COMP90020: Distributed Algorithms

14. Distributed (Decentralized) Transactions

Decentralized Locking and Deadlock Detection

Miquel Ramirez

THE UNIVERSITY OF MELBOURNE

Semester 1, 2019

Agenda

1. Decentralized Concurrency Control
2. Distributed Deadlock Detection
3. Biblio & Reading

Agenda

Concurrency Control in a Decentralized Setting

Many (challenging) issues

- Concurrency control needed in every server to guarantee consistency
  - This will reduce performance!
- For locking we need to extend protocols so
  - Processes running at different servers can lock objects,
  - servers can crash, run at different speeds, etc.
- For optimistic concurrency control we need to
  - Validate transaction at multiple servers before committing,
  - guarantee that same criterion used at every server,
  - e.g. in FARM the coordinator is the only one that decides whether to lock an object and explicitly tells participants when to do so.

Decentralized Locks

Reminder: Locks control availability of shared objects

Basic Facts

- Lock manager held at the same server as objects
- To acquire lock: contact server
- To release: must delay until transactions committed or aborted at all the servers involved in the transaction

Outstanding Issues

- Locks acquired independently, consistency difficult to enforce
- cyclic dependencies can arise,
- distributed deadlock detection and resolution needed.

Issues with Decentralized Locking

- Lock managers set locks independently of each other
- Each keep local wait-for graphs

	Transaction t			Transaction s		
$Time$	Operation	Server	Locking Event	Operation	Server	Locking Event
0	$write(A)$	X	locks A	–	–	–
1	–	–	–	$write(B)$	Y	locks B
2	$read(B)$	Y	waits for s	–	–	–
3	–	–	–	$read(A)$	X	waits for t

$\rightarrow$ So it is possible impose incompatible schedules

$\rightarrow$ or end up with cyclic dependencies like the one above

Optimistic Concurrency Control

Reminder: Alternative to locking (avoids overhead and deadlocks)

For a single server

- Transactions allowed to proceed but
- validated before allowed to commit, if conflict arises may be aborted.
- Transactions given numbers at the start of validation,
- and are serialized according to this order.

In distributed transactions

- Must be validated by multiple independent servers during voting phase of 2PC protocol,
- global validation needed to serialize across servers.

Commitment Deadlock

Optimistic Concurrently Control is highly optimized which makes it brittle too.

Consider this schedule of transactions t and s, accessing objects A and B on servers X and Y

	Transaction t		Transaction s	
$Time$	Operation	Server	Operation	Server
0	$read(A)$	X	$read(B)$	Y
1	$write(A)$	–	$write(B)$	–
2	$read(B)$	Y	$read(A)$	X
3	$write(B)$	–	$write(A)$	–

t and s start validation at the same time, but X validates t first, and Y validates s first

- we've got a deadlock, since validation is implemented as a critical section!

Phantom Deadlocks

Definition

When a deadlock detection algorithm has a false positive

Example

Consider scenario with two transactions t, u and v and two servers X and Y hosting objects A and B,

- Coordinator keeps a *global* wait-for graph, X and Y keep local graphs and notify the coordinator when updated.
- u has locked A as it is working on it,
- t makes requests to X and Y concurrently over objects A and B
- at the same time, u releases A, and tries to access B
- if messages to coordinator delayed, the coordinator determines a deadlock exists!

Agenda

1. Decentralized Concurrency Control
2. Distributed Deadlock Detection
3. Biblio & Reading

Communication and resource deadlock

Crucial problem in DS, all processes waiting for something to happen

Communication Deadlock

Every process is waiting for some other process to send message to group

Resource Deadlock

Every process waiting for other process to release lock on shared object

Communication and resource deadlock

Crucial problem in DS, all processes waiting for something to happen

Communication Deadlock

Every process is waiting for some other process to send message to group

Resource Deadlock

Every process waiting for other process to release lock on shared object

Both types of deadlock are captured by

N-out-of-M model

A process can wait for N grants out of M requests.

Examples:

- A process waits for one message from a group of processes: $N = 1$
- A transaction needs to lock several shared objects: $N = M$.

