Distributed computing
“Coordination and agreement”

Mathieu Delalandre
University of Tours, Tours city, France
mathieu.delalandre@univ-tours.fr

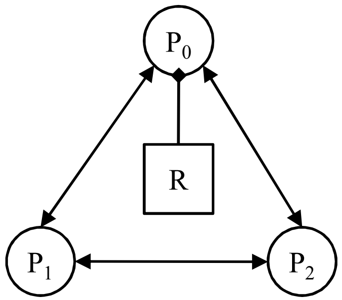
Lecture available at http://mathieu.delalandre.free.fr/teachings/dcomputing.html

Coordination and agreement

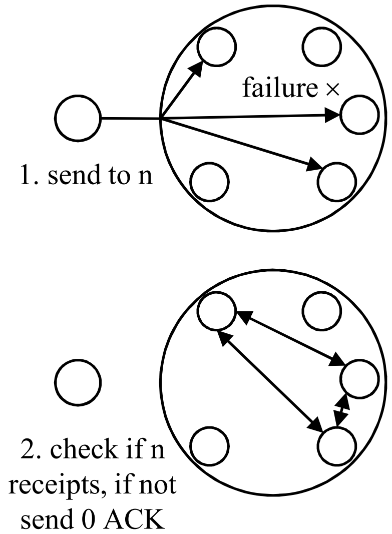
Introduction

Coordination and agreement concerns distributed algorithms whose goals aim to coordinate distributed processes on actions to perform, but also agreement on values to reach. Some typical problems are given here,

Distributed mutual exclusion: in a group, processes must agree on the access requirements to no-shareable resources.



Group communication: specific algorithms need to be designed to enable efficient group communication.



At the corner. leader election, consensus agreement, deadlock detection and resolution, global predicate detection, termination detection, failure detection, garbage collection, etc.

Coordination and agreement

1. Introduction
2. Distributed mutual exclusion
    2.1. Introduction
    2.2. The baseline methods
    2.3. The Ricart-Agrawala algorithm
    2.4. The performance metrics
3. Group communication
    3.1. Introduction
    3.2. Basic multicast
    3.3. Reliable multicast over B-multicast
    3.4. Reliable multicast over IP-multicast

Distributed mutual exclusion
"Introduction" (1)

Mutual exclusion: two events are mutually exclusive if they cannot occur at the same time. Mutual exclusion algorithms are used to avoid the simultaneous use of a resource by the critical section piece of code.

Requirements for mutual exclusion include the following properties.

Safety: at most one process may execute in the critical section.

Liveness: requests to enter and to exit the critical section eventually succeed.

Ordering: if one request to enter in the critical section happened-before another, the entry is granted in that order.

Distributed mutual exclusion
“Introduction” (2)

Mutual exclusion: two events are mutually exclusive if they cannot occur at the same time. Mutual exclusion algorithms are used to avoid the simultaneous use of a resource by the critical section piece of code.

The baseline ways to achieve mutual exclusion is message passing, to use a server or to arrange processes in a logical ring. These approaches suffers from performance and failure and are not supporting ordering. Different advanced algorithms have been proposed to solve these problems including the Ricart-Agrawala algorithm.

	Process number	**Requirements**			**Constraints and performances**
		Safety	**Liveness**	**Ordering**	
Message passing	2	yes			it works with 2 processes
Central server algorithm	≥2	yes		no	the coordinator can crash
Ring-based algorithm					synchronization delay large
Ricart-Agrawala algorithm		yes			linear complexity

Coordination and agreement

1. Introduction
2. Distributed mutual exclusion
    2.1. Introduction
    2.2. The baseline methods
    2.3. The Ricart-Agrawala algorithm
    2.4. The performance metrics
3. Group communication
    3.1. Introduction
    3.2. Basic multicast
    3.3. Reliable multicast over B-multicast
    3.4. Reliable multicast over IP-multicast

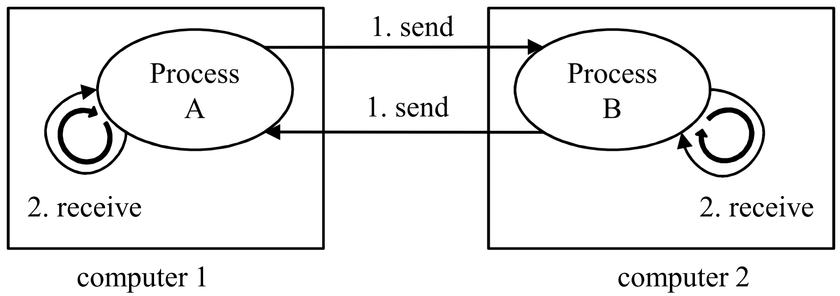
Distributed mutual exclusion
"The baseline methods" (1)

	Process number	Requirements: Safety	Requirements: Liveness	Requirements: Ordering	Constraints and performances
Message passing	2	yes			it works with 2 processes
Central server algorithm	≥2	yes		no	the coordinator can crash
Ring-based algorithm					synchronization delay large
Ricart-Agrawala algorithm		yes			linear complexity

Distributed mutual exclusion
"The baseline methods" (2)

Message passing provides synchronization and communication functions, that can be used in distributed systems as well in shared-memory systems. A number of design issues must be considered: synchronization primitives (blocking, non blocking), addressing, message format and queuing discipline (FIFO, priority, etc.).

The next algorithm achieves mutual exclusion with message passing, we assume here the use of a blocking receive primitive and a non blocking send primitive. The two processes exchange a token that ensures a mutual exclusion access.



command \ message	true	false
receive	continue	wait

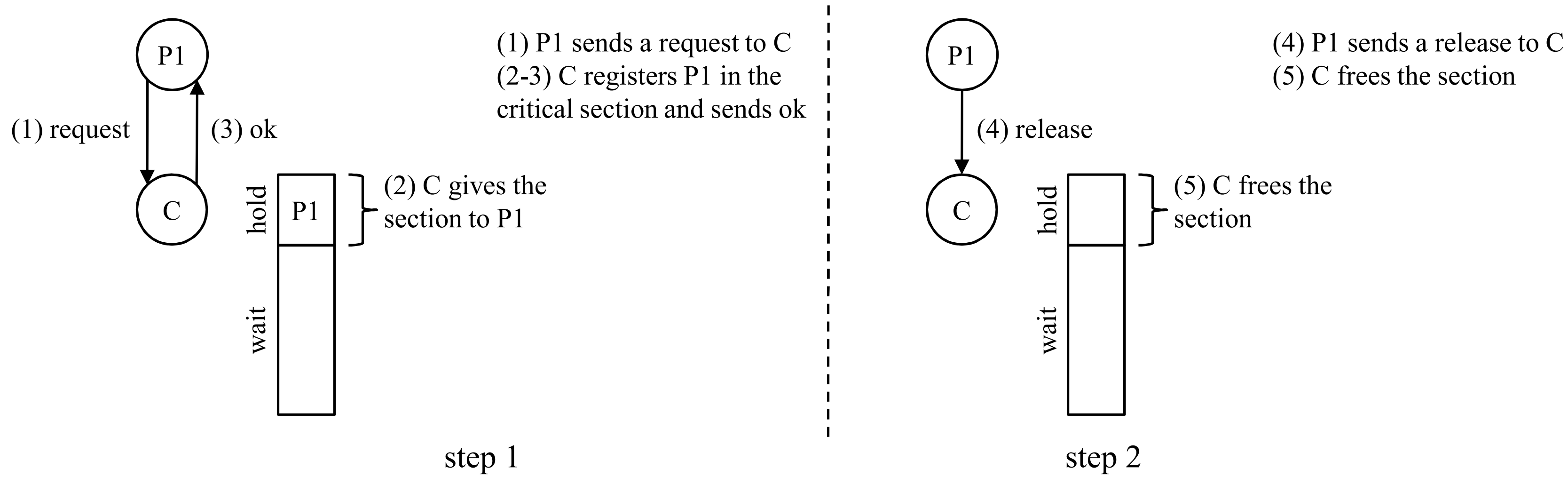
Process **A**
  loop
(1) receive **message** from **B**
(2) pop **item** from **message**
(3) send **message** to **B**

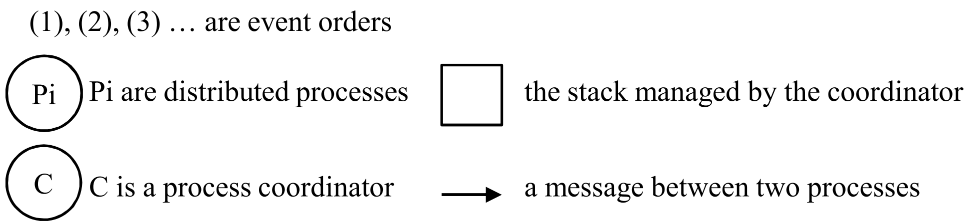
Process **B**
  loop
(1) receive **message** from **A**
(2) push a new **item** in **message**
(3) send **message** to **A**

Distributed mutual exclusion
“The baseline methods” (3)

Central server algorithm: the simplest way to achieve mutual exclusion with multiple processes is to employ a server that grants permission to enter in the critical section. This simulates how mutual exclusion is done in a one-processor system. The two access cases are no-concurrent and concurrent.

The no-concurrent access

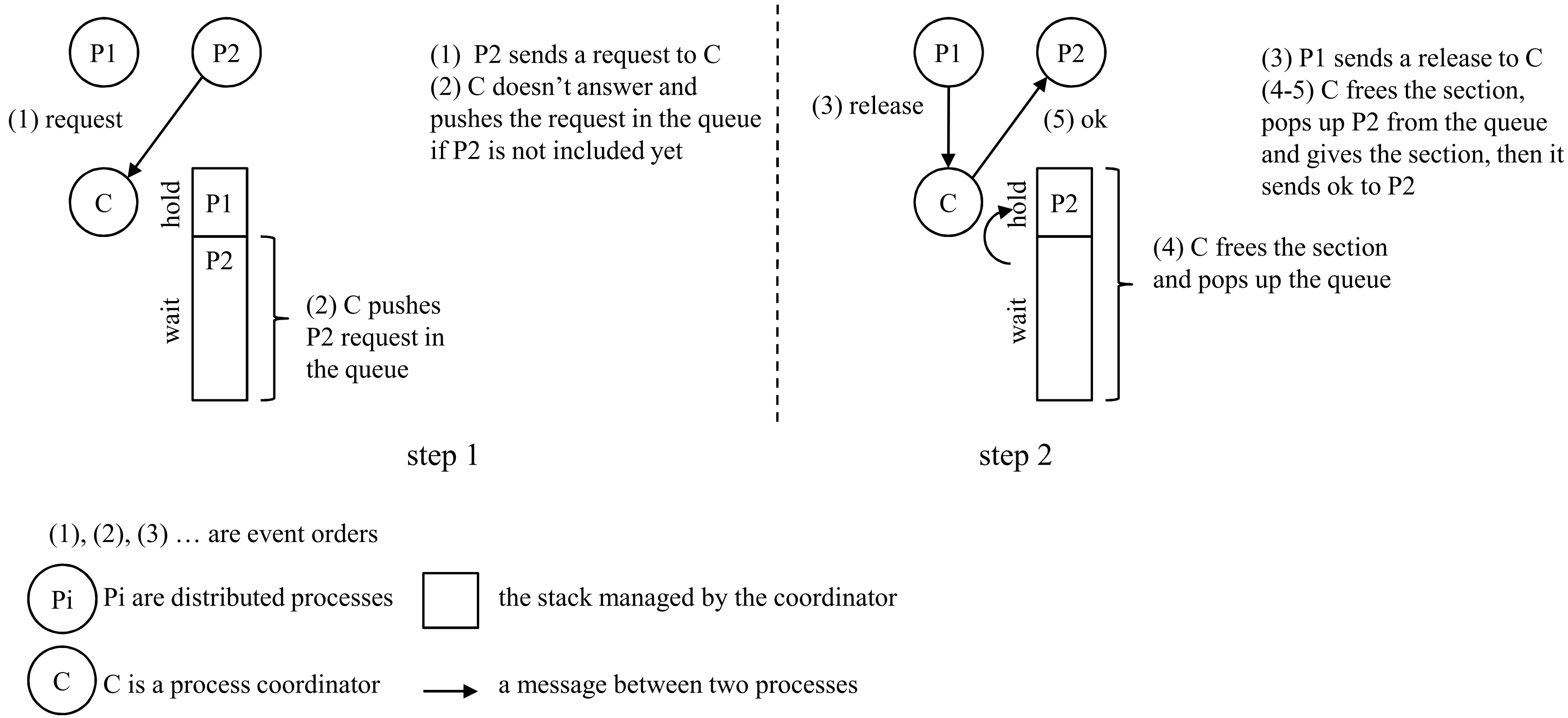




Distributed mutual exclusion
“The baseline methods” (4)

Central server algorithm: the simplest way to achieve mutual exclusion with multiple processes is to employ a server that grants permission to enter in the critical section. This simulates how mutual exclusion is done in a one-processor system. The two access cases are no-concurrent and concurrent.

