SDR

Jetons

Mutex par jetons

Olivier Lemer

Amélioration 1

Motivation et algorithme

Amélioration 1

Repenser les messages de la mutex de Lamport

3 types de messages

`REQ`
"Si jamais, moi je veux entrer en SC."

`ACK`
"Je note que tu veux entrer en SC."

`REL`
"Si jamais, j'ai fini ma SC."

Remarques ?

Significations assez indirectes par rapport au but : entrer en SC.
Quantité d'information par message sous-optimale.

Suggestion

Simplifier et aller droit au but.

`REQ`
"Si jamais, moi je veux entrer en SC."

`OK`
"Pour moi, tu peux entrer en SC."

→ *"Quand j'ai tous les* `OK`*, je peux entrer en SC."*

Amélioration 1

Repenser les messages de la mutex de Lamport

On passe de l'objectif

"chacun doit connaitre un état global"

`REQ`
"Si jamais, moi je veux entrer en SC."

`ACK`
"Je note que tu veux entrer en SC."

`REL`
"Si jamais, j'ai fini ma SC."

à l'objectif

"chacun se concentre sur soi, et donne sa permission"

`REQ`
"Si jamais, moi je veux entrer en SC."

`OK`
"Pour moi, tu peux entrer en SC."

→ *"Quand j'ai tous les OK, je peux entrer en SC."*

Amélioration 1

Algorithme avec messages simplifiés

App veut entrer en SC

J'envoie un `REQ` à tout le monde

App sort de SC

Je réponds par `OK` à tous les `REQ` en attente.

Réception d'un `REQ`

Si je ne suis pas en SC

- Si je ne veux pas entrer en SC
  - Je réponds avec `OK`
- Si je veux entrer en SC
  - Si je n'ai pas la priorité, je réponds avec `OK`.
  - Si j'ai la priorité, je stocke le `REQ` en attente.

Si je suis en SC

- J'attends d'avoir fini, je stocke le `REQ` en attente.

Réception d'un `OK`

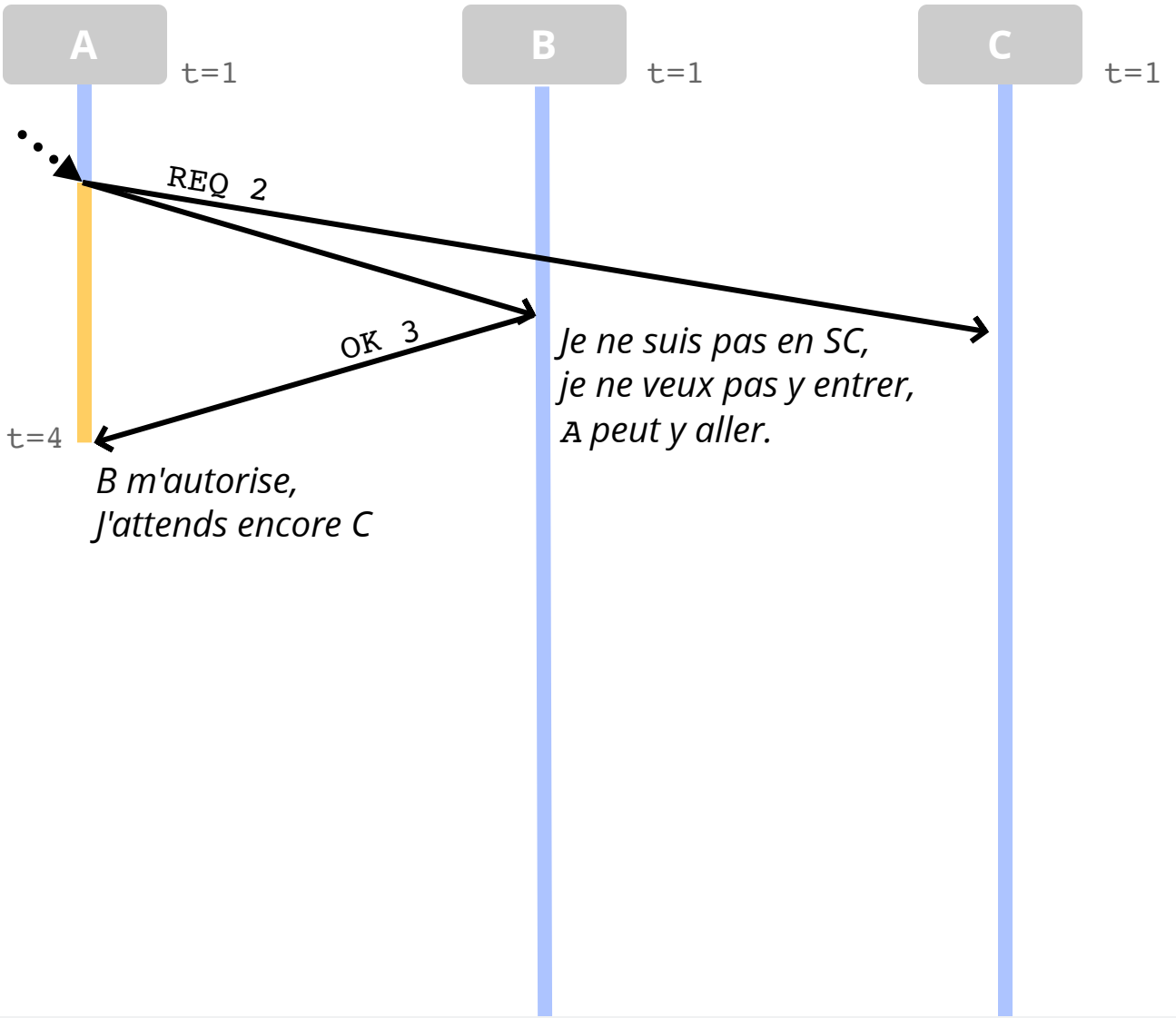
Je le comptabilise.

