Some *Classical Distributed Systems* Algorithms
(Time,) Locks, Consensus, Election, Distributed Transactions
The General Problem

An ensemble of processes cooperate on a common task
• Communicate via messages (over a network)
• Share objects
• Messages can get corrupted or lost and processes (their execution nodes) can fail
• Occasionally have to agree on what happened and/or on what to do

An important example: distributed transaction
Distributed Transaction

BeginTransaction()

Lock(a)

a.DoOperation()  /* a.withdraw(20)*/

Lock(b)

b.DoOperation()  /* b.deposit(20)*/

EndTransaction(); => UnLock(a, b, ...)

or in case of failures:

AbortTransaction(); => UnDo all Operations, UnLock
ACID Properties (Härder, Reuter)

Atomicity
   all or nothing

Consistency
   from one consistent state to another

Isolation
   can execute in parallel, but same effect as one possible serial execution

Durability
   affected state saved in persistent memory
Questions

Locks in distributed systems?
Agreement on success of transaction?
Recovery from message loss and node crashes?

Algorithms:
• (Time and Causality (Lamport Time), more in Real-Time Systems)
• Locking
• Leader Election
• Consensus (Agreement)
• Distributed Transactions (2 phase locking, 2 phase commit)
Omitting many important other topics: replication, ...
Time and Causality

Temporal vs. Causal Order of Events

Temporal Order:
• induced by (perfect) timestamp
• perfect time stamp is not possible

Causal Order:
• induced by some causal dependency between events
An Example (Tanenbaum)

Imperfect Timestamps can be misleading in establishing causal dependency.
Causal Order (for Computer Generated Events)

Definition:

Partial Order for Computer Generated Events

\( a \rightarrow b \) “a causes b”

called: “happened before” or “causally dependent”

1. if a, b events within a sequential process
   then \( a \rightarrow b \), if a is executed before b.

2. If a is „sending of a message“ by a process and
   b the „reception of that message“ by another process,
   then \( a \rightarrow b \).

3. \( \rightarrow \) is transitive.
Clocks: Physical and Logical (course: Real-Time Systems)

Physical Clocks
- devices to measure time
- necessarily imperfect (more later)

Problems:
- how to create knowledge about causal dependency of computer events without relying on physical clocks?
  => Logical Clocks
- how to establish a $x$ certainly occurred after $y$ relation (temporal order) for environmental events
  => Global Time
Logical Clocks

Definitions:
- monotonically increasing SW counters (COULOURIS)
- clocks on different computers that are somehow consistent (LAMPORT)

Events: \( a, b: a \rightarrow b \): \( a \) causes \( b \) (causally dependent)

Timestamps: \( C(a), C(b) \)

Potential Requirements for logical clocks:
- \( a \rightarrow b \implies C(a) < C(b) \)
- \( a \rightarrow b \iff C(a) < C(b) \)
A

B

C

D
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>30</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>18</td>
<td>40</td>
<td>24</td>
<td>40</td>
</tr>
<tr>
<td>24</td>
<td>59</td>
<td>32</td>
<td>59</td>
</tr>
<tr>
<td>30</td>
<td>60</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>36</td>
<td>70</td>
<td>55</td>
<td>70</td>
</tr>
<tr>
<td>42</td>
<td>80</td>
<td>56</td>
<td>80</td>
</tr>
<tr>
<td>73</td>
<td>90</td>
<td>64</td>
<td>90</td>
</tr>
<tr>
<td>74</td>
<td>100</td>
<td>72</td>
<td>100</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>80</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A, B, C, D表示不同的节点和方向。
Lamport’s Logical Clocks

- each Process has local clock $LC_i$
- tick:
  - with each local event $e$:
    $LC_i := LC_i + 1; e$
  - with each sending of a message by process $Pi$:
    $LC_i := LC_i + 1; \text{send} (LC_i,m)$
  - with each reception of a message $\langle M,LC_m \rangle$ by $Pj$:
    $LC_j := \max(LC_m, LC_j); LC_j := LC_j + 1$
Lamport Clocks

Properties:

• \(a \rightarrow b \Rightarrow C(a) < C(b)\),
  but not:
  \(C(a) < C(b) \Rightarrow a \rightarrow b\)

• partial order

sometimes total order convenient:
  extend it: e.g.: LC.Process_number
Locks

General:
• no assumptions about relative speeds
• no process outside the critical section should block another one from entering it