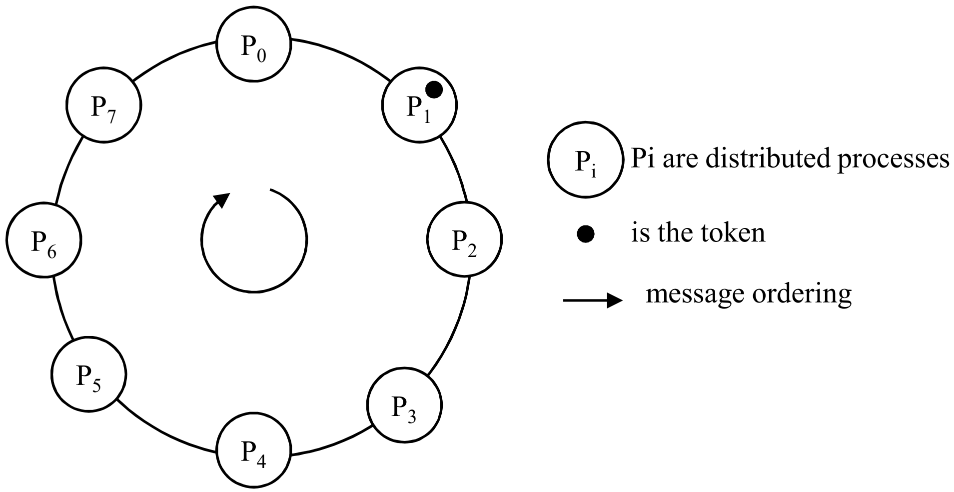
The concurrent access



Distributed mutual exclusion
"The baseline methods" (5)

Ring-based algorithm: requires that each process P_i has a communication channel to the next process in the ring $P_{(i+1) \bmod N}$. The idea is that exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction.

e.g. eight processes P0 to P7 are scheduled on different computers across a network and synchronize through a ring using a token with message passing.



Coordination and agreement

1. Introduction
2. Distributed mutual exclusion
    2.1. Introduction
    2.2. The baseline methods
    2.3. The Ricart-Agrawala algorithm
    2.4. The performance metrics
3. Group communication
    3.1. Introduction
    3.2. Basic multicast
    3.3. Reliable multicast over B-multicast
    3.4. Reliable multicast over IP-multicast

Distributed mutual exclusion
"The Ricart-Agrawala algorithm" (1)

	Process number	Requirements: Safety	Requirements: Liveness	Requirements: Ordering	Constraints and performances
Message passing	2	yes	yes	yes	it works with 2 processes
Central server algorithm	≥2	yes	yes	no	the coordinator can crash
Ring-based algorithm	≥2	yes	yes	no	synchronization delay large
Ricart-Agrawala algorithm	≥2	yes	yes	yes	linear complexity

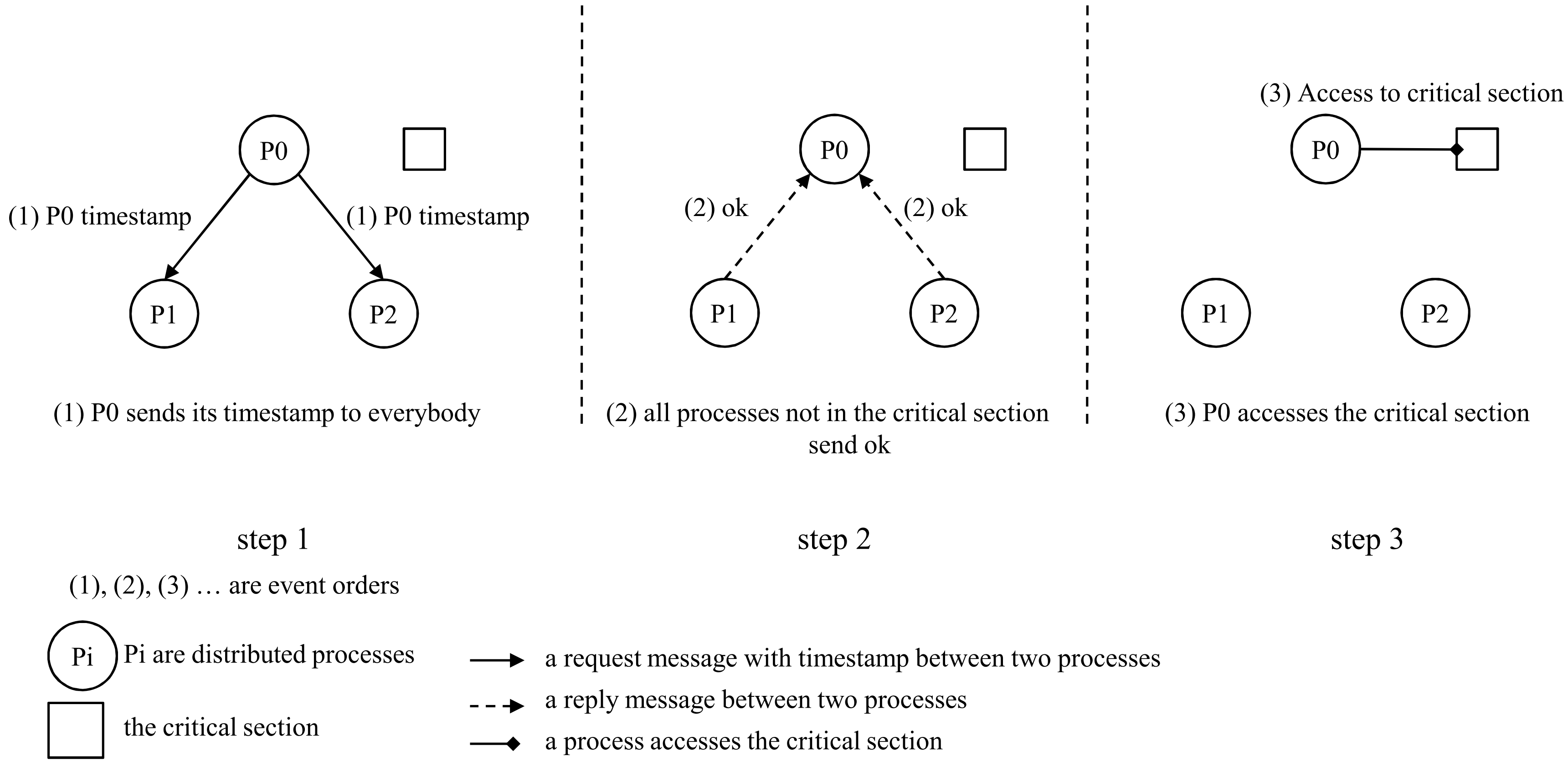
Distributed mutual exclusion
"The Ricart-Agrawala algorithm" (2)

Ricart-Agrawala algorithm: implements mutual exclusion between N peer processes. A process that requires entry to a critical section multicasts a request message, and can enter it only when all other processes have replied to this message. This algorithm requires a total ordering of all events in the system, logical clock can be used to provide timestamps.

Distributed mutual exclusion
"The Ricart-Agrawala algorithm" (3)

Ricart-Agrawala algorithm: implements mutual exclusion between N peer processes. A process that requires entry to a critical section multicasts a request message, and can enter it only when all other processes have replied to this message. This algorithm requires a total ordering of all events in the system, logical clock can be used to provide timestamps.

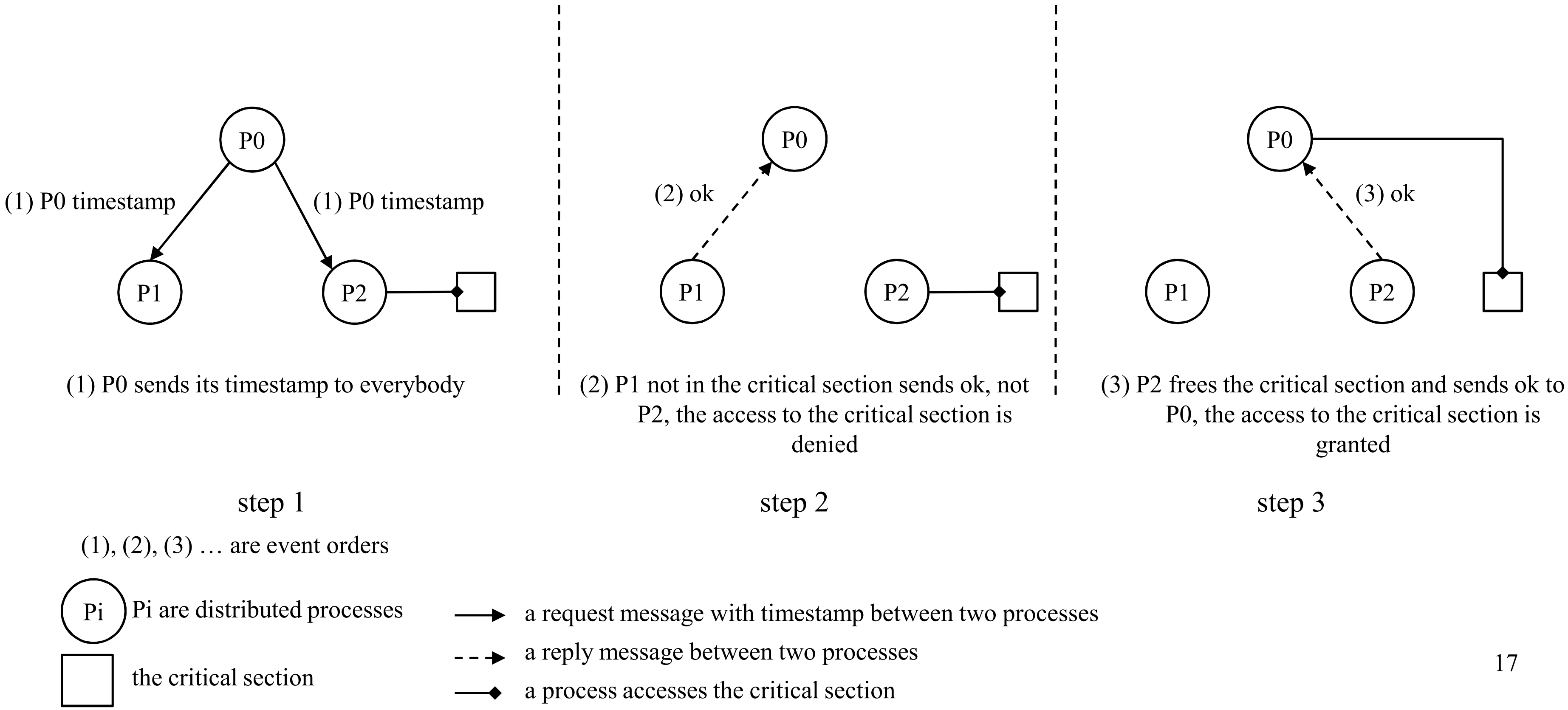
Let us try to understand how the algorithm works, considering **(1) the access case**



Distributed mutual exclusion "The Ricart-Agrawala algorithm" (4)

Ricart-Agrawala algorithm: implements mutual exclusion between N peer processes. A process that requires entry to a critical section multicasts a request message, and can enter it only when all other processes have replied to this message. This algorithm requires a total ordering of all events in the system, logical clock can be used to provide timestamps.