- Si j'ai le `OK` de tout le monde, alors je peux entrer en SC.

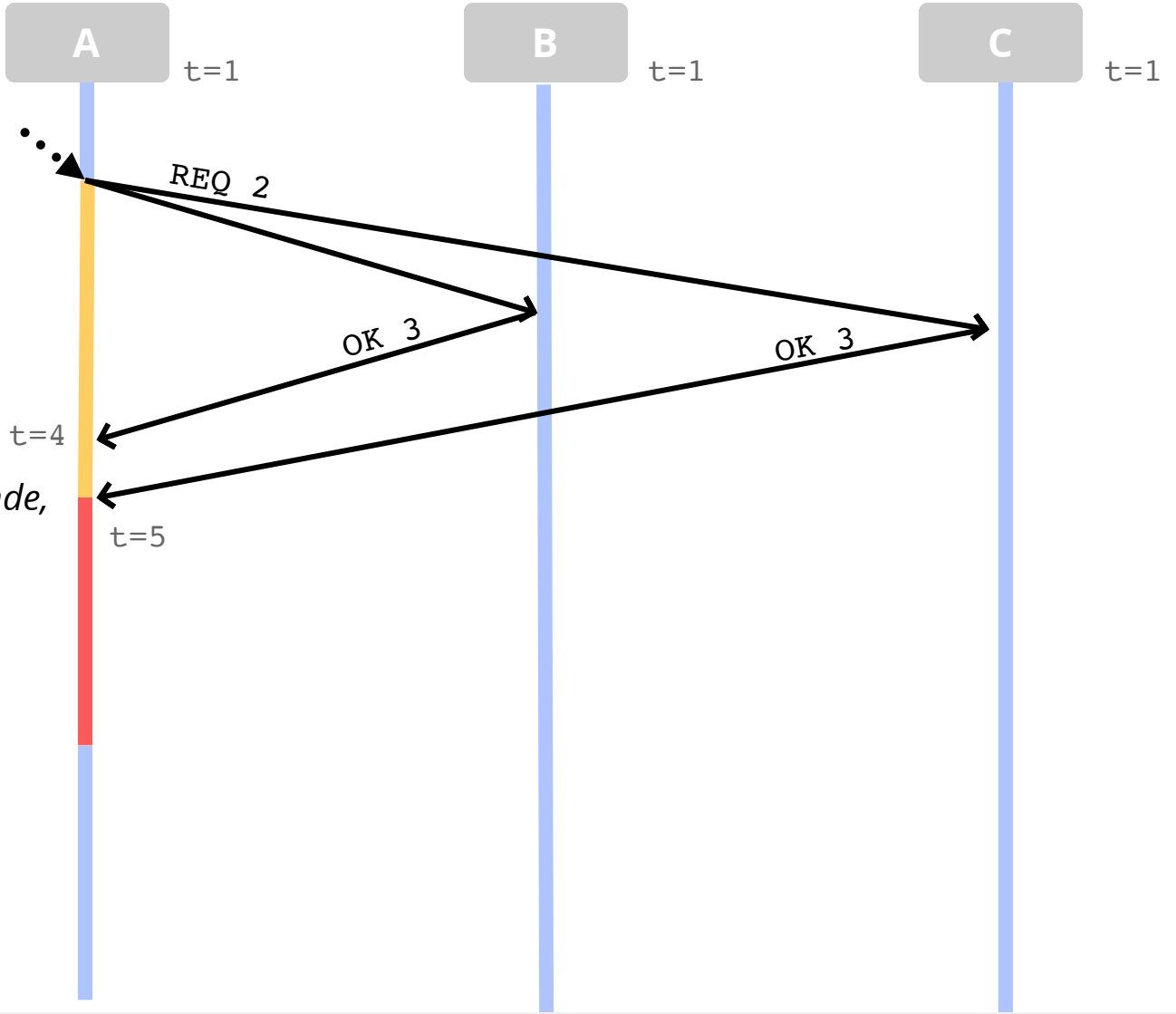
Amélioration 1

Exemple

Exemple

