
11455

Communication and resource deadlock

A deadlock occurs if there is a cycle of processes waiting until:

- another process on the cycle sends some input (communication deadlock)
- or resources held by other processes on the cycle are released (resource deadlock)

Both types of deadlock are captured by the N-out-of-M model:

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

Examples:

- A process is waiting for one message from a group of processes: $N = 1$
- A database transaction first needs to lock several files: $N = M$.

Communication and resource deadlock

A deadlock occurs if there is a cycle of processes waiting until:

- another process on the cycle sends some input (communication deadlock)
- or resources held by other processes on the cycle are released (resource deadlock)

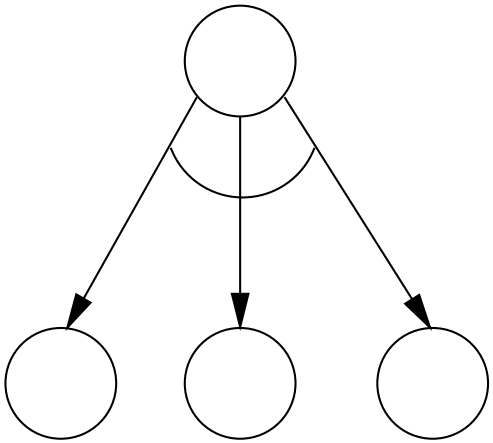
Both types of deadlock are captured by the N-out-of-M model:

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

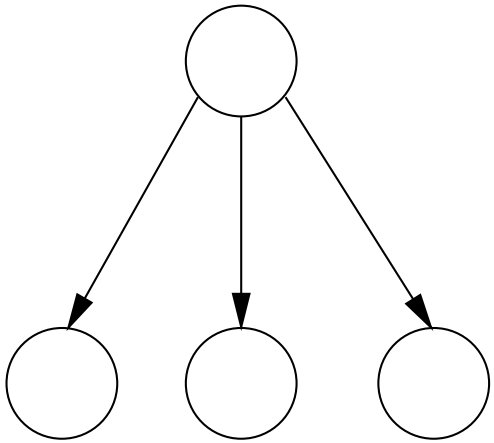
Examples:

- A process is waiting for one message from a group of processes: $N = 1$
- A database transaction first needs to lock several files: $N = M$.

Drawing wait-for graphs



AND (3–out–of–3) request



OR (1–out–of–3) request

Wait-for graph

A (non-blocked) process can issue a request to M other processes, and becomes blocked until N of these requests have been granted.

Then it informs the remaining $M - N$ processes that the request can be dismissed.

Only non-blocked processes can grant a request.

A (directed) wait-for graph captures dependencies between processes.

There is an edge from node p to node q if p sent a request to q that wasn't yet dismissed by p or granted by q.

Wait-for graph

A (non-blocked) process can issue a request to M other processes, and becomes blocked until N of these requests have been granted.

Then it informs the remaining $M - N$ processes that the request can be dismissed.

Only non-blocked processes can grant a request.

A (directed) wait-for graph captures dependencies between processes.

There is an edge from node p to node q if p sent a request to q that wasn't yet dismissed by p or granted by q.

Wait-for graph - Example 1

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

In the wait-for graph, 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 a 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 - Example 2

Suppose two processes p and q want to claim a resource.

In the wait-for graph, nodes u, v representing p, q send a request to node w representing the resource. Edges uw and vw are created.

Since the resource is free, the resource is given to say p.
So w sends a grant to u. Edge uw is removed.

The basic (mutual exclusion) algorithm requires that the resource must be released by p before q can claim it.
So w sends a request to u, creating edge wu in the wait-for graph.

After p releases the resource, u grants the request of w.
Edge wu is removed.

The resource is given to q. Hence w grants the request from v.
Edge vw is removed and edge wv is created.

Wait-for graph - Example 2

Suppose two processes p and q want to claim a resource.

In the wait-for graph, nodes u, v representing p, q send a request to node w representing the resource. Edges uw and vw are created.

Since the resource is free, the resource is given to say p.
So w sends a grant to u. Edge uw is removed.

The basic (mutual exclusion) algorithm requires that the resource must be released by p before q can claim it.
So w sends a request to u, creating edge wu in the wait-for graph.

After p releases the resource, u grants the request of w.
Edge wu is removed.

The resource is given to q. Hence w grants the request from v.
Edge vw is removed and edge wv is created.

Wait-for graph - Example 2

Suppose two processes p and q want to claim a resource.

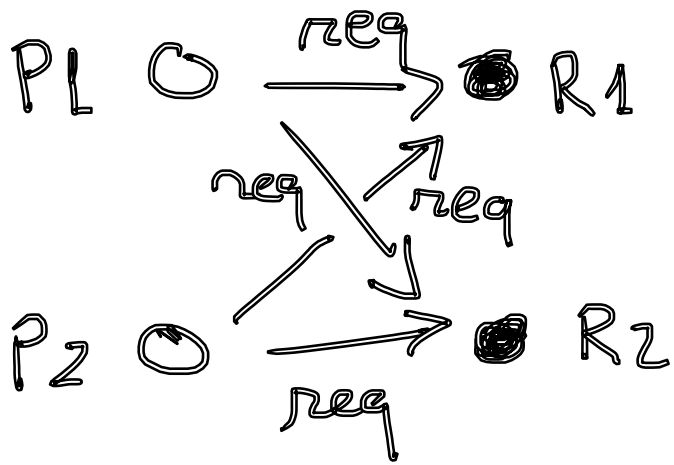
In the wait-for graph, nodes u, v representing p, q send a request to node w representing the resource. Edges uw and vw are created.