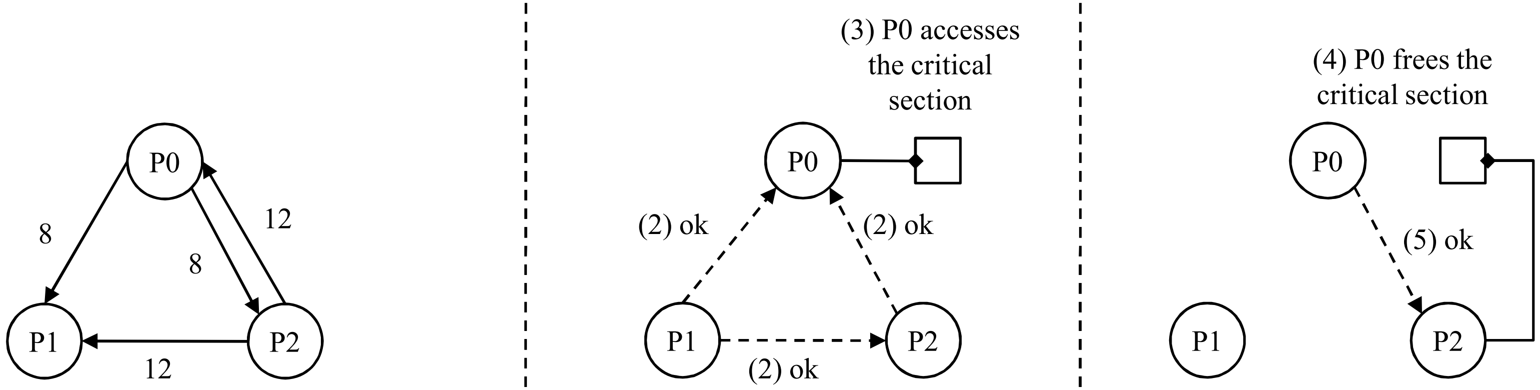
Let us try to understand how the algorithm works, considering **(2) the blocking case**



Distributed mutual exclusion
“The Ricart-Agrawala algorithm” (5)

Ricart-Agrawala algorithm: implements mutual exclusion between N peer processes. A process that requires entry to a critical section multicasts a request message, and can enter it only when all other processes have replied to this message. This algorithm requires a total ordering of all events in the system, logical clock can be used to provide timestamps.

Let us try to understand how the algorithm works, considering **(3) the concurrent case**



(1) P0, P2 are interested to access the critical section, they broadcast their timestamps

(2-3) P1 not in the critical section sends ok to both, P0’s timestamp is lowest thus it wins, P2 sends ok to P0, P0 accesses

(4-5) P0 frees the critical section and sends ok to P2, it grants P0

step 1 | step 2 | step 3

(1), (2), (3) … are event orders

Pi — Pi are distributed processes

☐ — the critical section

→ a request message with timestamp between two processes

- - → a reply message between two processes

—◆ a process accesses the critical section

Distributed mutual exclusion
"The Ricart-Agrawala algorithm" (6)

Ricart-Agrawala algorithm: implements mutual exclusion between N peer processes. A process that requires entry to a critical section multicasts a request message, and can enter it only when all other processes have replied to this message. This algorithm requires a total ordering of all events in the system, logical clock can be used to provide timestamps.

The Ricart-Agrawala algorithm can be defined as follows.

- The algorithm uses request and reply messages, a request message is sent to all other processes to request their permission to enter in the critical section, a reply messages is sent to a process to give a permission.

- Processes use a logical clock to assign timestamps to critical section requests. These timestamps are used to decide the priority of requests in the case of conflict. Considering $<T_i,p_i>$, $<T_j,p_j>$, where T are timestamps and p process identifiers, i and j are the receiver and the sender respectively.

  - If $T_i < T_j$ then p_i defers the reply to p_j while p_i is waiting for the critical section.

  - Otherwise p_i sends a reply message to p_j, thus the highest priority request succeeds in collecting all the reply messages.

Distributed mutual exclusion
“The Ricart-Agrawala algorithm” (7)

Ricart-Agrawala algorithm: implements mutual exclusion between N peer processes. A process that requires entry to a critical section multicasts a request message, and can enter it only when all other processes have replied to this message. This algorithm requires a total ordering of all events in the system, logical clock can be used to provide timestamps.

The Ricart-Agrawala algorithm can be defined as follows.

- Each process maintains a request-deferred array RD_i of size “number of processes in the system”, initially $\forall i\ \forall j\ RD_i[j] = 0$.
  - Whenever p_i defers the request sent by p_j, it sets $RD_i[j] = 1$.
  - After p_i has sent a reply message to p_j, it sets $RD_i[j] = 0$.

- When a process takes up a request for a critical section, it updates its logical clock. Also, when a process receives a timestamp, it updates its clock using this timestamp.

Distributed mutual exclusion
“The Ricart-Agrawala algorithm” (8)

Ricart-Agrawala algorithm: implements mutual exclusion between N peer processes. A process that requires entry to a critical section multicasts a request message, and can enter it only when all other processes have replied to this message. This algorithm requires a total ordering of all events in the system, logical clock can be used to provide timestamps.

Initialization with p_i
(0) state = RELEASED

Requesting the critical section with p_i
(1) state = WANTED, **update** T_i
(2) **broadcast** $<T_i,p_i>$

Receiving rule $<T_j,p_j>$ with p_i
if (state == HELD)
or
(state == WANTED) **and** ($T_i < T_j$)
(3) $RD_i[j] = 1$
otherwise
(4) **update** T_i, **send** a reply

Receiving a reply message with p_i
(5) $Msg_i = Msg_i + 1$
if $Msg_i == N-1$
(6) state = HELD

Releasing the critical section with p_i
(7) state = RELEASED
(8) **for** $\forall j$, **if** $RD_i[j] = 1$
send a reply message to p_j , **set** $RD_i[j] = 0$
$Msg_i = 0$

Ricart-Agrawala algorithm (9)

Initialization with p_i
(0) state = RELEASED

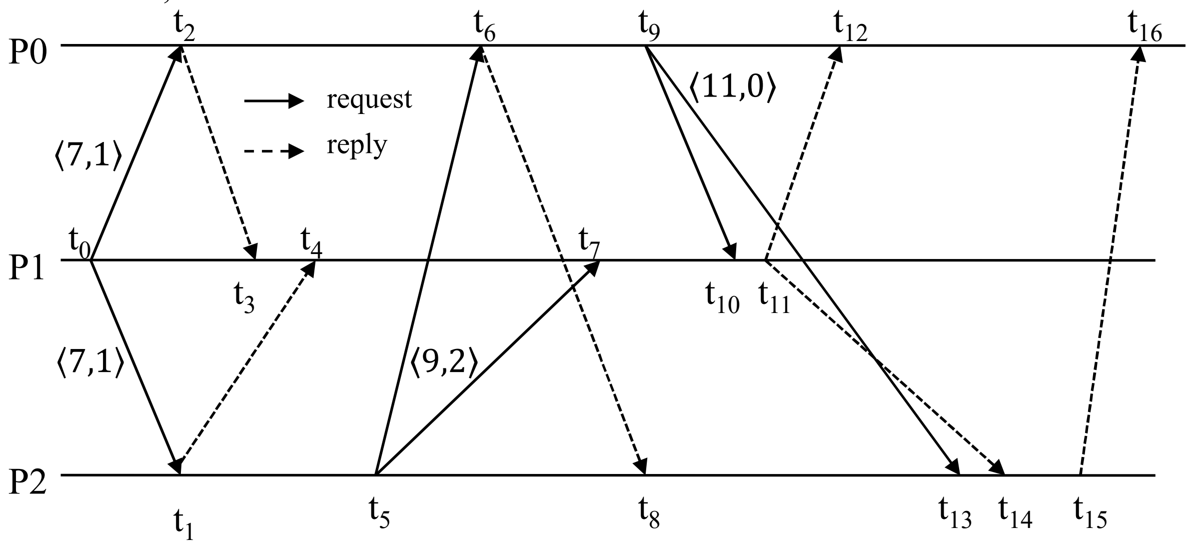
Requesting the critical section with p_i
(1) state = WANTED, **update** T_i
(2) **broadcast** $<T_i,p_i>$

Receiving rule $<T_j,p_j>$ with p_i
if (state == HELD)
or
(state == WANTED) **and** ($T_i < T_j$)
(3) $RD_i[j] = 1$
otherwise
(4) **update** T_i, **send** a reply

Receiving a reply message with p_i
(5) $Msg_i = Msg_i + 1$
if Msg_i == N-1
(6) state = HELD

Releasing the critical section with p_i
(7) state = RELEASED
(8) **for** $\forall j$, **if** $RD_i[j] = 1$
send a reply message to p_j, **set** $RD_i[j] = 0$
$Msg_i = 0$

e.g. considering three process P0, P1 and P2 and a resource R: P1 gets the section at first then P2 is blocked while accessing R, P0 and P2 enter in a concurrent access case, the section is released to P2 then P0.



	Events	Description
(1) access events	t_0 to t_4	P1 asks for the critical section, granted by P0, P1.
(2) blocking access events	t_5 to t_8	P2 asks for the critical section, while P1 holds it.
(3) concurrent access events	t_9, t_{10}, t_{13}	P0 asks for the critical section, while P1 holds it and P2 is still looking for it, as $CL_2 < CL_0$ at t_{13} P2 will not reply.
(2) blocking access events	t_{11}, t_{12}, t_{14}	P1 frees the section, as P2 was granted by P0 at t_8 it got it.
	t_{15} and t_{16}	when releasing the section at t_{15}, P2 grants P0.

Ricart-Agrawala algorithm (10)

Initialization with p_i

(0) state = RELEASED

Requesting the critical section with p_i

(1) state = WANTED, **update** T_i

(2) **broadcast** <T_i,p_i>

Receiving rule <T_j,p_j> with p_i

if (state == HELD)

or

(state == WANTED) **and** ($T_i < T_j$)

(3) $RD_i[j] = 1$

otherwise

(4) **update** T_i, **send** a reply

Receiving a reply message with p_i

(5) $Msg_i = Msg_i + 1$

if Msg_i == N-1

(6) state = HELD

Releasing the critical section with p_i

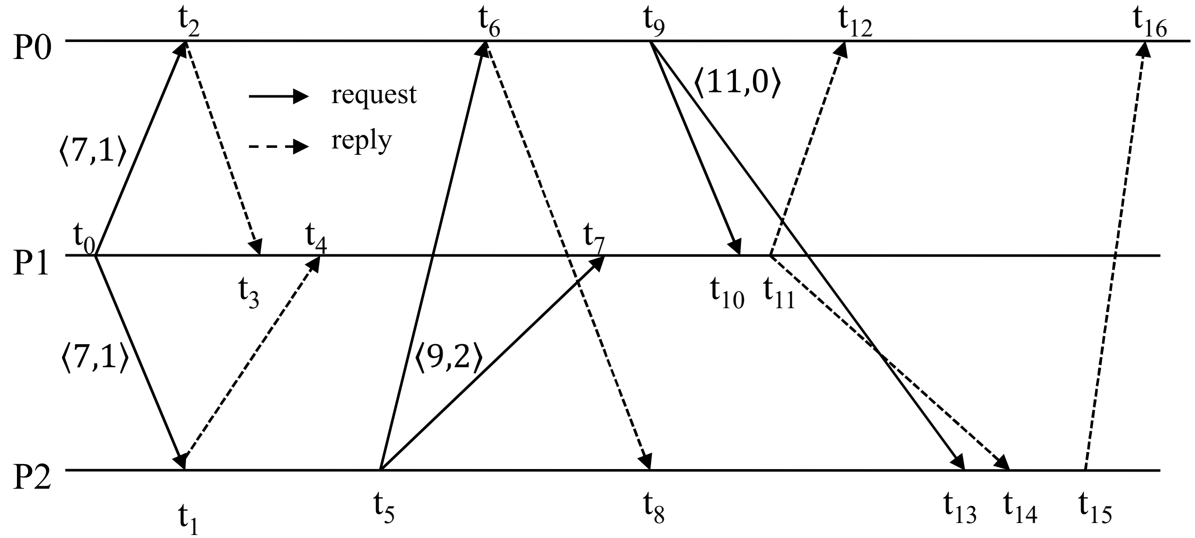
(7) state = RELEASED

(8) **for** $\forall j$, **if** $RD_i[j] = 1$

send a reply message to p_j , **set** $RD_i[j] = 0$

$Msg_i = 0$

e.g. considering three process P0, P1 and P2 and a resource R: P1 gets the section at first then P2 is blocked while accessing R, P0 and P2 enter in a concurrent access case, the section is released to P2 then P0.



