CS 425/ECE 428
Distributed Systems

Nitin Vaidya

T.A.s

- Persia Aziz
- Frederick Douglas
- Su Du
- Yixiao Lin

- Course handout

  … textbook
  … office hours
  … Piazza
  … grading policy
  … late submission policy

Course website

… mid-term exam schedule
… lectures page
… homework

… programming assignments
(for 4 credit hours only)

What’s this course about ?

What this course is not about …

As you can see, I have memorized this utterly useless piece of information long enough to pass a test question. I now intend to forget it forever. You've taught me nothing except how to cynically manipulate the system.

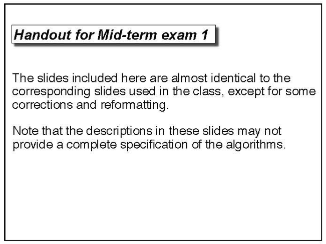
- ??????

Calvin and Hobbes

As you can see, I have memorized this utterly useless piece of information long enough to pass a test question. I now intend to forget it forever. You've taught me nothing except how to cynically manipulate the system.

\- *Calvin*

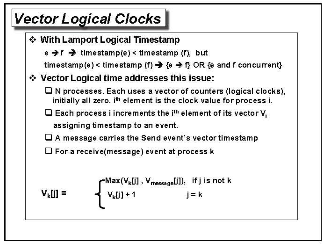
Handout provided for 1st mid-term in Spring 2014 … something similar this semester too



Handout for Mid-term exam 1

The slides included here are almost identical to the corresponding slides used in the class, except for some corrections and reformatting.

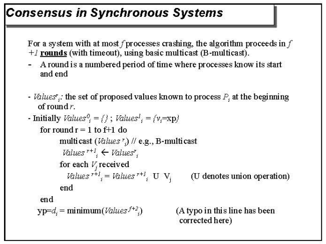
Note that the descriptions in these slides may not provide a complete specification of the algorithms.



Vector Logical Clocks

- ❖ **With Lamport Logical Timestamp**
  - e → f → timestamp(e) < timestamp (f), but
  - timestamp(e) < timestamp (f) → {e → f} OR {e and f concurrent}
- ❖ **Vector Logical time addresses this issue:**
  - ☐ N processes. Each uses a vector of counters (logical clocks), initially all zero. i^{th} element is the clock value for process i.
  - ☐ Each process i increments the i^{th} element of its vector V_i assigning timestamp to an event.
  - ☐ A message carries the Send event's vector timestamp
  - ☐ For a receive(message) event at process k

$$V_k[j] = \begin{cases} \mathrm{Max}(V_k[j],\ V_{message}[j]), & \text{if j is not k} \\ V_k[j] + 1 & j = k \end{cases}$$

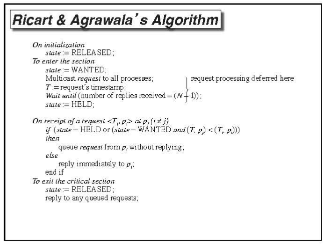


Consensus in Synchronous Systems

For a system with at most *f* processes crashing, the algorithm proceeds in *f +1* **rounds** (with timeout), using basic multicast (B-multicast).

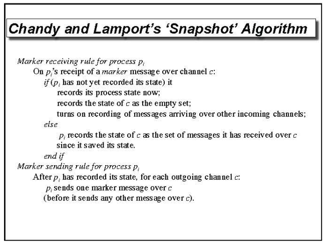
- A round is a numbered period of time where processes know its start and end

```
- $Values^r_i$: the set of proposed values known to process $P_i$ at the beginning
  of round $r$.
- Initially $Values^0_i = \{\}$ ; $Values^1_i = \{v_i = xp\}$
    for round r = 1 to f+1 do
        multicast ($Values^r_i$) // e.g., B-multicast
        $Values^{r+1}_i \leftarrow Values^r_i$
        for each $V_j$ received
            $Values^{r+1}_i = Values^{r+1}_i$ U $V_j$        (U denotes union operation)
        end
    end
  yp=$d_i$ = minimum($Values^{f+2}_i$)          (A typo in this line has been
                                                  corrected here)
```