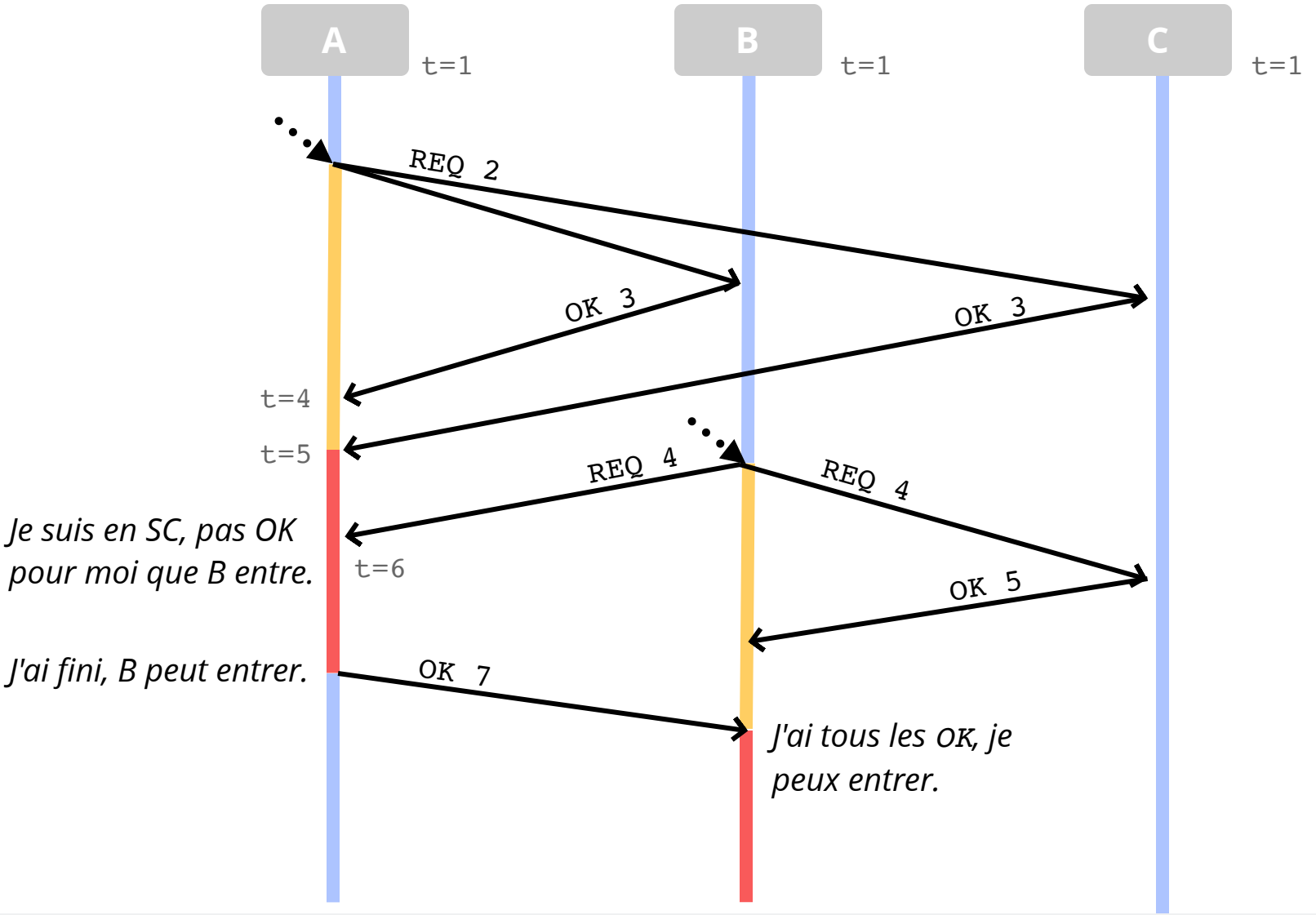

A
B
C
t=1
t=1
t=1
REQ 2
OK 3
Je ne suis pas en SC,
je ne veux pas y entrer,
A peut y aller.
t=4
B m'autorise,
J'attends encore C