time			t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
rules			(1)(2)	(4)	(4)	(5)	(5)(6)	(1)(2)	(4)	(3)	(5)	(1)(2)	(3)	(7)(8)
P0	**state**	R										W		
	CL_0	3			8				10			11		
	RD_0	∅												
	Msg_0	0												
P1	**sate**	R	W				H							R
	CL_1	6	7											
	RD_1	∅								P2			P2,0	∅
	Msg_1	0				1	2							0
P2	**Sate**	R						W						
	CL_2	5		8				9						
	RD_2	∅												
	Msg_2	0									1			

Ricart-Agrawala algorithm (11)

Initialization with p_i
(0) state = RELEASED

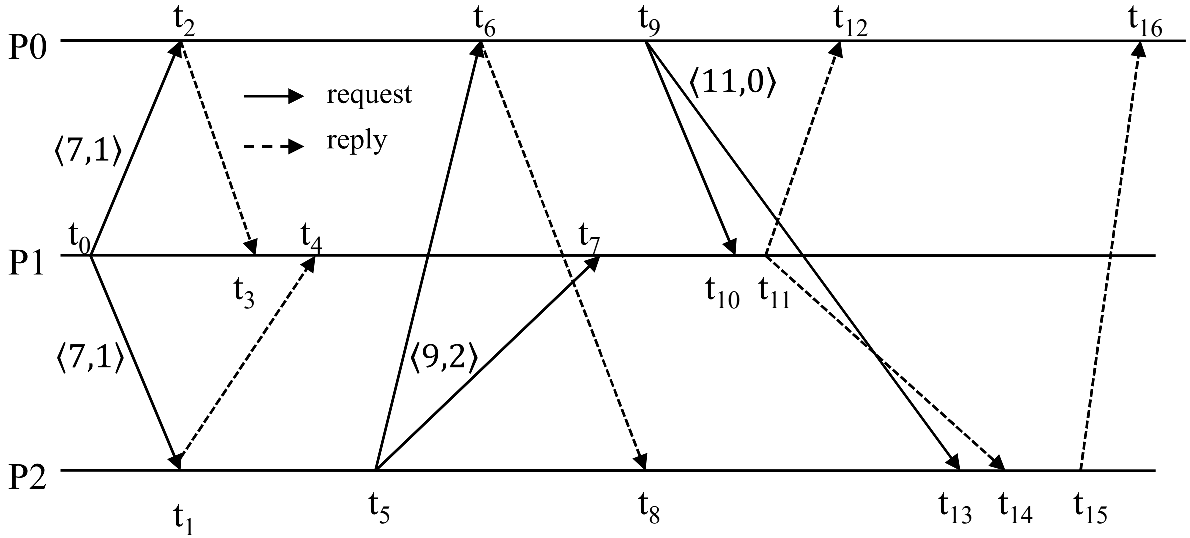
Requesting the critical section with p_i
(1) state = WANTED, **update** T_i
(2) **broadcast** $<T_i,p_i>$

Receiving rule $<T_j,p_j>$ with p_i
if (state == HELD)
or
(state == WANTED) **and** ($T_i < T_j$)
(3) $RD_i[j] = 1$
otherwise
(4) **update** T_i, **send** a reply

Receiving a reply message with p_i
(5) $Msg_i = Msg_i + 1$
if Msg_i == N-1
(6) state = HELD

Releasing the critical section with p_i
(7) state = RELEASED
(8) **for** $\forall j$, **if** $RD_i[j] = 1$
send a reply message to p_j , **set** $RD_i[j] = 0$
$Msg_i = 0$

e.g. considering three process P0, P1 and P2 and a resource R: P1 gets the section at first then P2 is blocked while accessing R, P0 and P2 enter in a concurrent access case, the section is released to P2 then P0.



time			t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
rules			(5)	(3)	(5)(6)	(7)(8)	(5)(6)
P0	**state**	W					H
	CL_0	11					
	RD_0	∅					
	Msg_0	0	1				2
P1	**sate**	R					
	CL_1	7					
	RD_1	∅					
	Msg_1	0					
P2	**Sate**	W			H	R	
	CL_2	9					
	RD_2	∅		P0		∅	
	Msg_2	1			2	0	

Coordination and agreement

1. Introduction
2. Distributed mutual exclusion
    2.1. Introduction
    2.2. The baseline methods
    2.3. The Ricart-Agrawala algorithm
    2.4. The performance metrics
3. Group communication
    3.1. Introduction
    3.2. Basic multicast
    3.3. Reliable multicast over B-multicast
    3.4. Reliable multicast over IP-multicast

Distributed mutual exclusion
“The performance metrics” (1)

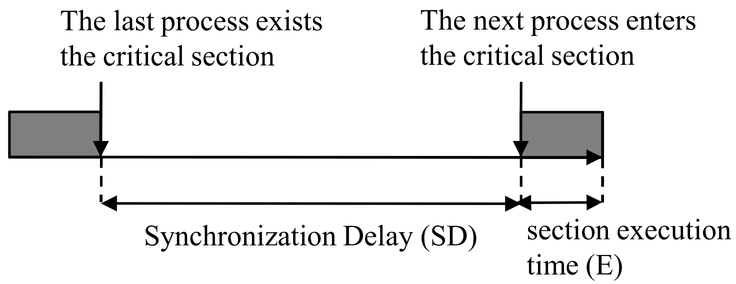
The performance metrics for a mutual exclusion access are generally measured as follows:

Message complexity = f(N) is the number of messages that are required per critical section execution with N the members of the system.

Synchronization delay (SD) is the time required before the next process enters in the critical section.

Section execution time (E) is the execution time that a process requires to be in the critical section.

System Throughput (ST) is the rate at which the system executes requests for the critical section.



$$ST = \frac{1}{\overline{SD} + \overline{E}}$$

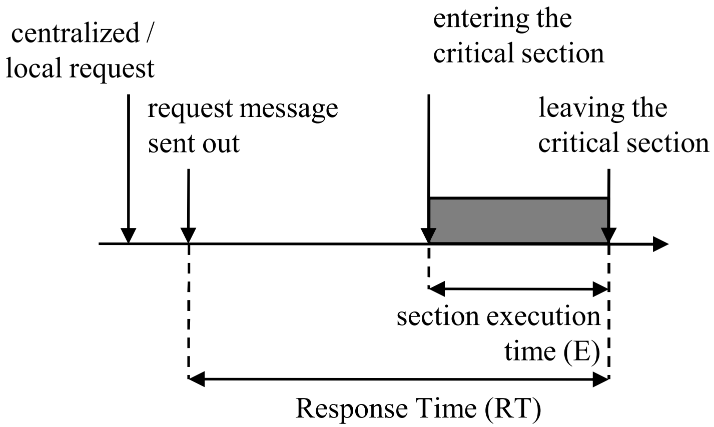
ST	System Throughput in request/second
$\overline{SD} = \frac{1}{n} \sum_{\forall i}^{[1,n]} SD_i$	is the average Synchronization Delay in second (n is the synchronization magnitude)
$\overline{E} = \frac{1}{n} \sum_{\forall i}^{[1,n]} E_i$	is the average section execution time in second (n is the synchronization magnitude)

Distributed mutual exclusion
“The performance metrics” (2)

The performance metrics for a mutual exclusion access are generally measured as follows:

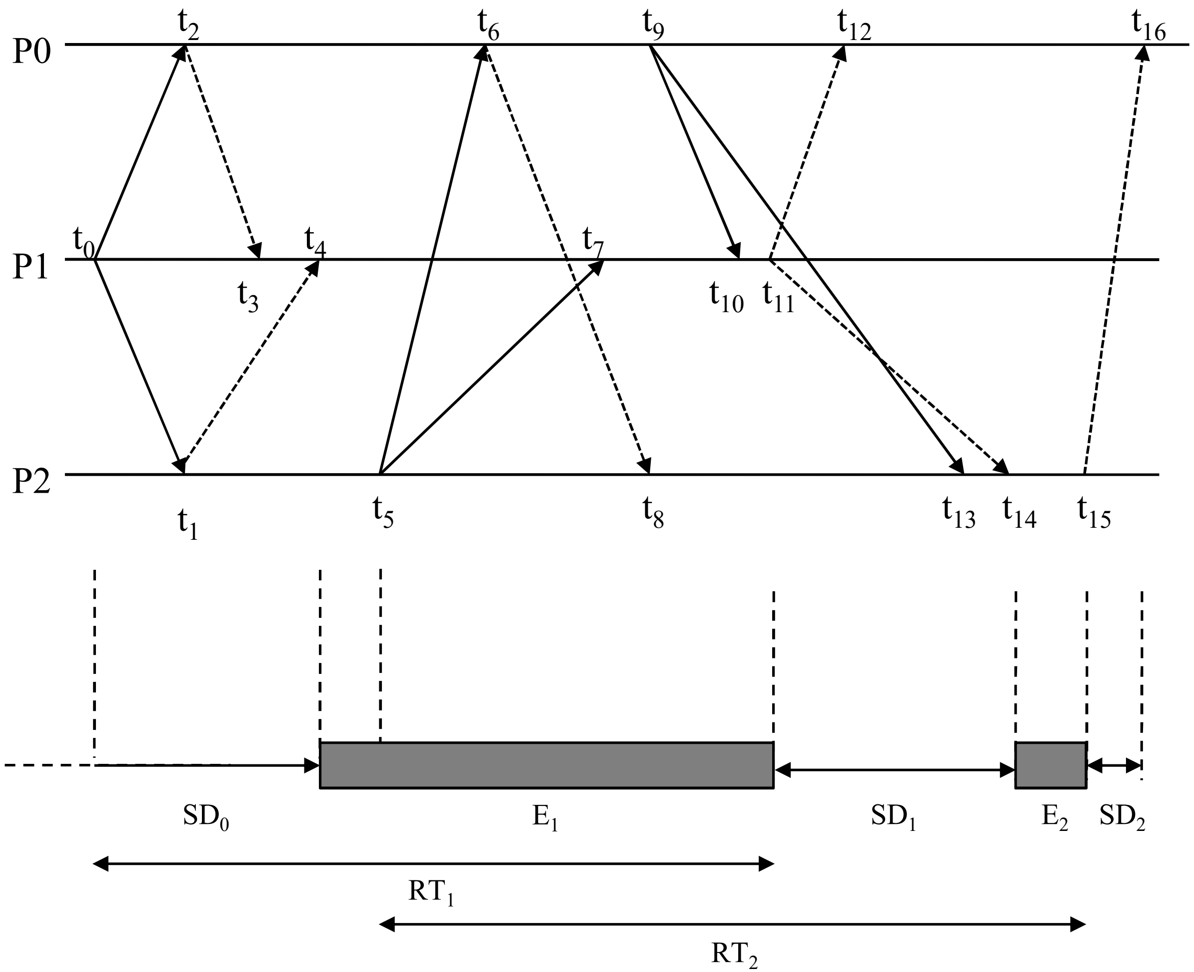
Response time (RT) is the time interval that a request waits its critical section to be over, after its request message has been sent out, we have

$RT > E$



Distributed mutual exclusion
"The performance metrics" (3)

The performance metrics e.g. with the Ricart-Agrawala algorithm



Message complexity is f(N) = 2(N-1)

Time in ms

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
50	63	65	72	78	96	107	126	133

t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
138	151	159	172	185	189	198	202

RT_1	109 ms	RT_2	102 ms

SD_1	30 ms	SD_2	4 ms	$\overline{SD}$	0,017 s
E_1	81 ms	E_2	9 ms	$\bar{E}$	0,045 s
				ST	16,12

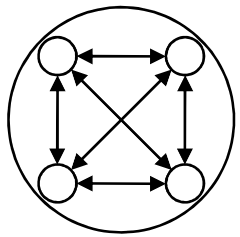
Coordination and agreement

1. Introduction
2. Distributed mutual exclusion
    2.1. Introduction
    2.2. The baseline methods
    2.3. The Ricart-Agrawala algorithm
    2.4. The performance metrics
3. Group communication
    3.1. Introduction
    3.2. Basic multicast
    3.3. Reliable multicast over B-multicast
    3.4. Reliable multicast over IP-multicast

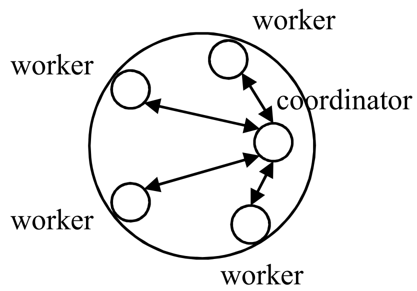
Group communication
"Introduction" (1)

Group communication: specific algorithms need to be designed to enable efficient group communication wherein processes can join and leave groups dynamically, or even fail.