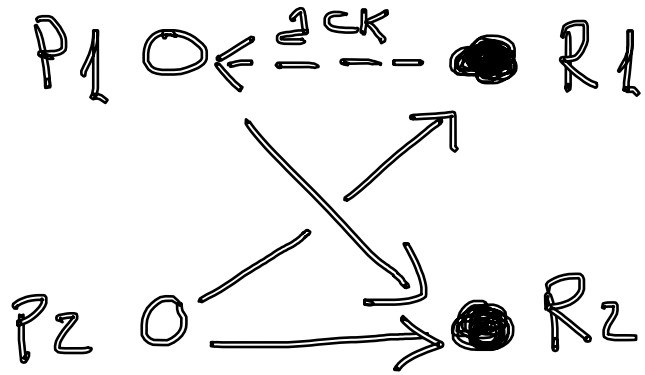
Since the resource is free, the resource is given to say p.
So w sends a grant to u. Edge uw is removed.

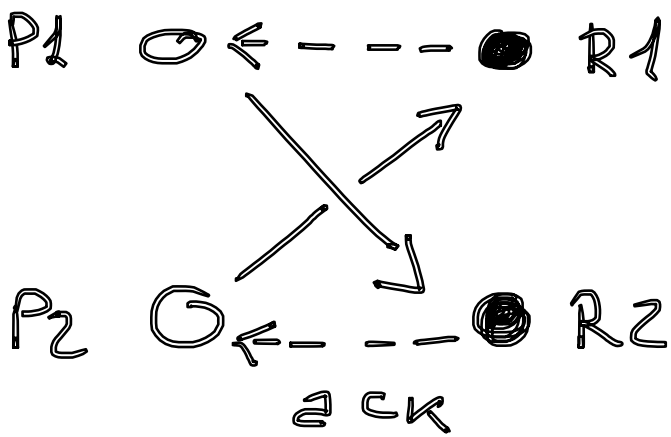
The basic (mutual exclusion) algorithm requires that the resource must be released by p before q can claim it.
So w sends a request to u, creating edge wu in the wait-for graph.

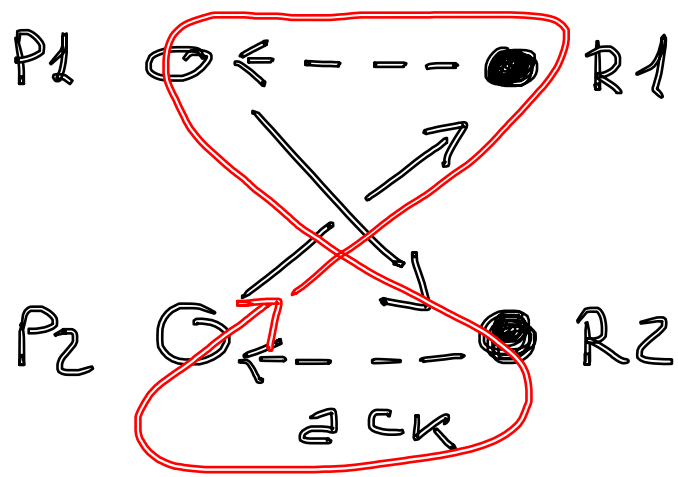
After p releases the resource, u grants the request of w.
Edge wu is removed.

The resource is given to q. Hence w grants the request from v.
Edge vw is removed and edge wv is created.


P1
req
R1
req
req
P2
R2
req


P1
ack
R1
P2
R2


P1
R1
P2
R2
ack


P1
R1
P2
R2
ack

deadlock

Static analysis on a wait-for graph

A snapshot is taken of the wait-for graph.

A *static analysis* on the wait-for graph may reveal deadlocks:

- 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.)

When no more grants are possible, nodes that remain blocked in the wait-for graph are deadlocked in the snapshot of the basic algorithm.

Static analysis on a wait-for graph

A snapshot is taken of the wait-for graph.

A *static analysis* on the wait-for graph may reveal deadlocks:

- 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.)

When no more grants are possible, nodes that remain blocked in the wait-for graph are deadlocked in the snapshot of the basic algorithm.

Static analysis on a wait-for graph

A snapshot is taken of the wait-for graph.