Deadlocks and Consensus

Question!

In the N-out-of-M model, could we have $M > N > 1$?

(A): Yes

(B): No

Deadlocks and Consensus

Question!

In the N-out-of-M model, could we have $M > N > 1$?

(A): Yes (B): No

$\rightarrow$ (Yes): Consider a transaction doing operations over 2 shared objects, which are managed by 2 servers each (a primary one and a replica). Another example is a transaction that invokes a Consensus algorithm, requiring M processes to decide for a value, that uses the $\mathrm{majority}$ function (e.g. $N = M/2 + 1$).

Generalized Wait-for Graphs

Wait-for Graph

Directed graph $W = (V, E)$ where

- V is set of processes p, q, u, v, w, x, etc.
- $E = \{(p, Q) \mid, p$ is process, p has outstanding request to $Q \subset V \}$

Keeps track of which nodes in V are blocked or non-blocked

- an edge is added whenever p sends request to processes Q,
- an edge is removed whenever $q \in Q$ grants request to p,
- nodes are non-blocked if they do not have any outgoing edges.

Notes:

- When $|Q| > 1$ the edge represents a one-to-many request.
- Transactions usually generate multiple edges.
- A process can have multiple nodes in the graph.

Wait-For Graph - Communication Dependencies

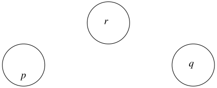
Suppose process p must wait for a message from process q.

Node p sends a request to node q. Then edge pq is created in the wait-for graph, and p becomes blocked.

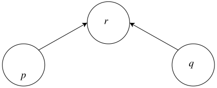
When q sends message to p, the request of p is granted. Then edge pq is removed from the wait-for graph, and p becomes unblocked.

Wait-For graph - Resource Dependencies

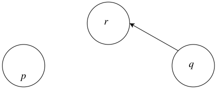
Suppose two processes p and q want to claim a resource managed by process r.



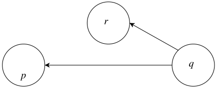
In the wait-for graph, nodes p, q send a request to node r representing the resource. Edges pr and qr are created.



Since the resource is free, the resource is given to say p. So r sends a grant to p. Edge pr is removed.

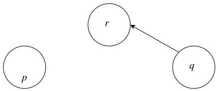


The concurrency control protocol in place requires resources to be released by p before q can claim it. So q sends a request to p, creating edge qp in the wait-for graph.

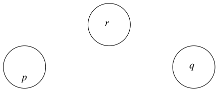


Wait-For graph - Resource Dependencies (cont'd)

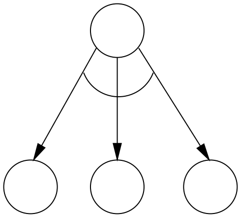
As p releases the resource, p is granting the request of q. Edge qp is removed.



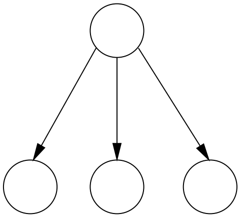
The resource is given to q, and r grants the request from q. Edge qr is removed.



Drawing wait-for graphs



AND (3–out–of–3) request



OR (1–out–of–3) request

- Left: all requests need to be granted before process is unblocked.
- Right: only one request needs to be granted to continue execution.

Static Analysis on Wait-for Graphs

1. A snapshot of DS is taken to construct the Wait-For graph W.
2. Apply rules below until a fixed point is reached

- Non-blocked nodes can grant requests.
- When a request is granted, the corresponding edge is removed.
- When an N-out-of-M request has received N grants, the requester becomes unblocked.

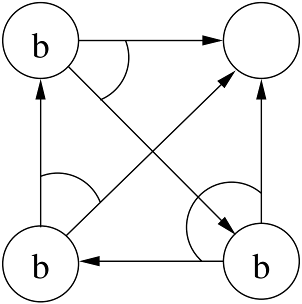
  The remaining $M - N$ outgoing edges are dismissed.