A group is a collection of processes that share a common context and collaborate on a common task within an application domain.



a peer group

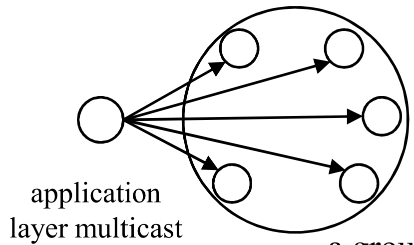


a hierarchical group

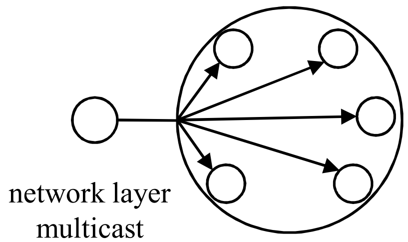
Multicast: the communication within a group can be ensured with a multicast operation driven at the application or network layer.

Application layer multicast is supported with one-to-one send operations. The implementation can use threads to perform the send operations concurrently.

Network layer multicast (or IP multicast) sends message to the group in a single transmission. Copies are automatically created into network elements such as routers, switches, etc. It can be implemented using the IP multicast, that is part of the IP protocol.



a group



a group

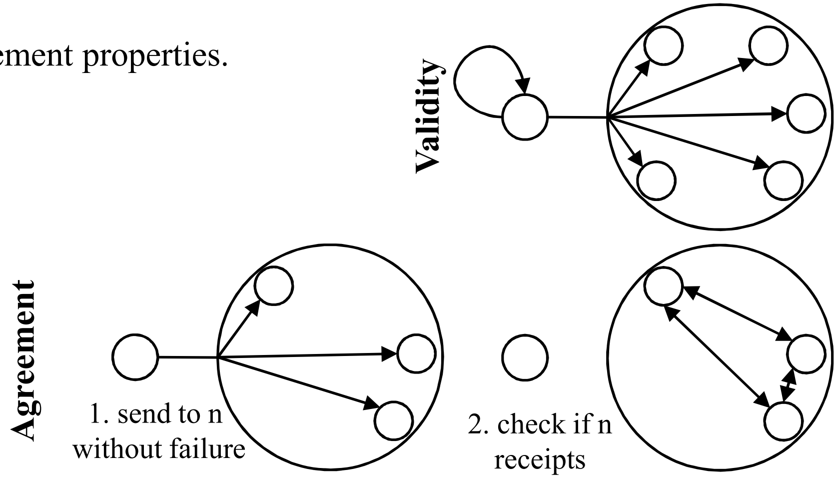
Group communication “Introduction” (2)

Reliable multicast is one that satisfies the integrity, validity and agreement properties.

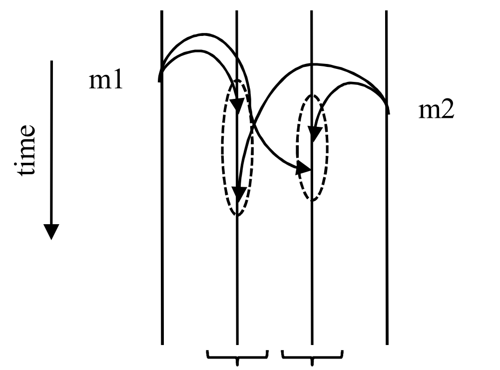
Integrity: the message received is the same as the one sent, and none message is delivered twice.

Validity: if a process multicasts a message m, then it will deliver m. The validity property guarantees liveness for the sender.

Agreement: if a process delivers a message m, then all other processes in the group will deliver m. The message must arrive to all the members of the group.



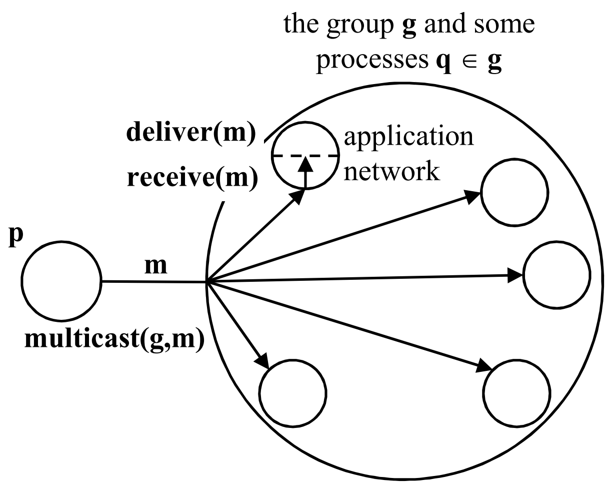
Ordering: different recipients may receive the messages in different orders, possibly violating the semantics of the distributed program. Formal specifications of ordered delivery need to be formulated and implemented.



Group communication
"Introduction" (3)

System model: the system contains a collection of processes, which can communicate reliably over one-to-one channels. The elements to model the system are described below.

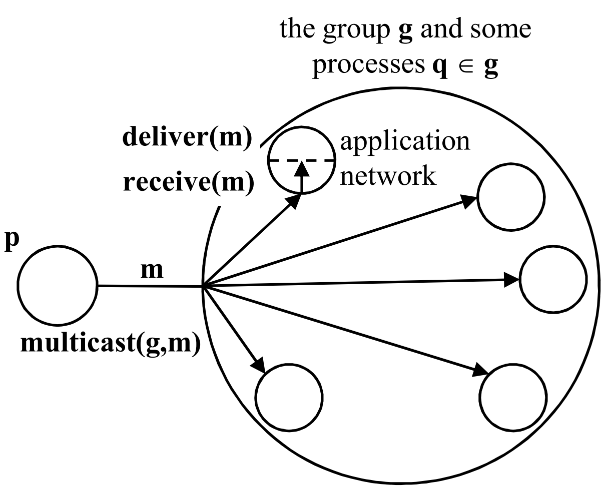
components	**m**	is the message.
	g, |g|	is the group, **|g|** the size of the group.
	p	is the sender process.
	q	is a process of the group such as **q** ∈ **g**.
baseline send operations	**multicast(g,m)**	sends the message **m** to all the processes **q** of the group **g** in a single transmission with IP protocol.
	send(q,m)	is the synchronous send operation, with message sending then ACK receipt.
identifiers	**sender(m)**	every message **m** carries an unique identifier **sender(m)** of the process that sent it.
	group(m)	every message **m** carries an unique identifier **group(m)** of the destination group.
baseline receive operations	**receive(m)**	a member **p** of the group receives a message **m** at the networking interface.
	deliver(m)	a multicast message is not directly delivered at the application layer as soon as it is received.



Group communication
"Introduction" (4)

System model: the system contains a collection of processes, which can communicate reliably over one-to-one channels. The elements to model the system are described below.

basic primitives	**B-multicast(g,m)**	sends the message **m** to all members **q** of the group **g** through one-to-one synchronous send operation.
	B-deliver(m)	the corresponding basic delivery primitive of **B-multicast(m)**.
reliebale primitives	**R-multicast(g,m)**	the reliable primitive to send a message **m** to all the members **q** of the group **g** over a **B-multicast(g,m)** primitive.
	R-deliver(m)	the corresponding delivery primitive of **R-multicast(m)**.



Group communication
“Introduction” (5)

Multicast primitives are implementations for the multicast communication that satisfy the reliability and ordering properties with performances.

	Multicast level	Requirements				Constraints and performances
		Integrity	**Validity**	**Agreement**	**Ordering**	
B-multicast	application layer	yes	no	no	no	ACK-implosion, $2\lvert g\rvert$ messages
R-multicast over B-multicast	application layer	yes	yes	yes	no	ACK-implosion, $\ll (2\lvert g\rvert)^2$ messages
R-multicast over IP multicast	network layer	yes	yes	yes	yes	storage queues to be implemented at each process

Coordination and agreement

Group communication
“Basic multicast” (1)

	Multicast level	Requirements: Integrity	Requirements: Validity	Requirements: Agreement	Requirements: Ordering	Constraints and performances
B-multicast	application layer	yes	no	no	no	ACK-implosion, $2\vert g\vert$ messages
R-multicast over B-multicast	application layer	yes	yes	yes	no	ACK-implosion, << $(2\vert g\vert)^2$ messages
R-multicast over IP multicast	network layer	yes	yes	yes	yes	storage queues to be implemented at each process

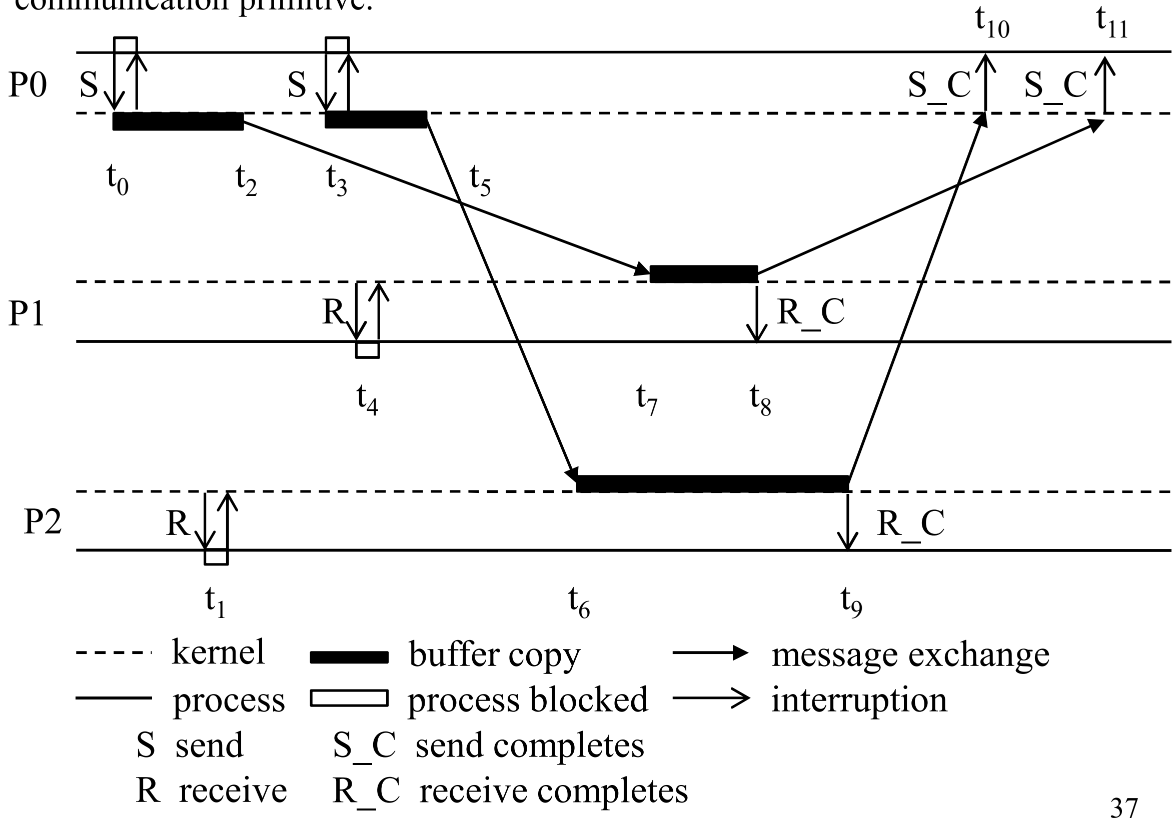
Group communication
"Basic multicast" (2)

Basic multicast B-multicast(g,m) uses the application layer for multicasting with a synchronous send operation. It makes sense to set the communication in a non blocking mode. This multicast primitive guarantees, unlike the IP multicast, that a process will deliver the message as long as the multicaster does not crash.