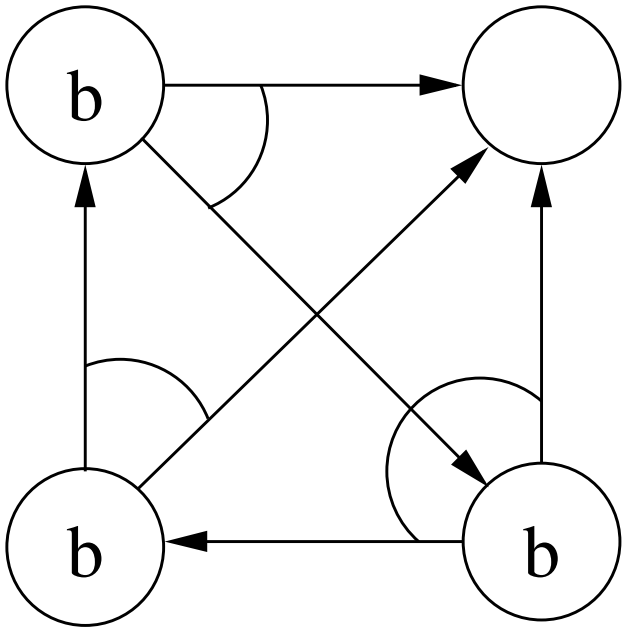
A *static analysis* on the wait-for graph may reveal deadlocks:

- 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.)

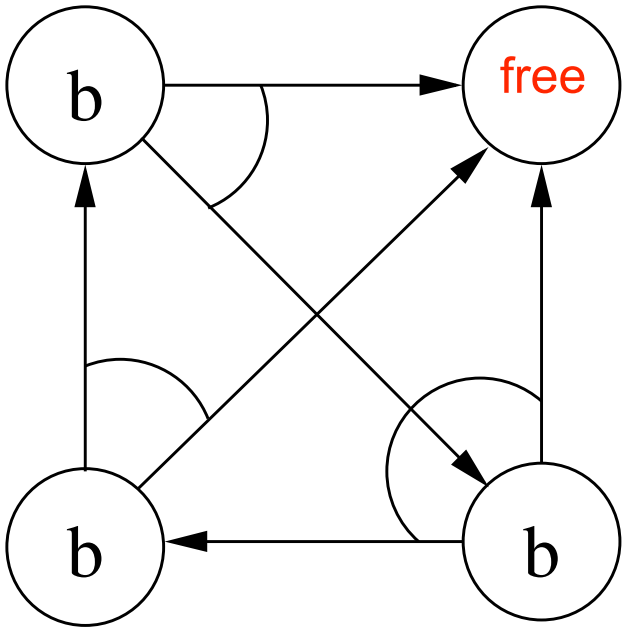
When no more grants are possible, nodes that remain blocked in the wait-for graph are deadlocked in the snapshot of the basic algorithm.

Static analysis - Example 1



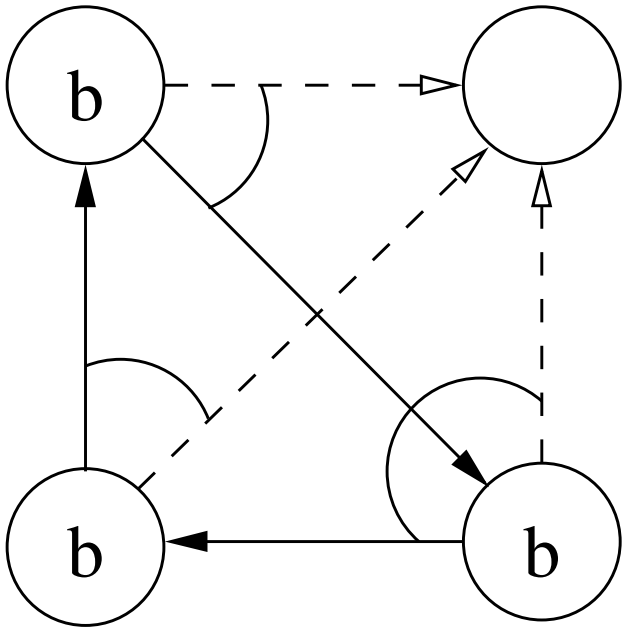
Is there a deadlock ?

Static analysis - Example 1



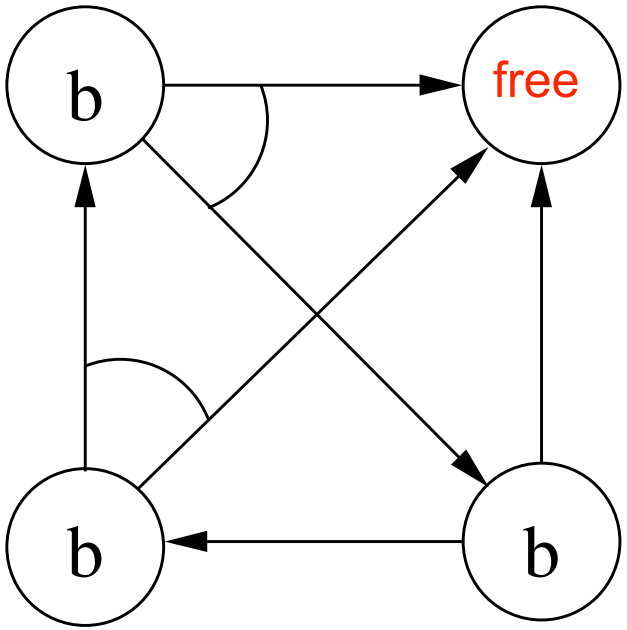
Is there a deadlock ?