Exemple

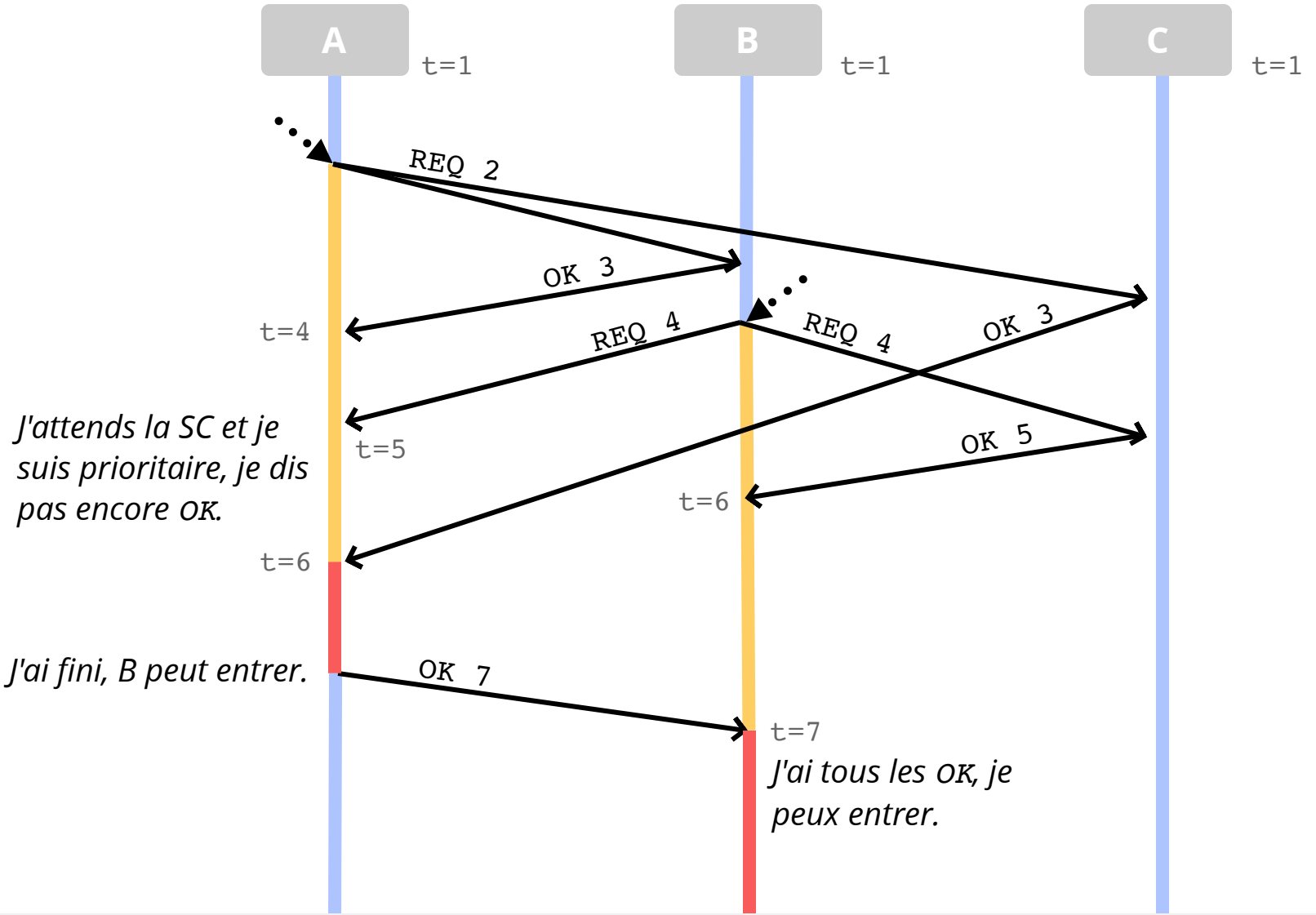


OK de tout le monde,
je peux entrer

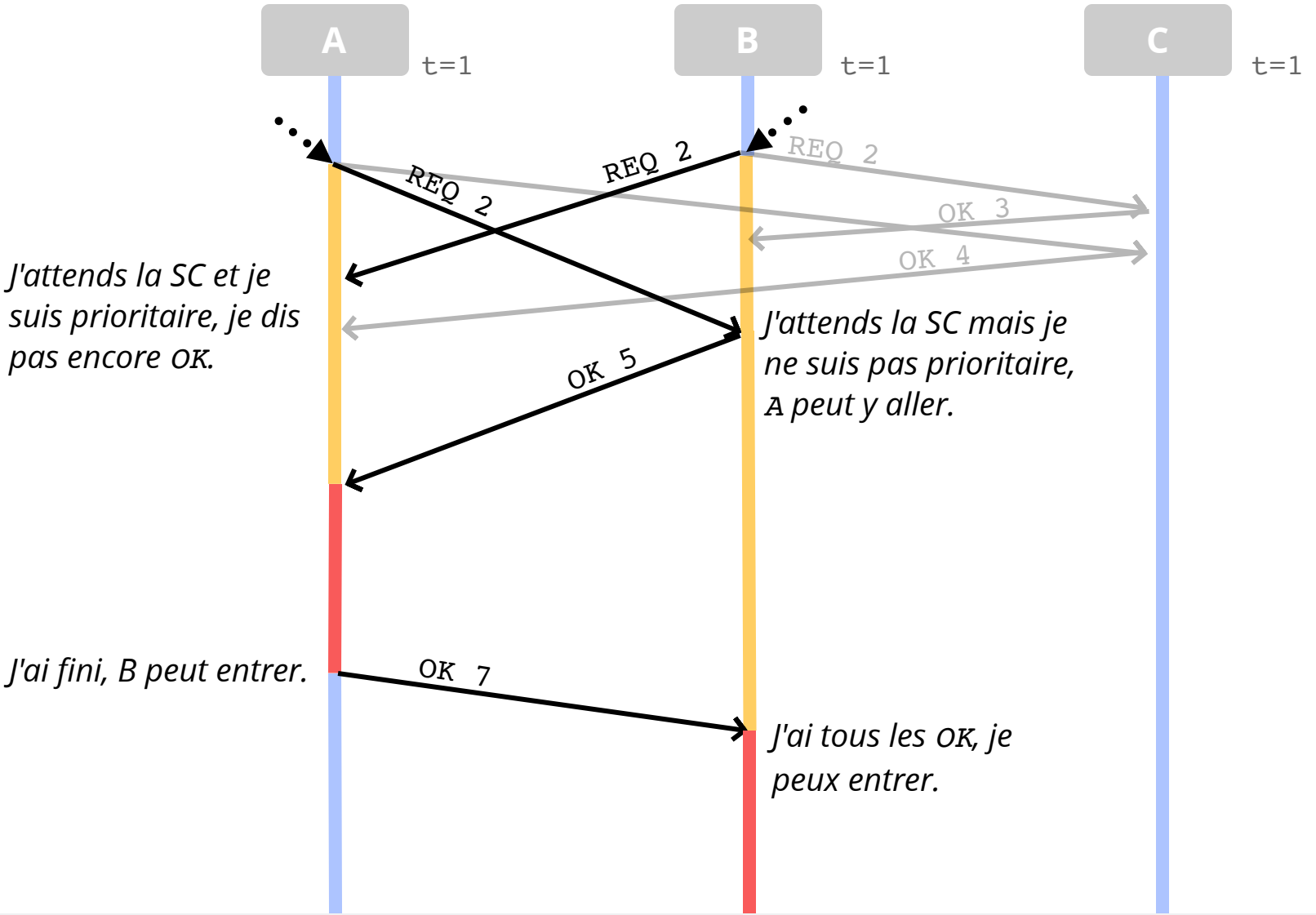
Exemple


A
B
C
t=1
t=1
t=1
REQ 2
OK 3
OK 3
t=4
t=5
REQ 4
REQ 4
Je suis en SC, pas OK pour moi que B entre.
t=6
OK 5
J'ai fini, B peut entrer.
OK 7
J'ai tous les OK, je peux entrer.

Exemple


A
B
C
t=1
t=1
t=1
REQ 2
OK 3
t=4
REQ 4
REQ 4
OK 3
J'attends la SC et je suis prioritaire, je dis pas encore OK.
t=5
OK 5
t=6
t=6
J'ai fini, B peut entrer.
OK 7
t=7