For a process **p** to **B-multicast(g,m)**:
for each process **q** $\in$ **g**, **send(q,m)**;

On **receive(m)** at **q**: **B-deliver(m)**;

e.g. three processes P0, P1 and P2 where P0 muticasts a message using the B-multicast communication primitive.



Group communication
"Basic multicast" (3)

Basic multicast B-multicast(g,m) uses the application layer for multicasting with a synchronous send operation. It makes sense to set the communication in a non blocking mode. This multicast primitive guarantees, unlike the IP multicast, that a process will deliver the message as long as the multicaster does not crash.

For a process **p** to **B-multicast(g,m)**:
for each process **q** $\in$ **g**, **send(q,m)**;

On **receive(m)** at **q**: **B-deliver(m)**;

e.g. three processes P0, P1 and P2 where P0 muticasts a message using the B-multicast communication primitive.

	Events	**Description**
send(q,m)	t_0, t_3	P0 performs concurrent send operations at the application layer.
	t_2, t_5	As the multicast operation is non blocking, sending on the network occurs latter.
	t_{10}, t_{11}	The send operations complete in a synchronous mode when the ACK are received at P0.
receive(m)	t_1 ,t_4	P1, P2 initiate the receive operation.
	t_6 ,t_7	P1, P2 receive the message at the networking interface.
R-deliver(m)	t_8, t_9	The delivering for P1, P2 occurs when the messages are copied to the buffer. The delivery time is different from the receive time.

Group communication
"Basic multicast" (4)

Basic multicast B-multicast(g,m) uses the application layer for multicasting with a synchronous send operation. It makes sense to set the communication in a non blocking mode. This multicast primitive guarantees, unlike the IP multicast, that a process will deliver the message as long as the multicaster does not crash.

For a process **p** to **B-multicast(g,m)**:
for each process **q** $\in$ **g**, **send(q,m)**;

On **receive(m)** at **q**: **B-deliver(m)**;

Communication: the **send(q,m)**, **receive(m)**, **B-deliver(m)** synchronization operations.

Validity: the primitive doesn't satisfy the validity, a process will never deliver the **m** to itself.

Agreement: it cannot guaranty agreement.

Ordering: it cannot guaranty ordering.

Performance: the communication requires 2|**g**| messages.

ACK-implosion: the ACK could arrive from many processes at the same time. The buffer of the process **p** could rapidly fill and must be set consequently.

Coordination and agreement

1. Introduction
2. Distributed mutual exclusion
    2.1. Introduction
    2.2. The baseline methods
    2.3. The Ricart-Agrawala algorithm
    2.4. The performance metrics
3. Group communication
    3.1. Introduction
    3.2. Basic multicast
    3.3. Reliable multicast over B-multicast
    3.4. Reliable multicast over IP-multicast

Group communication
"Reliable multicast over B-multicast" (1)

	Multicast level	Requirements				Constraints and performances
		Integrity	Validity	Agreement	Ordering	
B-multicast	application layer	yes	no			ACK-implosion, $2\lvert g\rvert$ messages
R-multicast over B-multicast		yes			no	ACK-implosion, $\ll (2\lvert g\rvert)^2$ messages
R-multicast over IP multicast	network layer	yes				storage queues to be implemented at each process

Group communication
“Reliable multicast over B-multicast” (2)

R-multicast over B-multicast(g,m) multicasts a message **m** with the **B-multicast(g,m)** primitive, including itself. When the message is delivered the recipient multicasts the message to the group, if it is not the original sender, before delivering. Since a message may arrive more than once, duplicates are deleted using the **sender(m)** identifier.

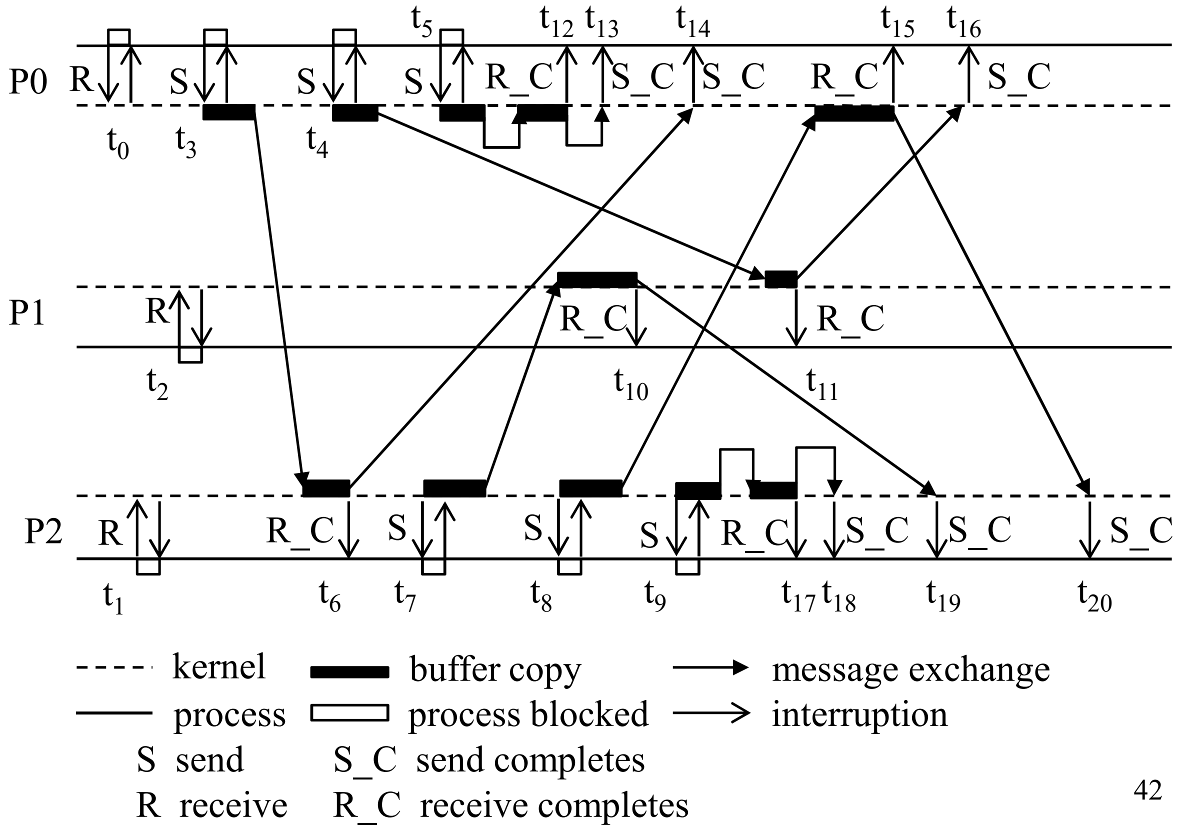
For process **p** to **R-multicast (g,m):**

```
On initialization
Received := {};

B-multicast(g,m);
// p ∈ g is included as a destination

On B-deliver(m) at process q ∈ g
    if (m ∉ Received )
    then
        Received := Received ∪ {m};
        if ( q ≠ p ) then
            B-multicast(g,m);
        end if
        R-deliver(m);
    end if
```

e.g. three processes P0, P1 and P2 where P0 multicasts a message using the R-multicast communication primitive over a B-multicast operation.



Group communication
“Reliable multicast over B-multicast” (3)

R-multicast over B-multicast(g,m) multicasts a message **m** with the **B-multicast(g,m)** primitive, including itself. When the message is delivered the recipient multicasts the message to the group, if it is not the original sender, before delivering. Since a message may arrive more than once, duplicates are deleted using the **sender(m)** identifier.

For process **p** to **R-multicast (g,m):**

On initialization
Received := {};

B-multicast(g,m);
// **p** ∈ **g** is included as a destination

On **B-deliver(m)** at process **q** ∈ **g**
  if (**m** ∉ Received)
  then
    Received := Received ∪ {**m**};
    if (**q** ≠ **p**) then
      B-multicast(g,m);
    end if
    R-deliver(m);
  end if

e.g. three processes P0, P1 and P2 where P0 muticasts a message using the R-multicast communication primitive over a B-multicast operation.

Events	**Description**
t_0, t_1, t_2	P0, P1, P2 initiate their receive operations.
t_3, t_4, t_5	P0 drives a **B-multicast** operation including itself.
t_6	P2 receives the message **m** as the first time, as **q** ≠ **p** it initiates a **B-multicast** operation.
t_7, t_8, t_9	P2 drives a B-multicast operation including itself, at t_9 it will **R-deliver(m)** as the **B-multicast** is over.
t_{10}, t_{11}	P1 receives the P0’s message at t_{10} trough P2, it will initiate a **B-multicast** operation later (>t_{20}). At t_{11}, as **m** ∈ **Received** the message **m** will be ignored.
t_{12}, t_{13}	P0 receives its own message **m** sent to itself, as **q** = **p** the **B-multicast** operation is ignored and P0 **R-deliver(m)**. P0 acknowledges the message to itself at t_{13}.
t_{14}, t_{16}	P0 receives ACK from P2, P1.
t_{15}	P0 receives a message **m** from P2, as **m** ∈ **Received** (see t_{12}) the message **m** will be ignored.

Group communication
"Reliable multicast over B-multicast" (4)

R-multicast over B-multicast(g,m) multicasts a message **m** with the **B-multicast(g,m)** primitive, including itself. When the message is delivered the recipient multicasts the message to the group, if it is not the original sender, before delivering. Since a message may arrive more than once, duplicates are deleted using the **sender(m)** identifier.

For process **p** to **R-multicast (g,m):**

On initialization
Received := {};

B-multicast(g,m);
// **p** $\in$ **g** is included as a destination

On **B-deliver(m)** at process **q** $\in$ **g**
    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

e.g. three processes P0, P1 and P2 where P0 muticasts a message using the R-multicast communication primitive over a B-multicast operation.

Events	**Description**
t_{17}, t_{18}	P2 receives its own message **m** sent to itself, as **m** $\in$ **Received** the message **m** will be ignored.
t_{19}, t_{20}	P2 receives ACK from P1, P0.
$>t_{20}$	P1 must still initiate a B-multicast operation …

Group communication "Reliable multicast over B-multicast" (5)

R-multicast over B-multicast(g,m) multicasts a message **m** with the **B-multicast(g,m)** primitive, including itself. When the message is delivered the recipient multicasts the message to the group, if it is not the original sender, before delivering. Since a message may arrive more than once, duplicates are deleted using the **sender(m)** identifier.

For process **p** to **R-multicast (g,m):**

On initialization
Received := {};

B-multicast(g,m);
// **p** ∈ **g** is included as a destination

On **B-deliver(m)** at process **q** ∈ **g**
  if (**m** ∉ Received)
  then
    Received := Received ∪ {**m**};
    if (**q** ≠ **p**) then
      B-multicast(g,m);
    end if
    R-deliver(m);
  end if

Communication: the **send(q,m)**, **receive(m)**, **B-deliver(m)** synchronization operations.

Validity: the primitive satisfies the validity property, a process will deliver **m** to itself.

Agreement: every process applies **B-multicast(g,m)** then **B-delivered(m)**. **|g|** messages are received per process, that enhances the agreement.

Ordering: this primitive cannot guaranty ordering.

Performance: this primitive is inefficient as each message is sent << $(2|\mathbf{g}|)^2$ times.