Static analysis - Example 1

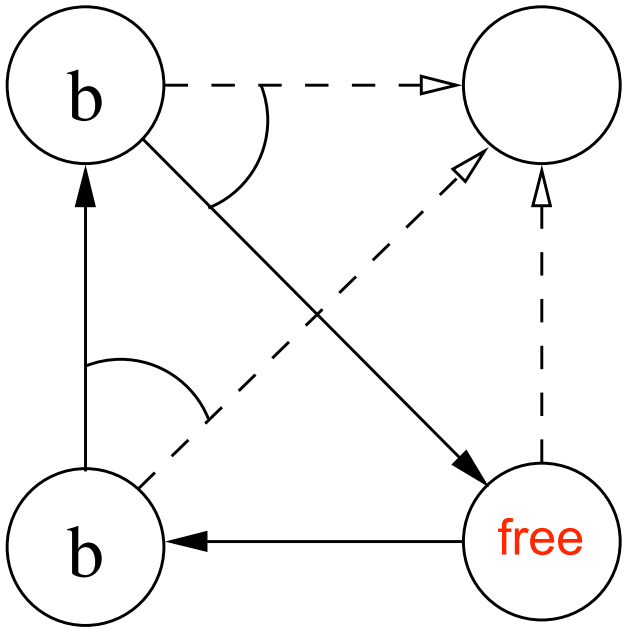


Deadlock

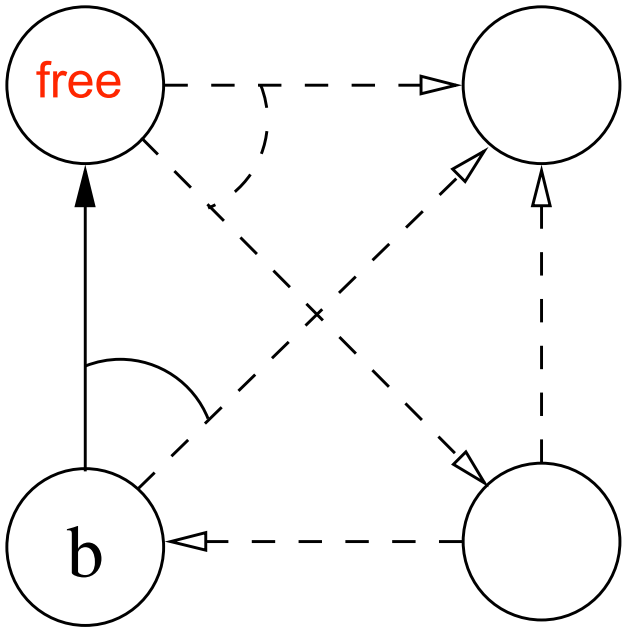
Static analysis - Example 2



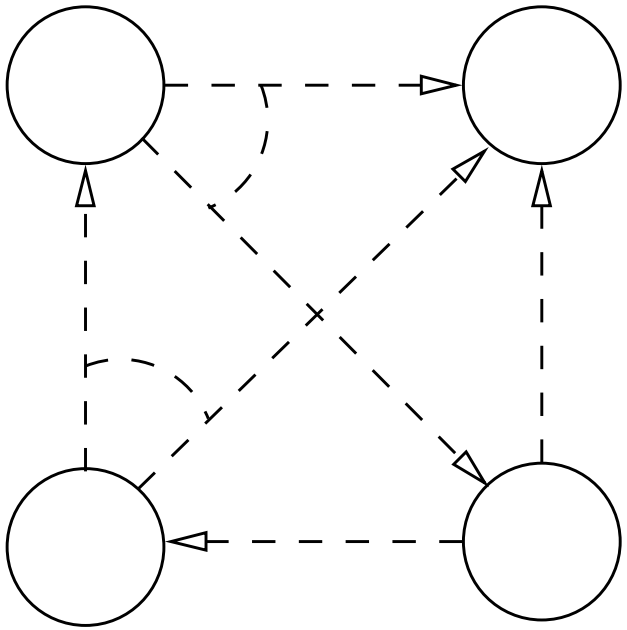
Static analysis - Example 2



Static analysis - Example 2



Static analysis - Example 2



No deadlock

Bracha-Toueg deadlock detection algorithm - Snapshot

(doing the static analysis of wait-for graph in a distributed manner)

Given an undirected network, and a basic algorithm.

A process that suspects it is deadlocked, initiates
a (Lai-Yang) *snapshot* to compute the wait-for graph.

Each node u takes a local snapshot of:

- requests it sent or received that weren't yet granted or dismissed;
- grant and dismiss messages in edges.

Then it computes:

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

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

Bracha-Toueg deadlock detection algorithm - Snapshot

(doing the static analysis of wait-for graph in a distributed manner)

Given an undirected network, and a basic algorithm.

A process that suspects it is deadlocked, initiates
a (Lai-Yang) *snapshot* to compute the wait-for graph.