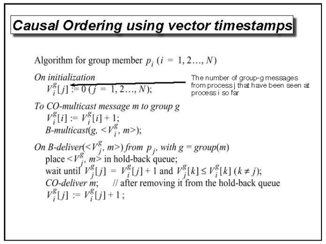
Ricart & Agrawala's Algorithm

```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast request to all processes;              } request processing deferred here
    T := request's timestamp;                        }
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request <$T_i$, $p_i$> at $p_j$ ($i \neq j$)
    if (state = HELD or (state = WANTED and $(T, p_j) < (T_i, p_i)$))
    then
        queue request from $p_i$ without replying;
    else
        reply immediately to $p_i$;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```



Chandy and Lamport's 'Snapshot' Algorithm

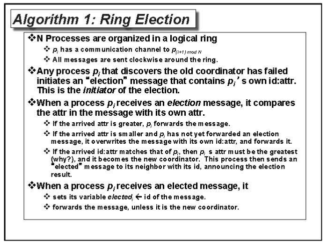
```
Marker receiving rule for process $p_i$
    On $p_i$'s receipt of a marker message over channel $c$:
        if ($p_i$ has not yet recorded its state) it
            records its process state now;
            records the state of $c$ as the empty set;
            turns on recording of messages arriving over other incoming channels;
        else
            $p_i$ records the state of $c$ as the set of messages it has received over $c$
            since it saved its state.
        end if
Marker sending rule for process $p_i$
    After $p_i$ has recorded its state, for each outgoing channel $c$:
        $p_i$ sends one marker message over $c$
        (before it sends any other message over $c$).
```



Causal Ordering using vector timestamps

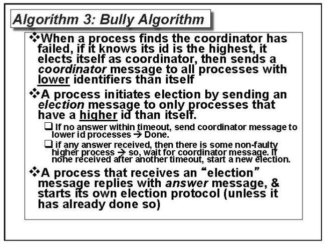
```
Algorithm for group member $p_i$ ($i = 1, 2\ldots, N$)

On initialization
    $V^g_i[j] := 0$ ($j = 1, 2\ldots, N$);

To CO-multicast message m to group g
    $V^g_i[i] := V^g_i[i] + 1$;
    B-multicast(g, <$V^g_i$, m>);

On B-deliver(<$V^g_j$, m>) from $p_j$, with g = group(m)
    place <$V^g_j$, m> in hold-back queue;
    wait until $V^g_j[j] = V^g_i[j] + 1$ and $V^g_j[k] \le V^g_i[k]$ ($k \neq j$);
    CO-deliver m;      // after removing it from the hold-back queue
    $V^g_i[j] := V^g_i[j] + 1$;
```

The number of group-g messages from process j that have been seen at process i so far



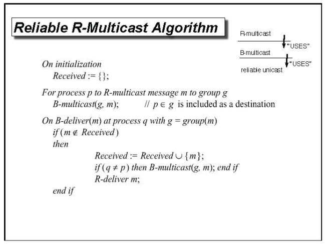
Algorithm 1: Ring Election

- ❖ **N Processes are organized in a logical ring**
  - ❖ p_i has a communication channel to $p_{(i+1)\ mod\ N}$
  - ❖ All messages are sent clockwise around the ring.
- ❖ **Any process p_i that discovers the old coordinator has failed initiates an "election" message that contains p_i' s own id:attr. This is the *initiator* of the election.**
- ❖ **When a process p_i receives an *election* message, it compares the attr in the message with its own attr.**
  - ❖ If the arrived attr is greater, p_i forwards the message.
  - ❖ If the arrived attr is smaller and p_i has not yet forwarded an election message, it overwrites the message with its own id:attr, and forwards it.
  - ❖ If the arrived id:attr matches that of p_i, then p_i s attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an "elected" message to its neighbor with its id, announcing the election result.
- ❖ **When a process p_i receives an elected message, it**
  - ❖ sets its variable $elected_i$ ← id of the message.
  - ❖ forwards the message, unless it is the new coordinator.



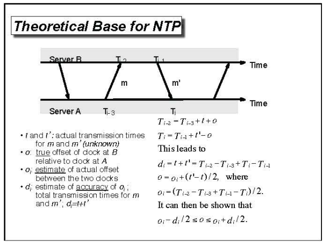
Algorithm 3: Bully Algorithm

- ❖ **When a process finds the coordinator has failed, if it knows its id is the highest, it elects itself as coordinator, then sends a *coordinator* message to all processes with lower identifiers than itself**
- ❖ **A process initiates election by sending an *election* message to only processes that have a higher id than itself.**
  - ☐ If no answer within timeout, send coordinator message to lower id processes → Done.
  - ☐ if any answer received, then there is some non-faulty higher process → so, wait for coordinator message. If none received after another timeout, start a new election.
- ❖ **A process that receives an "election" message replies with *answer* message, & starts its own election protocol (unless it has already done so)**



Reliable R-Multicast Algorithm

```
On initialization
    Received := {};

For process p to R-multicast message m to group g
    B-multicast(g, m);        // $p \in g$ is included as a destination