J'ai tous les OK, je peux entrer.

Exemple


A
B
C
t=1
t=1
t=1
REQ 2
REQ 2
REQ 2
OK 3
OK 4
J'attends la SC et je suis prioritaire, je dis pas encore OK.
J'attends la SC mais je ne suis pas prioritaire, A peut y aller.
OK 5
J'ai fini, B peut entrer.
OK 7
J'ai tous les OK, je peux entrer.

Amélioration 1

Propriétés

Amélioration 1

a.k.a. Ricart et Agrawala, 1981

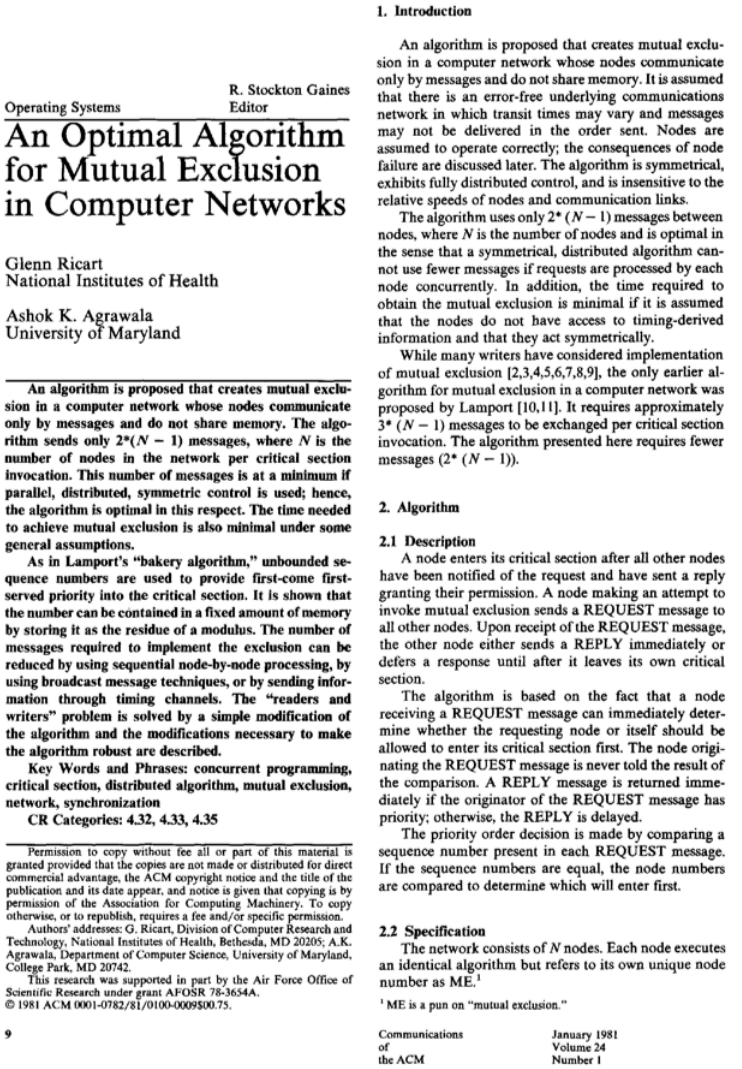
Propriétés

Correctness Jamais plus d'un processus ne sera en SC à un instant T.

