition:
    - ¬END and vcnt > 0
  - Effect:
    - vcnt := vcnt - 1

- **Ring:**
  - Pre-condition:
    - ¬END and vcnt = 0
  - Effect:
    - END := True

This is an equation, not an attribution!
Effects

• Effect equation:
  - \( vcnt := vcnt - 1 \)

• Read this as:
  - “vcnt-after = vcnt-before – 1 and the state otherwise unchanged”

• Could be written as:
  - \( vcnt-after + 1 = vcnt-before \)
  - \( vcnt-before - vcnt-after = 1 \)
  - ...
Safe behaviors

**Enumerating safe behaviors:**

- Start with a behavior for each state $s$ in $\text{start}(A)$
- For each transition $(s,a,s')$ in $\text{trans}(A)$ enabled for some state $s$ at the end of any known safe behavior:
  - Create a behavior with $(a,s')$ appended
- Repeat (possibly, for ever...)
Safety properties

- Proof of safety properties:
  - Invariant proof by induction

- Strategies:
  - Strengthen the invariant
  - Include trace in state
Invariants

- Goal: Prove that always $\text{vcnt} < 4$ (safety!).
- Proof by induction:
  - Base step: True for all initial states?
    - $3 < 4$: Yes!
  - Induction step: True for any next step?
    - Timeout transition:
      - $\text{vcnt-after} = \text{vcnt-before} - 1$
      - $\text{vcnt-before} < 4$
      - $\text{vcnt-after} + 1 < 4$
      - $\text{vcnt-after} < 3 < 4$: Done
    - Ring transition:
      - always $\text{vcnt-after} = \text{vcnt-before} = 0$
      - $0 < 4$: Done
Example: Reliable channel

- Reliable channel:
  - Unordered
  - FIFO

Why \( \text{Receive}(m) \) and not \( m := \text{Receive}() \)?
Example: Reliable channel

- **State:**
  - transit, bag of M, initially \{\}

- **Send(m), m ∈ M:**
  - **Pre-condition:** True
  - **Effect:** transit := transit + \{m\}

- **Receive(m), m ∈ M:**
  - **Pre-condition:** m in transit
  - **Effect:** transit := transit - \{m\}
Behaviors of a channel

- Concurrency is modeled by alternative enabled transitions:
  - Sender and receiver
  - Within the channel (reordering)
Liveness and fairness

- Some behaviors do not satisfy liveness:
  - If m is sent, eventually m is received
- Some transitions don't get a fair chance to run:
  - receive(m1) and receive(m*)
Partition transitions in tasks:

- Tasks:
  - For all m: \{receive(m)\}

Assume that no task can be forever prevented from taking a step.

What about a FIFO reliable channel?
Liveness and fairness

- FIFO order excludes a number of behaviors
  - Only executions with a finite number of receive(m) steps are unfair
- Fairness ensured by a single task:
  - \{For all m: receive(m)\}
Example: FIFO channel

- **State:**
  - transit, seq. of M, initially <>

- **Send(m), m ∈ M:**
  - Pre-condition:
    - True
  - Effect:
    - transit := transit + <m>

- **Receive(m), m ∈ M:**
  - Pre-condition:
    - m = head(transit)
  - Effect:
    - transit := tail(transit)
  - Tasks:
    - {For all m: receive(m)}
Example: Token ring

- Rotating token algorithm:
  - Mutual exclusion?
  - Deadlock freedom?
Example: Token ring

- **State:**
  - n is the number of nodes
  - token[0] = 1
  - token[i] = 0, for 0 < i < n

- **Move(i):**
  - **Pre-condition:**
    - token[i] = 1
  - **Effect:**
    - token[i] := 0
    - token[(i+1) mod n] := 1
Example: Token ring

- Mutual exclusion:
  - There is at most one token in the ring (i.e. sum of token[i] ≤ 1)

- Proof by induction:
  - Base step:
    - $\sum_{i} \text{token}[i] = 1$ trivially true
  - Induction step:
    - $\sum_{i} \text{token}-before[i] ≤ 1 \Rightarrow \sum_{i} \text{token}-after[i] ≤ 1
Example: Token ring

- No starvation:
  - Eventually i gets the token at least k times

- Proof with a progress function:
  - Function from state to a well-founded set
  - Helper actions decrease the value
  - Other actions do not increase the value
  - Helper actions are taken until goal is met (i.e. enabled and in separate tasks)

Invariant assertion
Define progress function $f$ as:

- Target is non-negative integers
- Value is $((k-1) \times n + i - 1) - \text{length(trace)}$

Example with $n=3$, $k=2$, and $i=3$:
Conclusion

- First goal achieved:
  - I/O Automata
  - Safety and liveness proofs
- More:
  - Composition
  - Refinement
Goal 2: Why is it correct?