ACK-implosion: the ACK implosion is here with $(2|\mathbf{g}|)^2$ messages

Coordination and agreement

1. Introduction
2. Distributed mutual exclusion
    2.1. Introduction
    2.2. The baseline methods
    2.3. The Ricart-Agrawala algorithm
    2.4. The performance metrics
3. Group communication
    3.1. Introduction
    3.2. Basic multicast
    3.3. Reliable multicast over B-multicast
    3.4. Reliable multicast over IP-multicast

Group communication
“Reliable multicast over IP multicast” (1)

	Multicast level	Requirements: Integrity	Requirements: Validity	Requirements: Agreement	Requirements: Ordering	Constraints and performances
B-multicast	application layer	yes	no			ACK-implosion, $2\vert g\vert$ messages
R-multicast over B-multicast		yes			no	ACK-implosion, $\ll (2\vert g\vert)^2$ messages
R-multicast over IP multicast	network layer	yes				storage queues to be implemented at each process

Group communication
“Reliable multicast over IP multicast” (2)

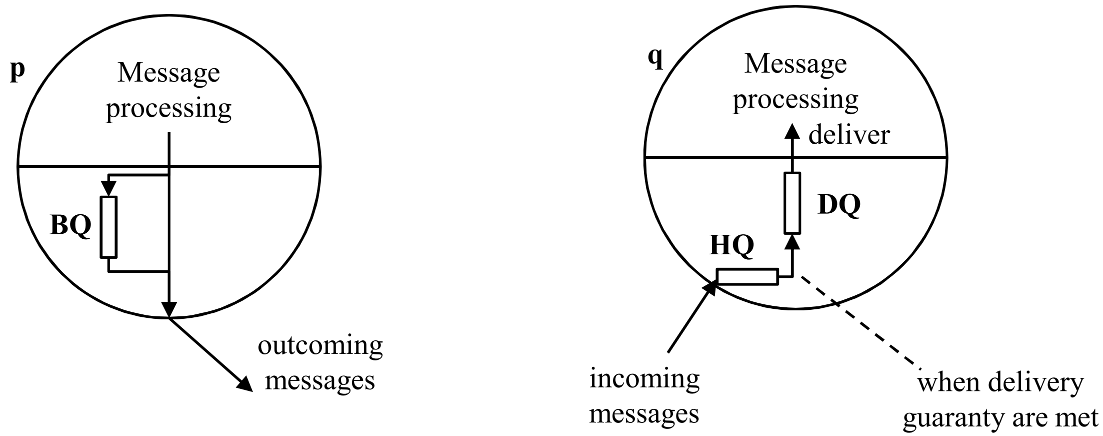
Reliable multicast over IP multicast can be supported with combination of (i) IP multicast, (ii) piggybacked acknowledgements and (iii) negative acknowledgements.

(i) IP multicast: the protocol is based on the observation that IP multicast communication is often successful.

(ii) Piggybacked acknowledgements: processes do not send separate acknowledgement messages; instead, they piggyback acknowledgements on the messages that they send to the group. The piggybacked values in a multicast message enable the recipient to learn about the messages that they have not received yet.

(iii) Negative acknowledgements: processes send a separate response message only when they detect that they have missed a message. A response indicating the absence of an expected message is known as a negative acknowledgement.

(iv) Storage queues: the protocol requires to manage different storage queues (BQ, HQ, DQ) at the process level for the message backup, hold-back and delivery.



Group communication
"Reliable multicast over IP multicast" (3)

Reliable multicast over IP multicast can be supported with combination of (i) IP multicast, (ii) piggybacked acknowledgements and (iii) negative acknowledgements.

We consider a sending processes **p** and a receiving process **q**,
such as **p**, **q** ∈ **g**.

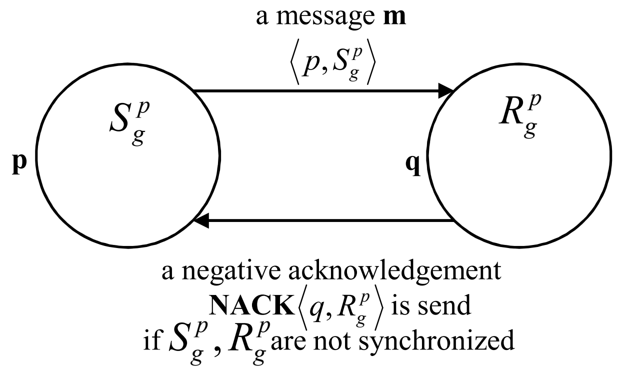
•S_g^p is a sequence number the process **p** for each group **g** to which it belongs.

•R_g^p is a sequence number of the latest message that the process **q** has delivered from **p**.

•**p** exchanges with **q** using piggybacked messages **m** and negative acknowledgements **NACK** based on the synchronization of the S_g^p, R_g^p values.

•The exchange can be extended to |**g**|>2 processes with a vector of k values R_g^k.

•The synchronization of S_g^p, R_g^p values is based on logical clock mechanism bounded to the sender side.



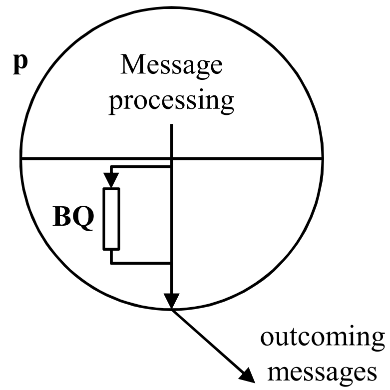
Group communication
"Reliable multicast over IP multicast" (4)

Reliable multicast over IP multicast can be supported with combination of (i) IP multicast, (ii) piggybacked acknowledgements and (iii) negative acknowledgements.

There are different key steps / components in the method,

- **At the initialization of p**: $S_g^p = 0, R_g^k = 0 \ \forall k, HQ = BQ = DQ = \varnothing$

- **To multicast m with p**:
  - we increment $S_g^p = S_g^p + 1$.
  - we piggyback onto the message **m** the value $\langle p, S_g^p \rangle$ and the delivered values $\langle k, R_g^k \rangle \ \forall k$ that **p** received since the last message it sent, this requires a differential vector clock.
  - the piggybacking process pushes messages to be sent in the backup queue **BQ**.

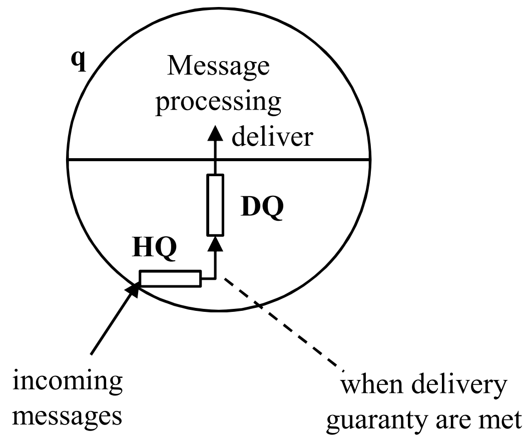


Group communication
"Reliable multicast over IP multicast" (5)

Reliable multicast over IP multicast can be supported with combination of (i) IP multicast, (ii) piggybacked acknowledgements and (iii) negative acknowledgements.

There are different key steps / components in the method,

- **To deliver m with q:**
  - S, R are the sequence number/vector extracted from **m** to be compared with R_g^p, R_g.
  - if $S = R_g^p + 1$ **q** has not received the message yet, **q** delivers **m** and applies $R_g^p = S$.
  - if $S \leq R_g^p$ then **q** has delivered **m** before and discards **m**.
  - if $S > R_g^p + 1 \; or \; R^k > R_g^k \; \forall k$ then there is one or more message that **q** has not received yet, **q** maintains $\langle p, S \rangle$ in the hold-back queue **HQ** and sends **NACK** in the form $\langle q, R_g^{p,k} \rangle$.



Group communication
"Reliable multicast over IP multicast" (6)

Initialization for all **p** (0) $S_g^p = 0, R_g^k = 0 \; \forall k, HQ = BQ = DQ = \varnothing$	to init **p** (either **q**)
For **p** to **R-multicast(g,m)** (1) $S_g^p = S_g^p + 1$ (2) piggyback the messages since the last sending and send **m**	to R-multicast
For **p** to reply to a negative ACK $\langle q, R_g^p \rangle$ (3) recover the messages $\langle p, S_g^p \rangle$ from **BQ** such as $S_g^p > R_g^p$ and send **m**	to reply to a negative ACK
For **q** to **R-deliver(m)** from **p** $S, R = get(m)$	
(4) case $S = R_g^p + 1$ **R-deliver(m)**, $R_g^p = S$	regular delivering case
(5) case $S \leq R_g^p$ discard **m**	**m** has been delivered before
(6) case $S > R_g^p + 1$ maintain $\langle p, S \rangle$ in **HQ** send a negative ACK $\langle q, R_g^p \rangle$ to **p**	there is one or more message that **q** has not received yet from **p**
(7) case $R > R_g^k \quad \forall k$ maintain $\langle p, S \rangle$ in **HQ** send a negative ACK $\langle q, R_g^k \rangle \; \forall k$	there is one or more message that **q** has not received yet from another process

. . . .

"Reliable over IP" (7)

Initialization for all **p**

(0) $S_g^p = 0, R_g^k = 0 \ \forall k, HQ = BQ = DQ = \varnothing$

For **p** to **R-multicast(g,m)**

(1) $S_g^p = S_g^p + 1$

(2) piggyback the messages since the last sending and send **m**

For **p** to reply to a negative ACK $\langle q, R_g^p \rangle$

(3) recover the messages $\langle p, S_g^p \rangle$ from **BQ** such as $S_g^p > R_g^p$ and send **m**

For **q** to **R-deliver(m)** from **p**

$S, R = get(m)$

(4) case $S = R_g^p + 1$

R-deliver(m), $R_g^p = S$

(5) case $S \leq R_g^p$

discard **m**

(6) case $S > R_g^p + 1$

maintain $\langle p, S \rangle$ in **HQ**

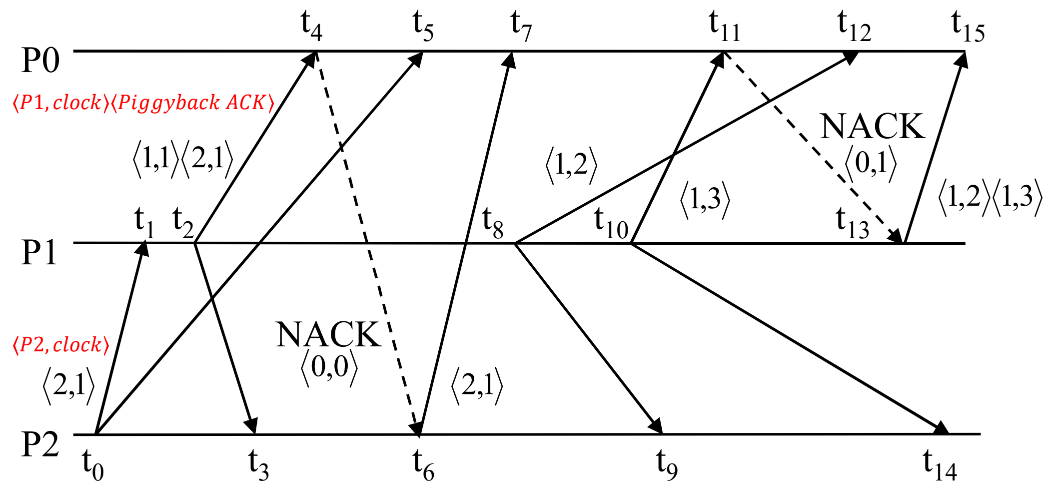
send a negative ACK $\langle q, R_g^p \rangle$ to **p**

(7) case $R > R_g^k \quad \forall k$

maintain $\langle p, S \rangle$ in **HQ**

send a negative ACK $\langle q, R_g^k \rangle \ \forall k$

e.g. considering three process P0, P1 and P2 cooperating in a distributed way with reliable multicast over IP multicast.