Each node u takes a local snapshot of:

- requests it sent or received that weren't yet granted or dismissed;
- grant and dismiss messages in edges.

Then it computes:

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

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

Bracha-Toueg deadlock detection algorithm - Snapshot

(doing the static analysis of wait-for graph in a distributed manner)

Given an undirected network, and a basic algorithm.

A process that suspects it is deadlocked, initiates
a (Lai-Yang) *snapshot* to compute the wait-for graph.

Each node u takes a local snapshot of:

- requests it sent or received that weren't yet granted or dismissed;
- grant and dismiss messages in edges.

Then it computes:

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

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

Bracha-Toueg deadlock detection algorithm

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

When u receives a grant message, $requests_u \leftarrow requests_u - 1$.

If $requests_u$ becomes 0, u sends grant messages to all nodes in In_u.

If after termination of the deadlock detection run, $requests > 0$ at the initiator, then it is deadlocked (in the basic algorithm).

Bracha-Toueg deadlock detection algorithm

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

When u receives a grant message, $requests_u \leftarrow requests_u - 1$.

If $requests_u$ becomes 0, u sends grant messages to all nodes in In_u.

If after termination of the deadlock detection run, $requests > 0$ at the initiator, then it is deadlocked (in the basic algorithm).

Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

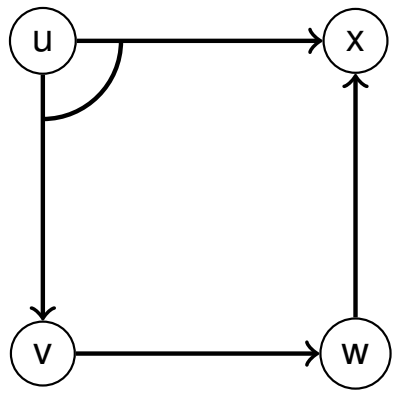
---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 2$
$requests_v = 1$
$requests_w = 1$
$requests_x = 0$



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

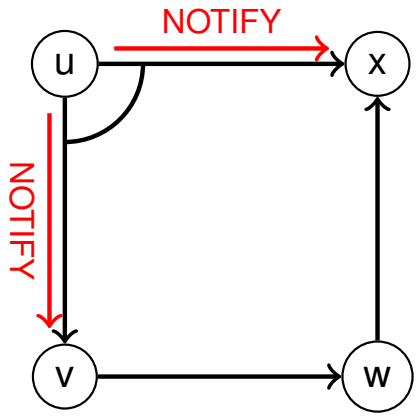
---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 2$
$requests_v = 1$
$requests_w = 1$
$requests_x = 0$



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

---

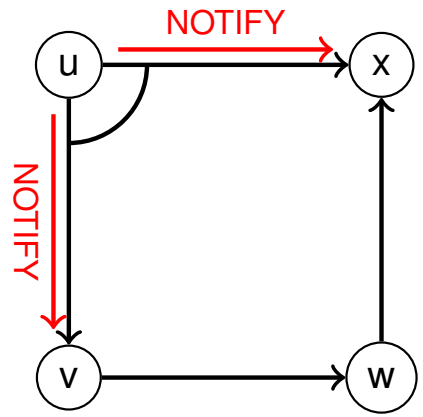
On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 2$
$requests_v = 1$
$requests_w = 1$
$requests_x = 0$

await DONE from v, x



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

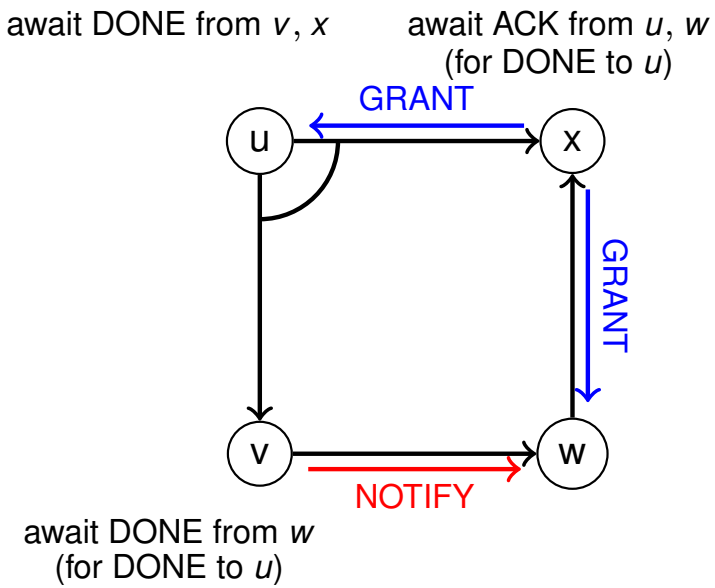
---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 2$
$requests_v = 1$
$requests_w = 1$
$requests_x = 0$