Progrès Toute demande d'entrée en SC sera autorisée un jour.

Complexité de communication

$2(n-1)$ messages par processus souhaitant entrer en SC.



Operating Systems — R. Stockton Gaines, Editor

An Optimal Algorithm for Mutual Exclusion in Computer Networks

Glenn Ricart
National Institutes of Health

Ashok K. Agrawala
University of Maryland

An algorithm is proposed that creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. The algorithm sends only $2*(N-1)$ messages, where N is the number of nodes in the network per critical section invocation. This number of messages is at a minimum if parallel, distributed, symmetric control is used; hence, the algorithm is optimal in this respect. The time needed to achieve mutual exclusion is also minimal under some general assumptions.

As in Lamport's "bakery algorithm," unbounded sequence numbers are used to provide first-come first-served priority into the critical section. It is shown that the number can be contained in a fixed amount of memory by storing it as the residue of a modulus. The number of messages required to implement the exclusion can be reduced by using sequential node-by-node processing, by using broadcast message techniques, or by sending information through timing channels. The "readers and writers" problem is solved by a simple modification of the algorithm and the modifications necessary to make the algorithm robust are described.

Key Words and Phrases: concurrent programming, critical section, distributed algorithm, mutual exclusion, network, synchronization

CR Categories: 4.32, 4.33, 4.35

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Authors' addresses: G. Ricart, Division of Computer Research and Technology, National Institutes of Health, Bethesda, MD 20205; A.K. Agrawala, Department of Computer Science, University of Maryland, College Park, MD 20742.

This research was supported in part by the Air Force Office of Scientific Research under grant AFOSR 78-3654A.

© 1981 ACM 0001-0782/81/0100-0009$00.75.

1. Introduction

An algorithm is proposed that creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. It is assumed that there is an error-free underlying communications network in which transit times may vary and messages may not be delivered in the order sent. Nodes are assumed to operate correctly; the consequences of node failure are discussed later. The algorithm is symmetrical, exhibits fully distributed control, and is insensitive to the relative speeds of nodes and communication links.

The algorithm uses only $2*(N-1)$ messages between nodes, where N is the number of nodes and is optimal in the sense that a symmetrical, distributed algorithm cannot use fewer messages if requests are processed by each node concurrently. In addition, the time required to obtain the mutual exclusion is minimal if it is assumed that the nodes do not have access to timing-derived information and that they act symmetrically.

While many writers have considered implementation of mutual exclusion [2,3,4,5,6,7,8,9], the only earlier algorithm for mutual exclusion in a computer network was proposed by Lamport [10,11]. It requires approximately $3*(N-1)$ messages to be exchanged per critical section invocation. The algorithm presented here requires fewer messages ($2*(N-1)$).

2. Algorithm

2.1 Description

A node enters its critical section after all other nodes have been notified of the request and have sent a reply granting their permission. A node making an attempt to invoke mutual exclusion sends a REQUEST message to all other nodes. Upon receipt of the REQUEST message, the other node either sends a REPLY immediately or defers a response until after it leaves its own critical section.

The algorithm is based on the fact that a node receiving a REQUEST message can immediately determine whether the requesting node or itself should be allowed to enter its critical section first. The node originating the REQUEST message is never told the result of the comparison. A REPLY message is returned immediately if the originator of the REQUEST message has priority; otherwise, the REPLY is delayed.

The priority order decision is made by comparing a sequence number present in each REQUEST message. If the sequence numbers are equal, the node numbers are compared to determine which will enter first.

2.2 Specification

The network consists of N nodes. Each node executes an identical algorithm but refers to its own unique node number as ME.[1]

[1] ME is a pun on "mutual exclusion."



Ricart et Agrawala

Pseudocode

Pseudocode

Variables

`n`	*entier, constant*	nombre de processus
`self`	*entier, entre 0 et n-1*	mon numéro de processus
`ts`	*entier, init 0*	mon timestamp Lamport actuel
`hasRequested`	*booléen, init false*	ssi j'ai fait une demande de SC
`selfRequestTs`	*entier*	timestamp de cette demande
`missingOKs`	*entier*	nombre de OKs que j'attends encore
`waitingPs`	*ensemble d'entiers*	numéros des processus qui attendent mon OK.

Pseudocode

Initialisation

Écouter infiniment les événements suivants:

- Demande de SC de la couche applicative
- Sortie de SC dans la couche applicative
- Passage de `missingOKs` à `0`
- Réception de `{REQ, tsi}` du processus `i`
- Réception de `{OK, tsi}` du processus `i`

(Avec traitement séquentiel de chaque événement)

Pseudocode

Traitement : Demande de SC de la couche applicative.

```
ts += 1
hasRequested ← true
selfRequestTs ← ts
missingOKs ← n-1
```

Envoi de `{REQ, ts}` à tous les autres processus.

Traitement : Passage de `missingOKs` à `0`.

Autorisation à la couche applicative d'entrer en SC.