→ a message **m** with piggybacked values - - → a negative ACK

Time	**Rule**
t_0	(1)(2)
t_1	(4)
t_2	(1)(2)
t_3	(4)
t_4	(7)
t_5	(4)
	(4)
t_6	(3)
t_7	(5)

t_0–t_3: regular sending receiving case, as P2 has not received a message yet there is no piggybacked message. P1 receives a P2's message and piggybacks it.

t_4: P0 has not received the P2's message yet, it pushes the P1's message in its holdback queue and sends a negative ACK to P2.

t_5: P0 processes the P2 message, then this triggers a delivery condition for the P1's message recovered from the holdback queue and P2 processes that message.

t_6: P2 resends a message to P1, recovered from its backup queue, to reply to the NACK.

t_7: P0 already received the P2's message while P2 replied to the negative ACK, the message is discarded.

. . . .

"Reliable over IP" (8)

Initialization for all **p**

(0) $S_g^p = 0, R_g^k = 0\ \forall k, HQ = BQ = DQ = \varnothing$

For **p** to **R-multicast(g,m)**

(1) $S_g^p = S_g^p + 1$

(2) piggyback the messages since the last sending and send **m**

For **p** to reply to a negative ACK $\langle q, R_g^p \rangle$

(3) recover the messages $\langle p, S_g^p \rangle$ from **BQ** such as $S_g^p > R_g^p$ and send **m**

For **q** to **R-deliver(m)** from **p**

$S, R = get(m)$

(4) case $S = R_g^p + 1$

R-deliver(m), $R_g^p = S$

(5) case $S \le R_g^p$

discard **m**

(6) case $S > R_g^p + 1$

maintain $\langle p, S \rangle$ in **HQ**

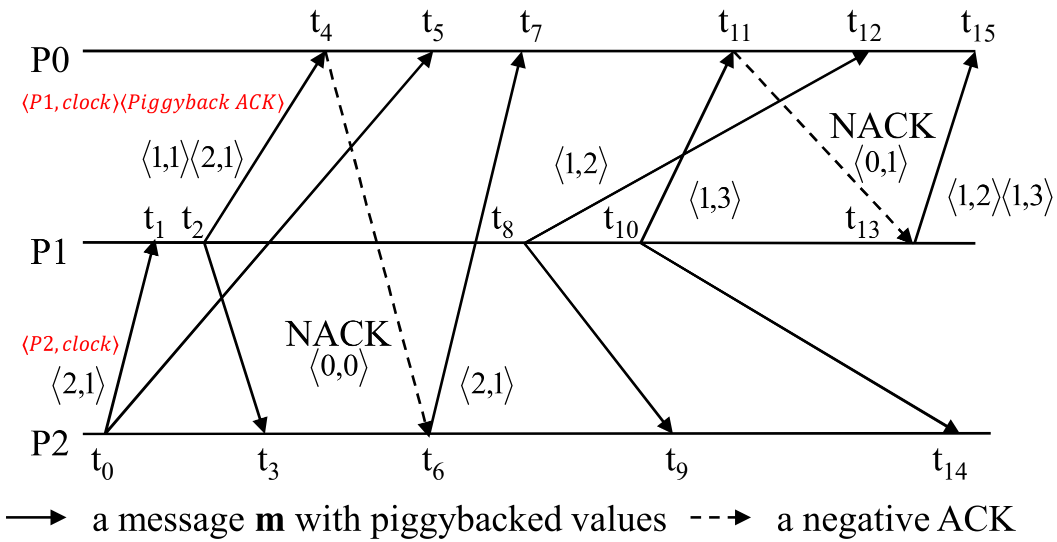
send a negative ACK $\langle q, R_g^p \rangle$ to **p**

(7) case $R > R_g^k \quad \forall k$

maintain $\langle p, S \rangle$ in **HQ**

send a negative ACK $\langle q, R_g^k \rangle\ \forall k$

e.g. considering three process P0, P1 and P2 cooperating in a distributed way with reliable multicast over IP multicast.



Time	Rule	P0			P1			P2		
		S_g^0	(R_g^0, R_g^1, R_g^2)	**HQ**	S_g^1	(R_g^0, R_g^1, R_g^2)	**HQ**	S_g^2	(R_g^0, R_g^1, R_g^2)	**HQ**
		0	(×, 0, 0)	∅	0	(0, ×, 0)	∅	0	(0, 0, ×)	∅
t_0	(1)(2)							1		
t_1	(4)					(0, ×, 1)				
t_2	(1)(2)				1					
t_3	(4)								(0, 1, ×)	
t_4	(7)			⟨1,1⟩						
t_5	(4)		(×, 0, 1)							
	(4)		(×, 1, 1)	∅						
t_6	(3)									
t_7	(5)									

….

"Reliable over IP" (9)

Initialization for all **p**

(0) $S_g^p = 0, R_g^k = 0 \ \forall k, HQ = BQ = DQ = \varnothing$

For **p** to **R-multicast(g,m)**

(1) $S_g^p = S_g^p + 1$

(2) piggyback the messages since the last sending and send **m**

For **p** to reply to a negative ACK $\langle q, R_g^p \rangle$

(3) recover the messages $\langle p, S_g^p \rangle$ from **BQ** such as $S_g^p > R_g^p$ and send **m**

For **q** to **R-deliver(m)** from **p**

$S, R = get(m)$

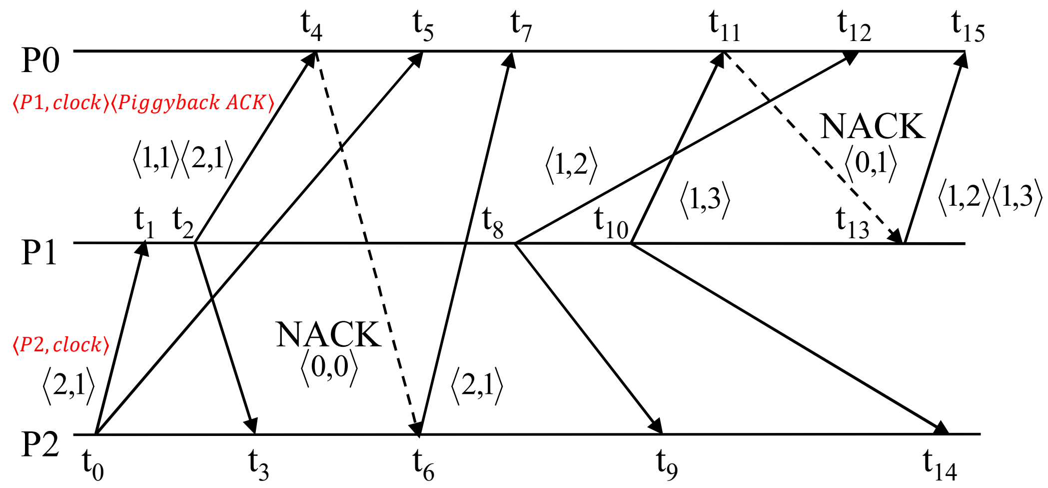
(4) case $S = R_g^p + 1$
R-deliver(m), $R_g^p = S$

(5) case $S \leq R_g^p$
discard **m**

(6) case $S > R_g^p + 1$
maintain $\langle p, S \rangle$ in **HQ**
send a negative ACK $\langle q, R_g^p \rangle$ to **p**

(7) case $R > R_g^k \quad \forall k$
maintain $\langle p, S \rangle$ in **HQ**
send a negative ACK $\langle q, R_g^k \rangle \ \forall k$

e.g. considering three process P0, P1 and P2 cooperating in a distributed way with reliable multicast over IP multicast.



→ a message **m** with piggybacked values - - → a negative ACK

Time	Rule
t_8	(1)(2)
t_9	(4)
t_{10}	(1)(2)
t_{11}	(6)
t_{12}	(4)
	(4)
t_{13}	(3)
t_{14}	(4)
t_{15}	(5)(5)

t_8–t_{10}: regular sending receiving case, as P1 didn't receive a message since t_2, it sent none piggybacked message, same at t_{10}.

t_{11}: P0 receives a message from P1 but it didn't receive the previous one, it pushes this message in its holdback queue and sends a negative ACK to P1.

t_{12}: P0 processes the first P1 message, then this triggers a delivery condition for the previous P1 message recovered from the holdback queue and P0 processes that message.

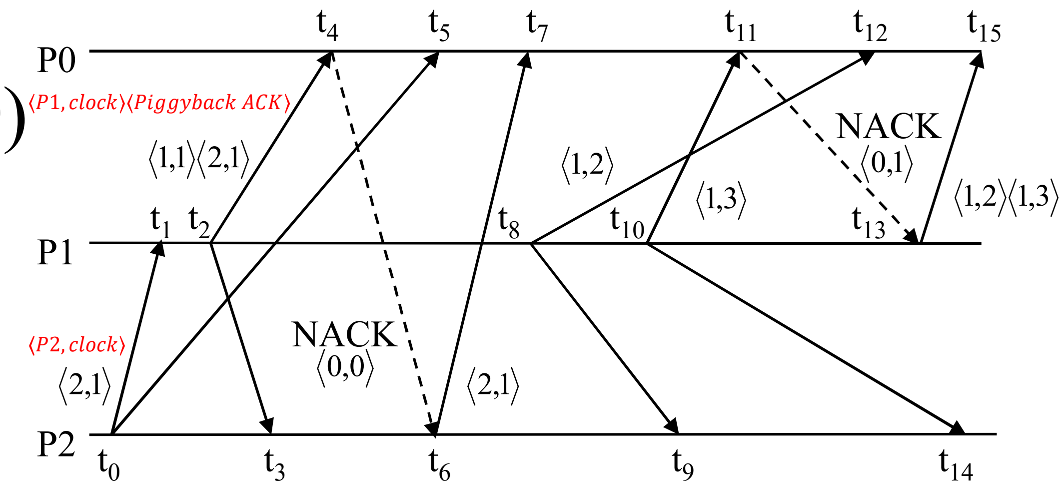
t_{13}: P1 resends a message to P0, recovered from its backup queue, to reply to the NACK.

t_{15}: P0 has already delivered the P1's messages while P1 replied to the negative ACK, the messages are discarded.

....

"Reliable over IP" (10)

e.g. considering three process P0, P1 and P2 cooperating in a distributed way with reliable multicast over IP multicast.



→ a message **m** with piggybacked values - - → a negative ACK

Initialization for all **p**

(0) $S_g^p = 0, R_g^k = 0 \ \forall k, HQ = BQ = DQ = \varnothing$

For **p** to **R-multicast(g,m)**

(1) $S_g^p = S_g^p + 1$

(2) piggyback the messages since the last sending and send **m**

For **p** to reply to a negative ACK $\langle q, R_g^p \rangle$

(3) recover the messages $\langle p, S_g^p \rangle$ from **BQ** such as $S_g^p > R_g^p$ and send **m**

For **q** to **R-deliver(m)** from **p**

$S, R = get(m)$

(4) case $S = R_g^p + 1$

R-deliver(m), $R_g^p = S$

(5) case $S \leq R_g^p$

discard **m**

(6) case $S > R_g^p + 1$

maintain $\langle p, S \rangle$ in **HQ**

send a negative ACK $\langle q, R_g^p \rangle$ to **p**

(7) case $R > R_g^k \quad \forall k$

maintain $\langle p, S \rangle$ in **HQ**

send a negative ACK $\langle q, R_g^k \rangle \ \forall k$

Time	**Rule**	**P0**			**P1**			**P2**		
		S_g^0	(R_g^0, R_g^1, R_g^2)	**HQ**	S_g^1	(R_g^0, R_g^1, R_g^2)	**HQ**	S_g^2	(R_g^0, R_g^1, R_g^2)	**HQ**
		0	(×, 1, 1)	∅	1	(0, ×, 1)	∅	1	(0, 1, ×)	∅
t_8	(1)(2)				2					
t_9	(4)								(0, 2, ×)	
t_{10}	(1)(2)				3					
t_{11}	(6)			⟨1,3⟩						
t_{12}	(4)		(×, 2, 1)							
	(4)		(×, 3, 1)	∅						
t_{13}	(3)									
t_{14}	(4)								(0, 3, ×)	
t_{15}	(5)(5)									