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

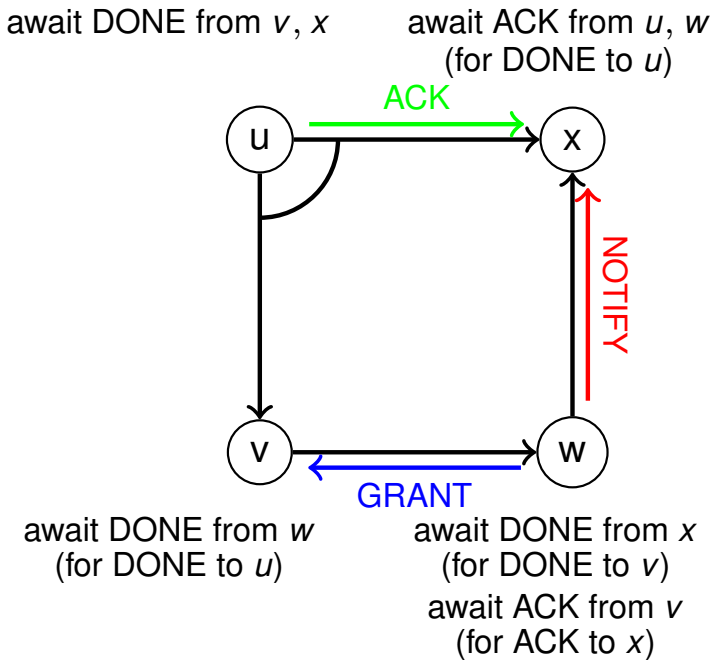
---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 1$
$requests_v = 1$
$requests_w = 0$
$requests_x = 0$



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

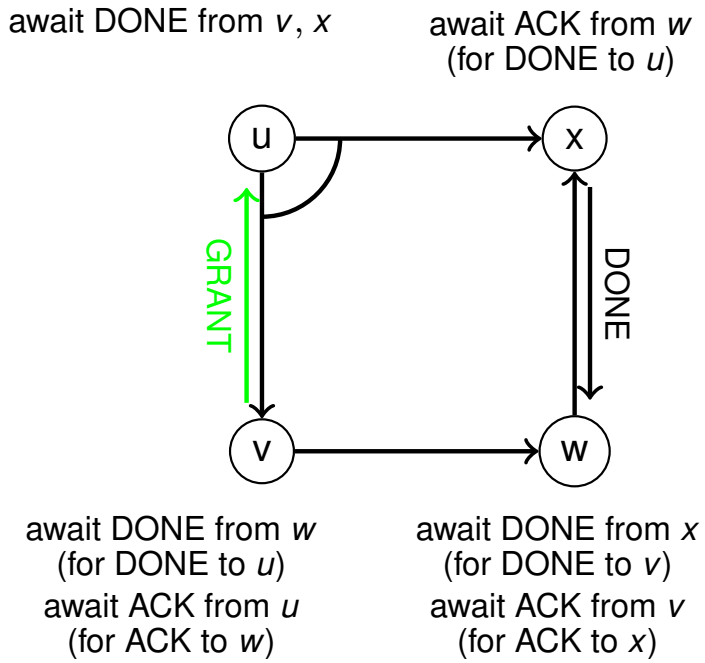
---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 1$
$requests_v = 0$
$requests_w = 0$
$requests_x = 0$



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

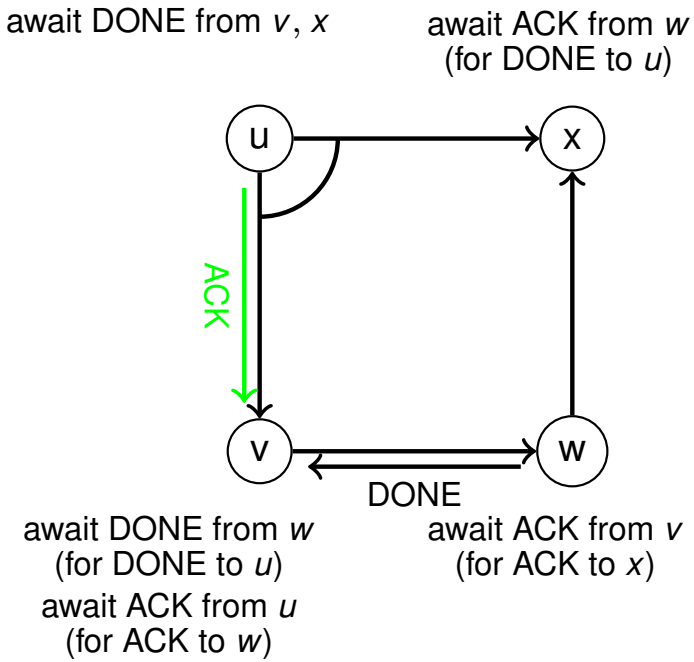
---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 0$
$requests_v = 0$
$requests_w = 0$
$requests_x = 0$