Safety:
• no two processes in critical section at the same time

Livelihood:
• requests to enter and exit the critical section eventually succeed
Centralized Algorithm

Use a Lock-Server process C.
Lock(KA) ... Unlock(KA)

lock(KA)
send a request message to C;
wait for an “ok” message from C
C: either sends “ok” or enters request into a queue

unlock(KA)
sends a release message.
C: inspects its queue and sends “ok” message to “oldest” request
Example

Diagram showing a distributed system with nodes labeled 0, 1, 2, and a central node labeled C. The diagram illustrates communication and state transitions between nodes. Nodes communicate using messages labeled 'rq' and 'ok', and the system transitions from an initial state to a final state.
A Distributed Algorithm
LAMPORT 1978; RICART/AGRAWALA 1981

Based on Logical Clock. Requirements:
• Timestamps must induce total order.
• $a \rightarrow b \Rightarrow C(a) < C(b)$ (Lamport Clock)

processes: $P_1, \ldots, P_n$,
local variables of each process:
  state: (released, wanted, held); /* On init: state = RELEASED;
  TimeMyPendingRequest

messages:
  request from $P_i$ to all other Processes, notation: $P_i T_i$
  ok-to-enter

  time stamp of request
A Distributed Algorithm
LAMPORT 1978; RICART/AGRAWALA 1981

Lock:
state = WANTED;
TimeOfMyLastRequest = LogicalTime;
request.T = TimeOfMyLastRequest;
Multicast to all processes: request;

Wait until (number of ok-to-enter messages received == (n - 1));
state := HELD;
Simple Example

p1

1,25

p2

1,25

p3
ContendedExample

1. p1
2. p2
3. p3

Connections:
- p1 to p2: 2, 8
- p1 to p3: 3, 12
- p2 to p3: 3, 12
Distributed Algorithm

On receipt of a request \(<T_m; p_m>\) at \(p_j\) \((m \neq j)\):

if ( state == HELD or ( state == WANTED

and TimeOfMyPendingRequest < T_m ) )

then

queue request from \(p_i\) without replying;

else

reply “ok-to-enter” immediately to \(p_i\);

end if

To release token:

state := RELEASED;

reply “ok-to-enter” to all queued requests;
ContendedExample

Diagram of processes p1, p2, and p3 with arrows indicating communication or dependencies between them.
Election

- elect one out of set of processes (e.g., coordinator)
- Processes $P_1 \ldots, P_n$ ordered, e.g. based on indices
- select one with highest index
- processes know indices of all other involved
- should work, even when one process fails during election
Requirements

Safety:
  at any time, each process has either no coordinator or
  the one which -after a run- has the highest ID among the non-crashed
  processes

Liveness:
  all processes eventually elect a coordinator
Bully-Algorithm

Three types of messages:

- *Election* - message to trigger an election
- *Answer* - message to respond to an election message
- *Coordinator* - message to announce itself as (new) coordinator

Notation: from the point of view of $P_i$

- $P_j$ with $j>i$, are “higher” processes
- $P_j$ with $j<i$, are “lower” processes

Synchronous communication (timeout to detect crashes)
Bully-Algorithmus, f2

P discovers the crash of the coordinator, 
starts an Election by 
sending election messages to all “higher” processes 
and waits for an answer message

if P receives an election message from a “lower” process 
then it returns an answer message.

if P does not receive an answer message for his election-message, 
then he is the coordinator and sends a coordinator message to all 
lower processes
Bully-Algorithmus, f3

- Once P has received an *Answer* message, it waits for a *Coordinator* message.
  - if P does not receive a *Coordinator* message (i.e., the “in spe” coordinator process has crashed) then P restarts election.
- (if P receives an *election* message from a “lower” process then it returns an *answer* message.)
  - If P in this situation does not receive a *Coordinator*-message, then P starts election.
- if a crashed process is restarted, it starts election.
Bully-Algorithmus, f4

Complexity:

- $o(n^2)$
- happens, when $p_0$ discovers crash of PN

Problems:

- reliable communication
- known process topology
Distributed Transactions as important example

BeginTransaction()

a.DoOperation()  /* a.withdraw(20) */

b.DoOperation()  /* b.deposit(20) */

EndTransaction();

or in case of failures:

AbortTransaction(); => UnDo all Operations
Locking (2 Phase)

BeginTransaction()

Lock(a)

a.DoOperation() /* a.withdraw(20)