Deadlock Condition

When no more grants are possible, nodes that remain blocked in W are deadlocked.

Static Analysis - Example 1

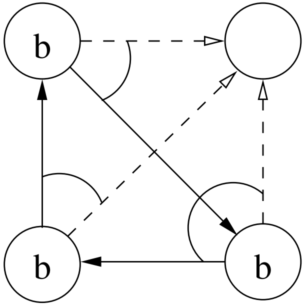
We have three processes blocked after crossing 2-out-of-2 requests as per below



Is there a deadlock ?

Static Analysis - Example 1

We have three processes blocked after crossing 2-out-of-2 requests as per below

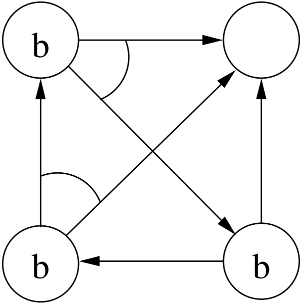


Deadlock

If resource was granted we still have a three way cycle.

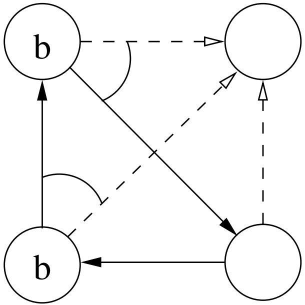
Static analysis - Example 2

Now we have again three processes blocked, two have send 2-out-of-2 requests, and the third one a 1-out-of-2 request.



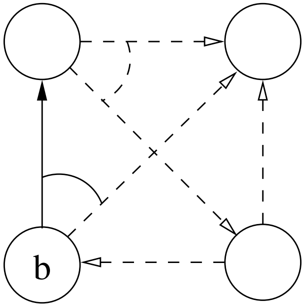
Static analysis - Example 2

Now we have again three processes blocked, two have send 2-out-of-2 requests, and the third one a 1-out-of-2 request.



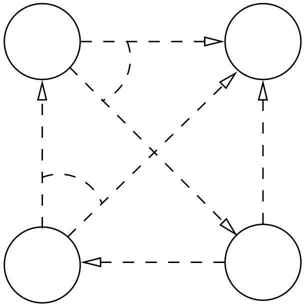
Static analysis - Example 2

Now we have again three processes blocked, two have send 2-out-of-2 requests, and the third one a 1-out-of-2 request.



Static analysis - Example 2

Now we have again three processes blocked, two have send 2-out-of-2 requests, and the third one a 1-out-of-2 request.



No deadlock

Bracha-Toueg Algorithm - Overview

Purpose

Provides distributed method to perform the analysis to clean out a Wait-For Graph to uncover deadlocks.

→NO GLOBAL WAIT-FOR GRAPH

Assumptions

- Communications between processes in DS is undirected network,
- the Wait-For graph W has been computed,
- runs parallel to *whatever* the DS is doing.

Think of it as distributed reformulation of classic Tarjan's algorithm.

Distributed Versus Centralized Wait-For Graphs

Question!

Why would we want to decentralize deadlock detection?

(A): Security
(B): Reliability
(C): Performance
(D): Privacy

Distributed Versus Centralized Wait-For Graphs

Question!

Why would we want to decentralize deadlock detection?

(A): Security

(B): Reliability

(C): Performance

(D): Privacy

→ (A): Complicates adversarial penetration and disruption.

→ (B): If M-out-of-N requests form sparse graph, the DS can retain a significant degree of functionality.

→ (D): Preempts eavesdropping and involuntary dissemination of privileged information.

Bracha-Toueg Algorithm - Snapshots

Initiation

Process p suspects it is deadlocked, initiates a Snapshot Algorithm (e.g. Chandy-Lamport or Lai-Yang) to compute Wait-For graph.

Each node u takes a local snapshot of:

- requests it sent or received still to be granted or dismissed,
- grant and dismiss messages in edges.