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

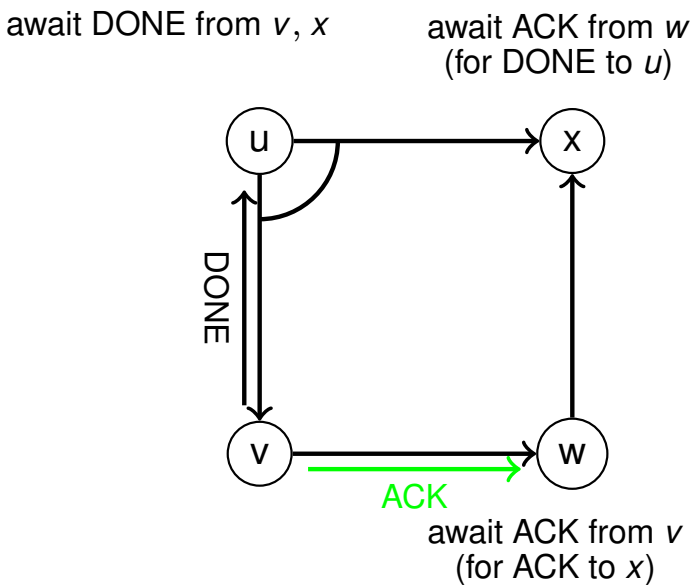
---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 0$
$requests_v = 0$
$requests_w = 0$
$requests_x = 0$



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

---

On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

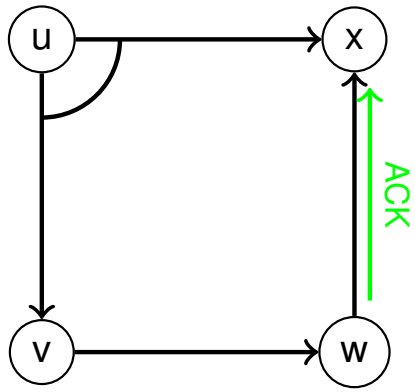
On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 0$
$requests_v = 0$
$requests_w = 0$
$requests_x = 0$

await DONE from x

await ACK from w
(for DONE to u)



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

Initially, for all u, $notified_u = free_u = false$. The initiator executes *Notify*.

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

---

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send $\langle \text{GRANT} \rangle$ to w
for all $w \in In_u$ await $\langle \text{ACK} \rangle$ from w

---

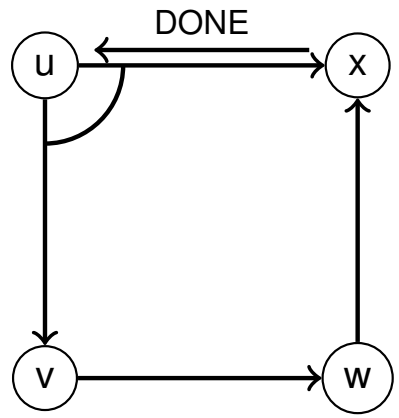
On receive $\langle \text{NOTIFY} \rangle$:
If $notified_u = false$, then $Notify_u$.
u sends back $\langle \text{DONE} \rangle$.

On receive $\langle \text{GRANT} \rangle$:
If $requests_u > 0$, then
$requests_u \leftarrow requests_u - 1$
if $requests_u = 0$, then $Grant_u$.
u sends back $\langle \text{ACK} \rangle$.

A node awaiting $\langle \text{DONE} \rangle$ or $\langle \text{ACK} \rangle$ can process incoming $\langle \text{NOTIFY} \rangle$ and $\langle \text{GRANT} \rangle$ messages.

$requests_u = 0$
$requests_v = 0$
$requests_w = 0$
$requests_x = 0$

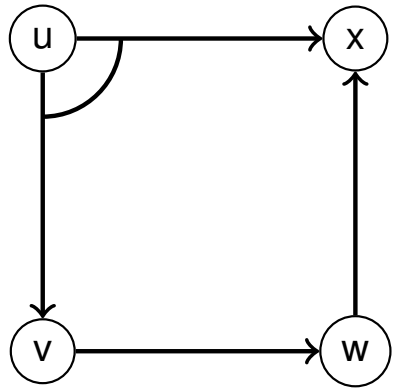
await DONE from x



Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

$requests_u = 0$
$requests_v = 0$
$requests_w = 0$
$requests_x = 0$

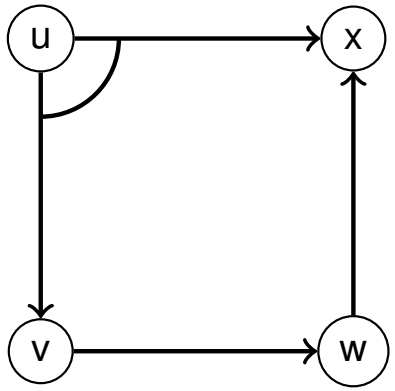
When the initiator receives $\langle \text{DONE} \rangle$ from all nodes in its *Out* set, it checks the value of its *free* field. If it is still *false*, the initiator concludes it is deadlocked.