Traitement : Sortie de SC par la couche applicative.

```
ts += 1
hasRequested ← false
```

Envoi de `{OK, ts}` à tous les processus de `waitingPs`

Réinitialisation de `waitingPs`

Pseudocode

Traitement : Réception `{REQ, tsi}` du processus `i`.

```
ts ← max(tsi, ts) + 1
Si hasRequested et
   (selfRequestTs < tsi ou
   (selfRequestTs == tsi et self < i))
     Ajout de i dans waitingPs
Sinon
     Envoi de {OK, ts} à i
```

Traitement : Réception `{OK, tsi}` du processus `i`.

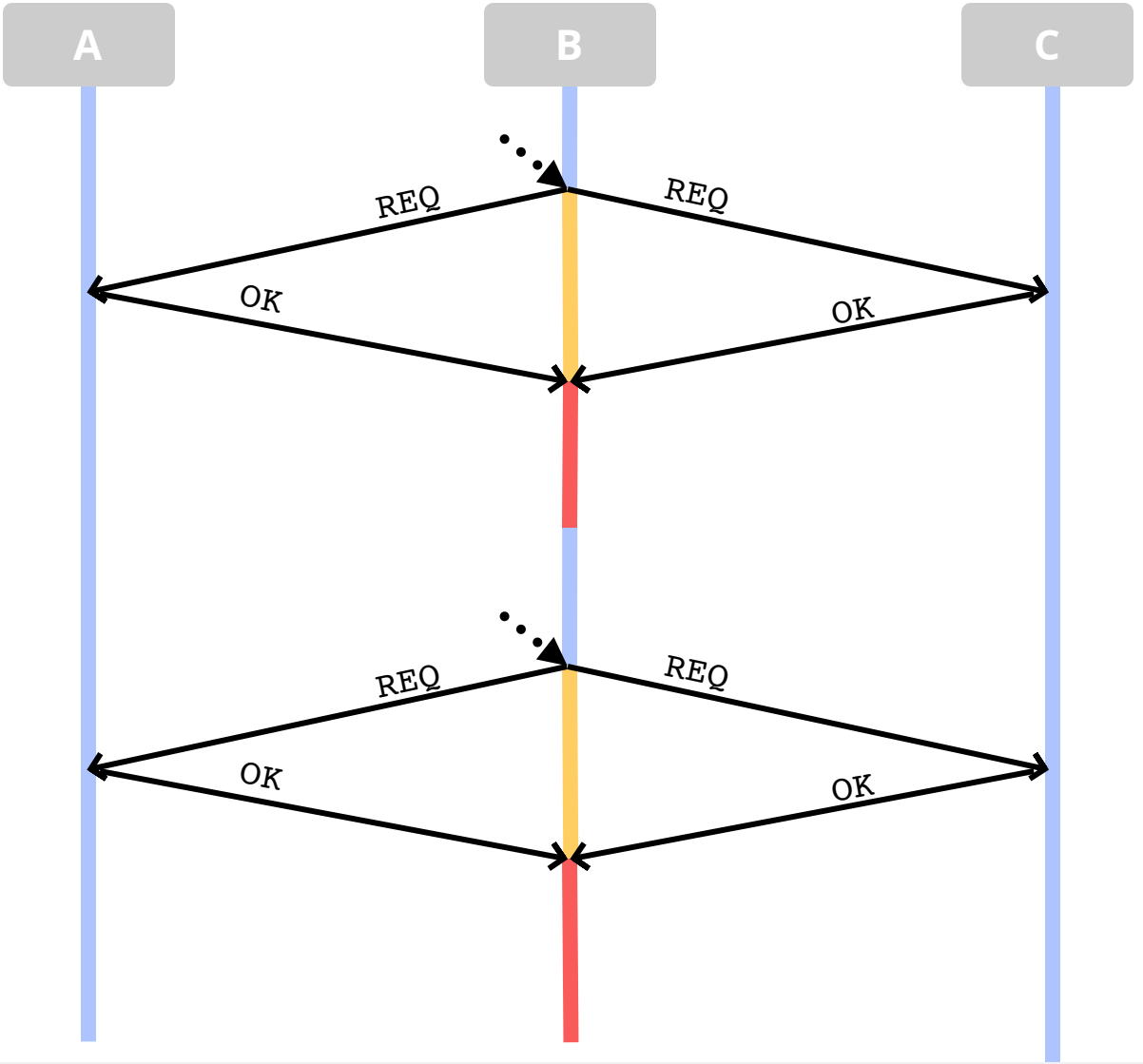
```
ts ← max(tsi, ts) + 1
missingOKs -= 1
```

Amélioration 1

Observations

Ricart et Agrawala, 1981

Un cas particulier



Amélioration 2
Ce cas particulier

2 types de messages

`REQ`	`OK`
"Si jamais, moi je veux entrer en SC"	*"Pour moi, tu peux entrer en SC"*

Suggestion
Tant qu'on ne m'a rien dit depuis "tu peux entrer en SC", je me permets.

Remarque
C'est comme partager un "jeton de permission" unique avec son voisin.