Then it computes sets:

Out_u: the nodes it sent a request to (not granted)

In_u: the nodes it received a request from (not dismissed)

Bracha-Toueg Algorithm - Grant Resolution

Each node proceeds to resolve grants similarly as we did statically

Initially

- $requests_u$ is the number of grants u requires to become unblocked

$requests_u$ is updated as follows:

1. When u receives a grant message, $requests_u \leftarrow requests_u - 1$.
2. If $requests_u$ becomes 0, u sends grant messages to all nodes in In_u.

Once deadlock detection run finished:

- If $requests > 0$ at initatior process, then it is deadlocked.

Challenge

The initiator process must detect *termination* of deadlock detection

- Non–trivial problem, many termination detection algorithms exist offering very different guarantees.
- Bracha-Toueg solves this in a specialised manner.

Bracha-Toueg Algorithm - Termination Detection

Initially $notified_u = false$ and $free_u = false$ at all nodes u.

The initiator starts a deadlock detection run executing *Notify*.

$Notify_u$: $notified_u \leftarrow true$
for all $w \in Out_u$ send NOTIFY to w
if $requests_u = 0$ then $Grant_u$
for all $w \in Out_u$ await DONE from w

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send GRANT to w
for all $w \in In_u$ await ACK from w

Parallel Processing

While a node is awaiting DONE or ACK messages, it can process incoming NOTIFY and GRANT messages.

Bracha-Toueg Algorithm - Callbacks

Nodes receving NOTIFY and GRANT handle incoming messages as follows:

Event: u receives NOTIFY

1. If $notified_u = false$, then u executes $Notify_u$.
2. u sends back DONE.

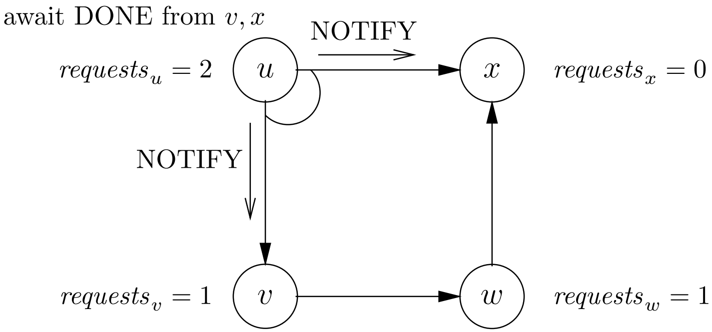
Event: u receives GRANT

1. If $requests_u > 0$, then $requests_u \leftarrow requests_u - 1$,
2. If $requests_u$ becomes 0, then u executes $Grant_u$.
3. u sends back ACK.

When initiatior has received DONE from all nodes in its *Out* set, it checks the value of its *free* field

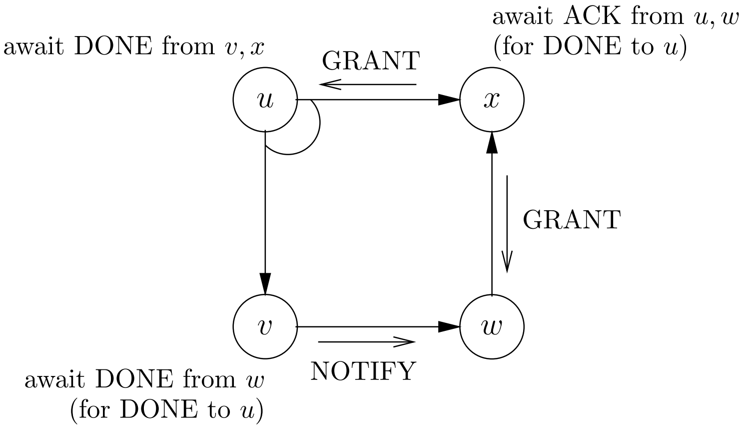
- If it is still *false*, the initiator concludes it is deadlocked.