On B-deliver(m) at process q with g = group(m)
    if ($m \notin Received$)
    then
            Received := Received $\cup$ {m};
            if ($q \neq p$) then B-multicast(g, m); end if
            R-deliver m;
    end if
```



Theoretical Base for NTP

- *t* and *t'*: actual transmission times for *m* and *m' (unknown)*
- *o*: true offset of clock at *B* relative to clock at *A*
- o_i: estimate of actual offset between the two clocks
- d_i: estimate of accuracy of o_i ; total transmission times for *m* and *m'*; $d_i = t + t'$

$$T_{i-2} = T_{i-3} + t + o$$

$$T_i = T_{i-1} + t' - o$$

This leads to

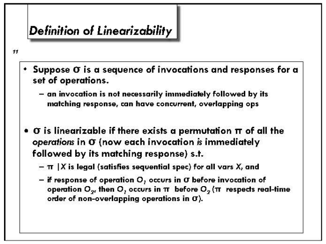
$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

$$o = o_i + (t' - t)/2, \quad \text{where}$$

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2.$$

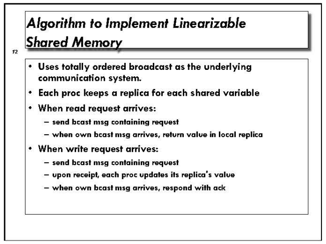
It can then be shown that

$$o_i - d_i/2 \le o \le o_i + d_i/2.$$



Definition of Linearizability

- **Suppose σ is a sequence of invocations and responses for a set of operations.**
  - an invocation is not necessarily immediately followed by its matching response, can have concurrent, overlapping ops
- **σ is linearizable if there exists a permutation π of all the *operations* in σ (now each invocation *is* immediately followed by its matching response) s.t.**
  - π |*X* is legal (satisfies sequential spec) for all vars *X*, and
  - if response of operation O_1 occurs in σ before invocation of operation O_2, then O_1 occurs in π before O_2 (π respects real-time order of non-overlapping operations in σ).



Algorithm to Implement Linearizable Shared Memory

- **Uses totally ordered broadcast as the underlying communication system.**
- **Each proc keeps a replica for each shared variable**
- **When read request arrives:**
  - send bcast msg containing request
  - when own bcast msg arrives, return value in local replica
- **When write request arrives:**
  - send bcast msg containing request
  - upon receipt, each proc updates its replica's value
  - when own bcast msg arrives, respond with ack

What is distributed computing?

What is distributed computing?

Parallel computing versus *distributed* computing

Example:

To add N numbers where N very large
use 4 processors, each adding up N/4,
then add the 4 partial sums

Parallel or distributed ?

What is distributed computing?

- *Parallel* computing versus *distributed* computing
- Role of uncertainty in distributed systems
  - Clock drift
  - Network delays
  - Network losses
  - Asynchrony
  - Failures

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

-- Leslie Lamport

What is distributed computing?

- *Parallel* computing versus *distributed* computing
- Role of uncertainty in distributed systems
  - Clock drift
  - Network delays
  - Network losses
  - Asynchrony
  - Failures

Clocks

- Notion of *time* very useful in real life, and so it is in distributed systems

- Example …

Submit programming assignment
by e-mail by 11:59 pm Monday

By which clock ?

How to synchronize clocks?

How to synchronize clocks?

Role of delay uncertainty

Ordering of Events

- If we can’t have “perfectly” synchronized clocks, can we still determine what happened first?

What is distributed computing?

- *Parallel* computing versus *distributed* computing
- Role of uncertainty in distributed systems
  - Clock drift
  - Network delays
  - Network losses
  - Asynchrony
  - Failures

Mutual Exclusion

- We want only one person to speak
- Only the person holding the microphone may speak
- Must acquire microphone before speaking

Mutual Exclusion

- How to implement in a message-passing system?

Mutual Exclusion

- What if messages may be lost?

What is distributed computing?

- *Parallel* computing versus *distributed* computing
- Role of uncertainty in distributed systems
  - Clock drift
  - Network delays
  - Network losses
  - Asynchrony
  - Failures

Agreement

- Where to meet for dinner?

Agreement with Failure

- Non-faulty nodes must agree

Agreement with
Crash Failure & Asynchrony

What if nodes misbehave?

- Crash failures are benign
- Other extreme … Byzantine failures

Agreement with Byzantine failures (synchronous system)

How to improve system availability?

- Potentially large network delays … network partition
- Failures

Replication is a common approach

Consider a storage system

- If data stored only in one place, far away user will incur significant access delay

➔ Store data in multiple replicas,

Clients prefer to access “closest” replica

Replicated Storage

- How to keep replicas “consistent” ?
- What does “consistent” really mean?

What’s this course about?

- Learn to “reason” about distributed systems

  … not just facts, but principles

- Learn important canonical problems, and some solutions

- Programming experience

- In class: we will focus on principles
- Supplemental readings: read about practical aspects, recent industry deployments

Distributed Computing … our scope

- Communication models:
  - message passing
  - shared memory
- Timing models:
  - synchronous
  - Asynchronous
- Fault models
  - Crash
  - Byzantine

Shared Memory

- Different processes (or threads of execution) can communicate by writing to/reading from (physically) shared memory

Shared Memory

Distributed Shared Memory

- The “shared memory” may be simulated by using local memory of different processors

Distributed Shared Memory

Key-Value Stores

Consistency Model

- Since shared memory may be accessed by different processes concurrently, we need to define how the updates are observed by the processes

- *Consistency model* captures these requirements

Consistency #1

Alice: My cat was hit by a car.

Alice: But luckily she is fine.

Bob: That’s great!

What should Calvin observe?

Consistency #1

Alice: My cat was hit by a car.

Alice: But luckily she is fine.

Bob: That’s great!

What should Calvin observe?

Consistency #2

Alice: My cat was hit by a car.

Alice: But luckily she is fine. Bob: That's terrible!

What should Calvin observe?

Consistency #2

Alice: My cat was hit by a car.

Alice: But luckily she is fine.

Bob: That’s terrible!

What should Calvin observe?