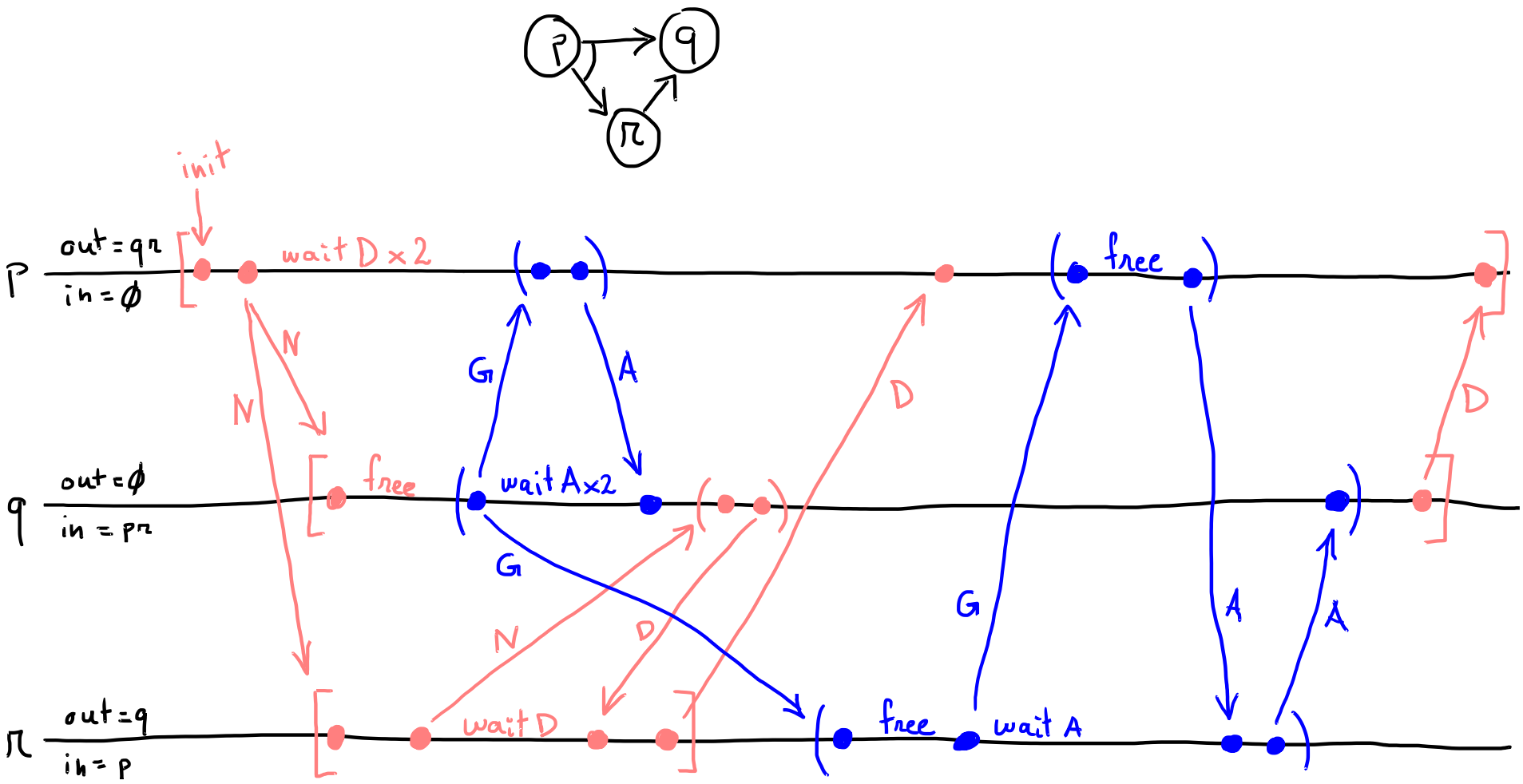
Bracha-Toueg Deadlock Detection Algorithm (echo-like wave algorithm)

When the initiator receives $\langle \text{DONE} \rangle$ 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.

$$requests_u = 0$$
$$requests_v = 0$$
$$requests_w = 0$$
$$requests_x = 0$$



$free_u = true$, so u concludes that it isn't deadlocked.


p
q
π
init
out = qπ
in = ϕ
wait D x 2
N
N
G
A
D
free
out = ϕ
in = pπ
free
wait A x 2
G
N
D
G
A
A
D
out = q
in = p
wait D
free
wait A

Bracha-Toueg deadlock detection algorithm - Correctness

The Bracha-Toueg algorithm is deadlock-free:

The initiator eventually receives DONE's from all nodes in its *Out* set.

At that moment the Bracha-Toueg algorithm has terminated.

Two types of trees are constructed, similar to the echo algorithm:

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 $requests_v = 0$.

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

Bracha-Toueg deadlock detection algorithm - Correctness

The Bracha-Toueg algorithm is deadlock-free:

The initiator eventually receives DONE's from all nodes in its *Out* set.

At that moment the Bracha-Toueg algorithm has terminated.

Two types of trees are constructed, similar to the echo algorithm:

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 $requests_v = 0$.

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

Bracha-Toueg deadlock detection algorithm - Correctness

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

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.

Vice versa, if its *free* field is *true*, there is no deadlock (yet), *(if resource requests are granted nondeterministically)*.

Bracha-Toueg deadlock detection algorithm - Correctness

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

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.

Vice versa, if its *free* field is *true*, there is no deadlock (yet), *(if resource requests are granted nondeterministically)*.

Question

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

Question

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

Answer: No.

The Bracha-Toueg algorithm can only establish whether a deadlock is present in a snapshot of one computation of the basic algorithm.