Lock(b)

b.DoOperation() /* b.deposit(20)

EndTransaction(); => UnLock(a, b, ...)

or in case of failures:

AbortTransaction(); => UnDo all Operations, UnLock
Distributed Transactions

Coordinator

Node a

→

Participant

Node b
Two Army Problem (Coordinated Attack)

p, q processes
communicate using messages
messages can get lost
no upper time for message delivery known
do not crash, do not cheat

d, q to agree on action (e.g. attack, retreat, ...)

how many messages needed?

first mentioned: Jim Gray 1978
Two Army Problem (Coordinated Attack)

Result:
there is no protocol with finite messages

Prove:
by contradiction
assume there are finite protocols \((m_{p\rightarrow q}, m_{q\rightarrow p})^*\)
choose the shortest protocol MP,
last message MX: \(m_{p\rightarrow q}\) or \(m_{q\rightarrow p}\)
MX can get lost
=> must not be relied upon => can be omitted
=> MP not the shortest protocol.
=> no finite protocol
Byzantine Agreement

n processes, f traitors, n-f loyals
communicate by reliable and timely messages
(synchronous messages)
traitors lie, also cheat on forwarding messages
try to confuse loyals

goal:
loyals try to agree on action (attack, retreat)
more specific:
one process is commander
if commander is loyal and gives an order,
loyals follow the order
otherwise loyals agree on arbitrary action
3 Processes: 1 traitor, 2 loyals

- Commander
- Lt. Commander
- Lt. Commander

Commander attacks Lt. Commander,Lt. Commander attacks Lt. Commander.

Lt. Commander says: retreat.
3 Processes: 1 traitor, 2 loyals

he said: retreat

3 processes not sufficient to tolerate 1 traitor
4 Processes

- Commander
  - attack to Lieutenant
  - attack to Lieutenant
  - attack to Lieutenant

- Lieutenant
  - attack from Commander
  - attack to Lieutenant 1
  - he said: attack

- Lieutenant 1
  - he said: attack

- Lieutenant
  - he said: retreat
all lieutenant receive x, y, z

can decide

General result:

3 f + 1 processes needed to tolerate f traitors
Distributed Transactions

one coordinator and several participants
goal: agree on Commit or Abort

failure assumptions:
  - messages can get lost, corrupted messages can be detected
  - node can crash, crashed nodes can be detected by failure detector
  - no byzantine failures

transactions are well formed (“2 Phase locking”):
  - all objects are locked before operations
  - locks freed at commit or abort of transaction

coordinator and participants have persistent LOGs
  can be used for recovery from node crashes
Operation

Coordinator

Participant 1 provides a
join(TID)
implements a.withdraw

Participant 2 provides b
join(TID)

BeginTransaction(out TID)
• a.withdraw(TID, ...)
• b.withdr(TID, ...)
EndTransaction

Client
Operation

Coordinator

Participant 1
provides a
join(TID)
implements a.withdraw

Participant 2
provides b
join(TID)

BeginTransaction(out TID)
• a.withdraw(TID, ...)
• b.withdr(TID, ...)

EndTransaction(TID)

Client
Operation

Coordinator

Participant 1
provides a
join(TID)

Participant 2
provides b
join(TID)

BeginTransaction(out TID)
- a.withdraw(TID, ...)
- b.withdr(TID, ...)

EndTransaction

Client

commit protocol

BeginTransaction(out TID)
- a.withdraw(TID, ...)
- b.withdr(TID, ...)

EndTransaction
2 Phase Commit Protocol

coordinator

participant 1

participant 2

canCommit?

wait for votes

prepared

state: uncertain -> Log

state: committed -> Log

DoCommit

done

state: committed -> Log
Failures

Messages

• coordinator gets no response
  before commit -> abort
  after commit -> block and retry

• participant in uncertain state gets no DoCommit
  -> ask coordinator
Failures

Crashes

• any node:
  restart from checkpoint
  use logs to redo all operations
  (commit, abort, prepare, operations on objects)

• coordinator crashes, restarts from log
  before commit -> transaction aborts,
  after commit
  -> block and retry 2. Phase

• participant in uncertain state, restarts from log
  -> ask coordinator
More such classical algorithms

- Vector time
- Message delivery orders
- distributed snapshots
- deadlock detection
- ...

Hermann Härtig, TU Dresden
Literature

Coulouris ... used for this lecture
all text books on distributed systems