Bracha-Toueg Algorithm - Example

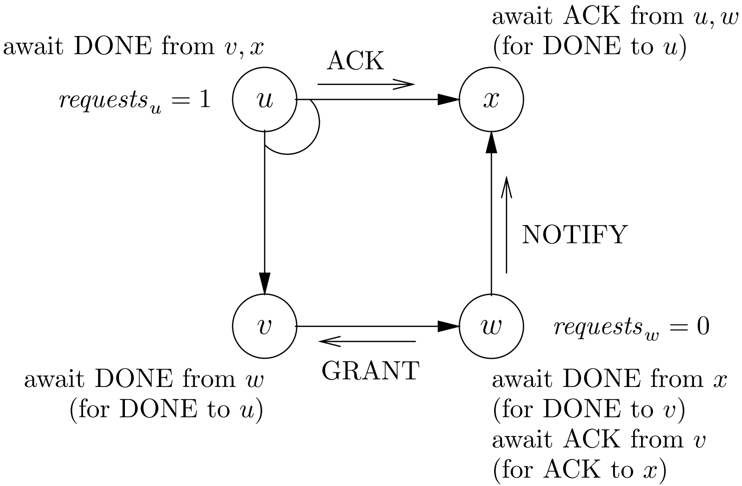


u is the initiator

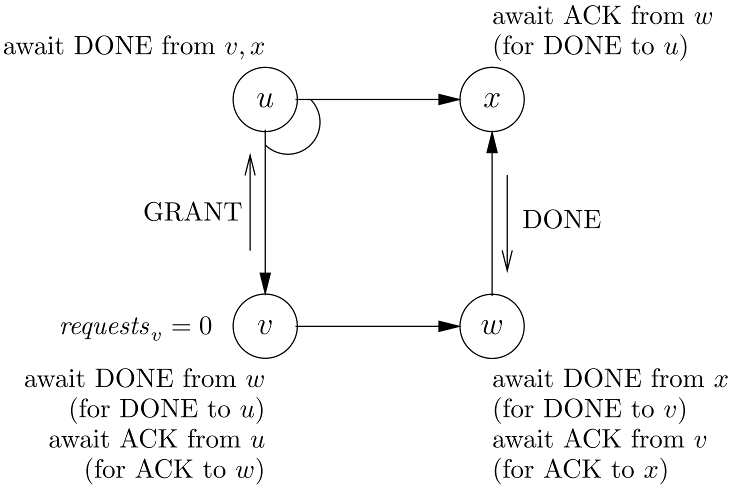
Bracha-Toueg Algorithm - Example



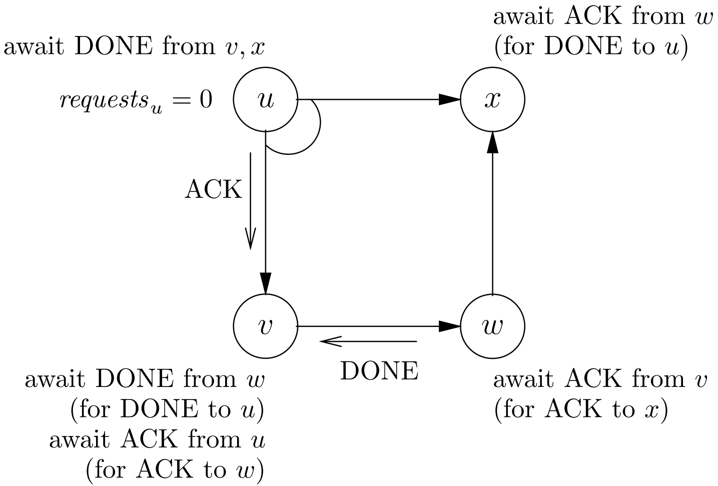
Bracha-Toueg Algorithm - Example



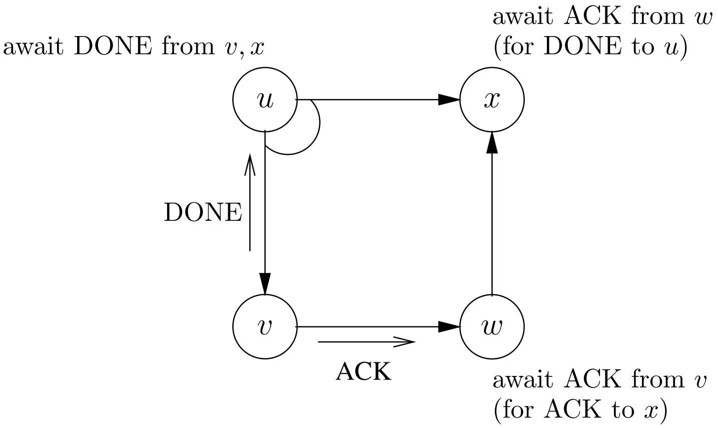
Bracha-Toueg Algorithm - Example



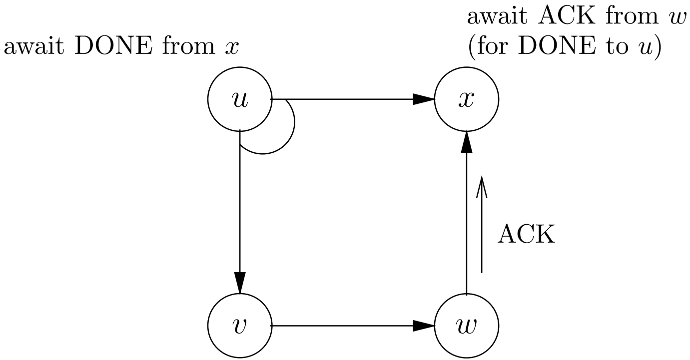
Bracha-Toueg Algorithm - Example



Bracha-Toueg Algorithm - Example

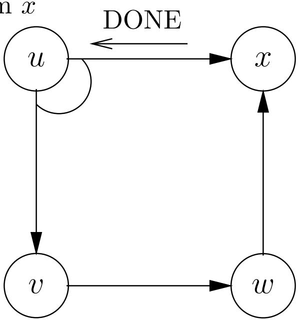


Bracha-Toueg Algorithm - Example

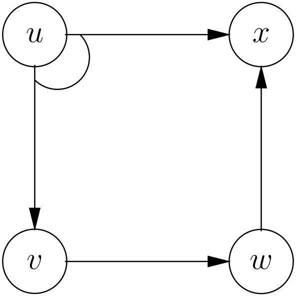


Bracha-Toueg Algorithm - Example

await DONE from x



Bracha-Toueg Algorithm - Example



$free_u = true$, so u concludes that it is not.

Bracha-Toueg Algorithm - Correctness

Guarantee: No Deadlocks Introduced

The Bracha-Toueg algorithm is deadlock-free:

- Initiator eventually receives DONE's from all nodes in its *Out* set.

At that moment the Bracha-Toueg algorithm has terminated.

Flow of calls can be represented as a tree (see Wave algorithms)

1. NOTIFY/DONE's construct a tree T rooted in the initiator.
2. GRANT/ACK's construct disjoint trees T_v, rooted in a node v where from the start $requests_v = 0$.

NOTIFY/DONE's only complete when all GRANT/ACK's have completed.

Bracha-Toueg Algorithm - Correctness

Proof Sketch

a. In a deadlock detection run, requests are granted as much as possible.

b. Therefore, if the initiator has received DONE's from all nodes in its *Out* set and its *free* field is still *false*, it is deadlocked.

c. Analogously, if its *free* field is *true*, there is no deadlock (yet), if and only if resource requests are granted nondeterministically.

Question

Question!

Could we apply the Bracha-Toueg algorithm to itself, to establish that it is a deadlock-free algorithm ?

(A): Yes

(B): No

Question

Question!

Could we apply the Bracha-Toueg algorithm to itself, to establish that it is a deadlock-free algorithm ?

(A): Yes

(B): No

→ (No): The Bracha-Toueg algorithm can only establish whether a deadlock is present in a snapshot of the DS "hosting" it. Otherwise, we would be solving the Halting Problem.

Agenda

Further Reading

Coulouris et al. *Distributed Systems: Concepts & Design*

- Chapter 17, Sections 4 & 5

Wan Fokkink's *Distributed Algorithms: An Intuitive Approach*

- Chapter 5, Deadlock Detection