- To what extent does local state reflect global state?
Example: Distributed deadlock

- Remote invocation
- All processes request and reply to invocations
- A mutex is held while invoking remotely or handling remote invocations
- Distributed deadlock possible when multiple processes invoke each other
Example: Distributed deadlock

- Deadlock-free run:
Example: Distributed deadlock

- Distributed deadlock:
  
  blocked waiting for process 3...
  
  blocked waiting for process 1...
  
  blocked waiting for process 2...
Example: Distributed deadlock

- Instant observation is impossible:

  - blocked waiting for process 3...
  - blocked waiting for process 1...
  - blocked waiting for process 2...
Example: Distributed deadlock

Deadlock detection with a “wait for” graph:
Example: Distributed deadlock

- A more complex deadlock-free run:
Example: Distributed deadlock

- A deadlock-free WFG:
Example: Distributed deadlock

- A WFG with a ghost deadlock:
Global Property Evaluation

- All these problems are instances of the Global Property Evaluation (GPE) problem
- Can it be solved in an asynchronous system?
- Methods that can be used? Relative cost?
Passive monitor process

- Report all events to monitor:
First try: Synchronous system

- Global clock, $\delta$ upper bound on message delay
- Tag events with real time
- Consider events only up to $t-\delta$
  - With synchronous rounds, this means using messages from the previous round!
First try: Synchronous system
Clock properties

- What properties of a real-time clock make this approach correct?
- RC(i) the time at which i i happened
Definition: Causality

- Events i and j are **causally related** \((i \rightarrow j)\) iff:
  - i precedes j in some process \(p\)
  - for some \(m\), \(i = \text{send}(m)\) and \(j = \text{receive}(m)\)
  - for some \(k\), \(i \rightarrow k\) and \(k \rightarrow j\) (transitivity)

- Events i and j are concurrent \((i || j)\) iff neither \(i \rightarrow j\) or \(j \rightarrow i\)
Causality

causally precedes

concurrent
Clock properties

- If $i \rightarrow j$ then $RC(i) < RC(j)$
- For some event $j$:
  - When we are sure that there is no unknown $i$ such that $RC(i) < RC(j)$
  - Then there is no $i$ such that $i \rightarrow j$
- Can we build a logical clock with the same property?
Second try: Logical clock

- Tag events as follows:
  - Local events: increment counter
  - Send events: increment and then tag with counter
  - Receive events: update local counter to maximum and then increment

- Use FIFO channels

- Consider events only up to the minimum of maximum tags
Second try: Logical clock
Scalar clocks

- Synchronous system (RC):
  - Delay $\delta$ to consistency

- Asynchronous system (LC):
  - Possible unbounded delay to consistency
  - Blocks if some process stops sending messages
Third try: Vector clock

- Tag events with a vector as follows:
  - Local event at i: increment counter i
  - Send event at i: increment counter i and tag with vector
  - Receive event at i: update each counter to maximum and increment counter i
Third try: Vector clock

- \([1,0,0]\)
- \([2,1,0]\)
- \([3,1,4]\)
- \([4,1,4]\)
- \([5,1,4]\)
- \([6,1,4]\)
- \([7,1,4]\)
- \([0,1,0]\)
- \([1,2,5]\)
- \([5,3,5]\)
- \([5,4,5]\)
- \([0,0,1]\)
- \([1,0,2]\)
- \([1,0,4]\)
- \([1,0,5]\)
- \([1,0,6]\)
- \([6,1,7]\)
- \([6,1,8]\)
- \([5,2,6]\)
Causal delivery

- The monitor delivers events as follows:
  - With local vector l[...]
  - For some r[...] from i
  - Wait until:
    - l[i]+1=r[i]
    - For all j≠i: r[i]≤l[i]
- The monitor is always in a consistent cut
- Blocking can be avoided by forwarding past messages
No reporting to monitor process

- Reporting all events to a monitor causes a large overhead
- Can a query be issued at some point in time?
Fourth try: No reporting, synchronous

- Monitor broadcasts tss in the future

- At tss, each process:
  - Records state
  - Sends messages to all others
  - Starts recording messages until receiving a message with RC > tss

- After stopping, sends all data to monitor
Fourth try: No reporting, synchronous
Fifth try: No reporting, logical clock

At 8!
Send a “Snapshot” message to some process

Upon receiving for the first time:

- Records state
- Relays “Snapshot” to all others
- Starts recording on each channel until receiving “Snapshot”

Send all data to monitor
Chandy and Lamport

Snapshot!
Global Property Evaluation

- GPE requires no gaps in observed history, regarding causality
- What properties can be evaluated?
Cuts and consistency

- A **cut** is the union of prefixes of process history
- A **consistent cut** includes all causal predecessors of all events in the cut
- Intuitive methods:
  - If a cut is an instant, there are no messages from the future
  - In the diagram, no arrows enter the cut
  - All events in the frontier are concurrent
Consistent cuts
Conclusion

- Second goal achieved:
  - Causality
  - Consistent cuts