`REQ`	`OK`
"Passe-moi le jeton de permission"	*"Tiens"*

Si j'ai les jetons de tous mes voisins, je fais ce que je veux.

Amélioration 2

Avec des jetons ?

Jeton de permission

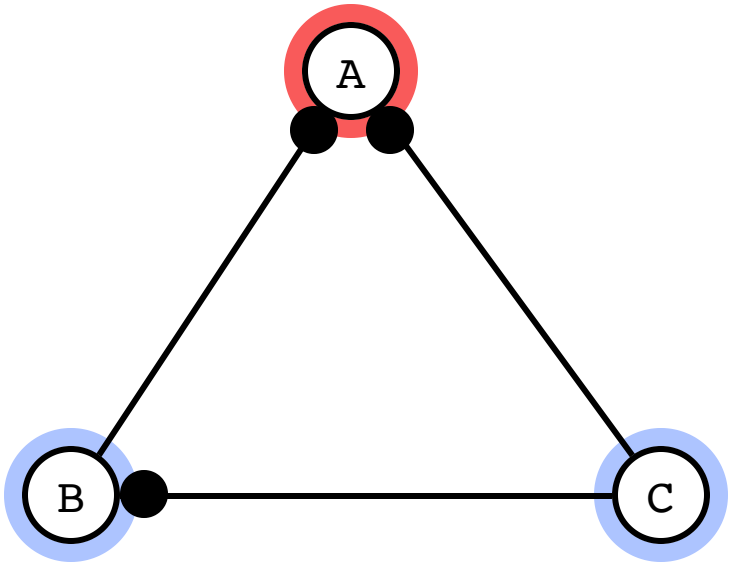
Exemple 1

3 jetons :

- `A` ↔ `B` : `A`
- `A` ↔ `C` : `A`
- `B` ↔ `C` : `B`

`A` a tous les jetons

`A` est en SC



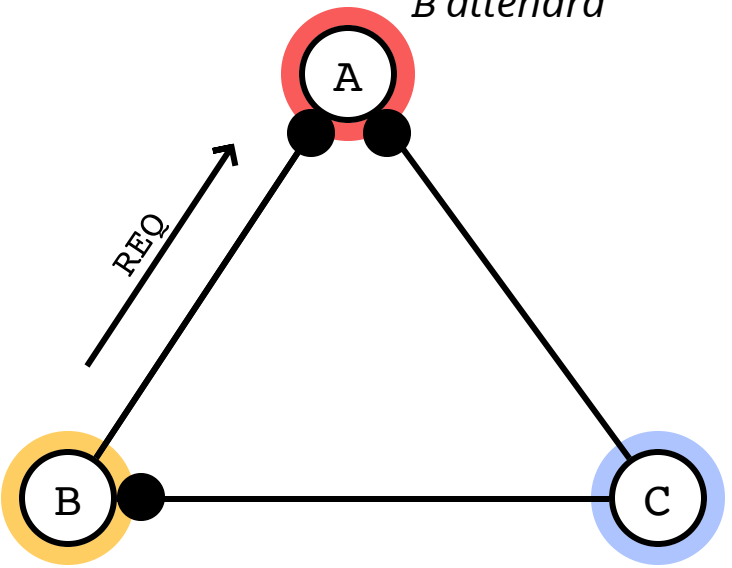
Jeton de permission

Exemple 1

3 jetons :

- A ↔ B : A
- A ↔ C : A
- B ↔ C : B

"J'ai encore besoin du jeton, B attendra"



"Je veux entrer en SC mais il me manque un jeton"

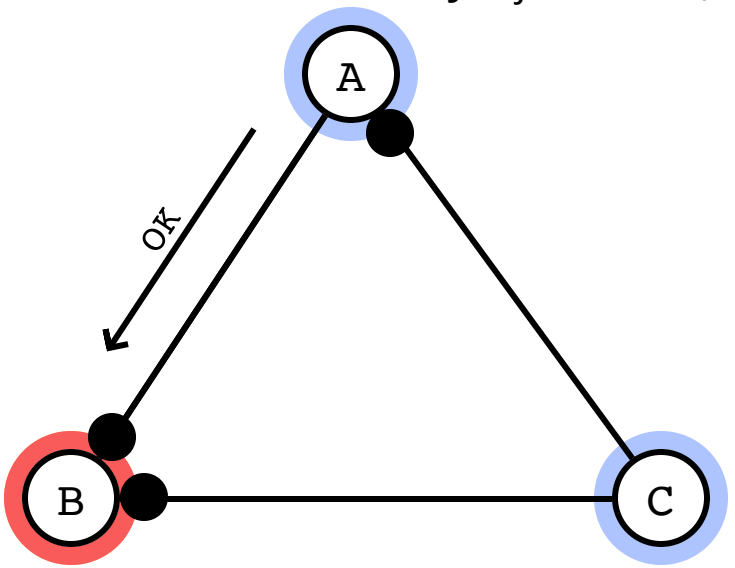
Jeton de permission

Exemple 1

3 jetons :

- A ↔ B : B
- A ↔ C : A
- B ↔ C : B

"J'ai fini ma SC, tiens"



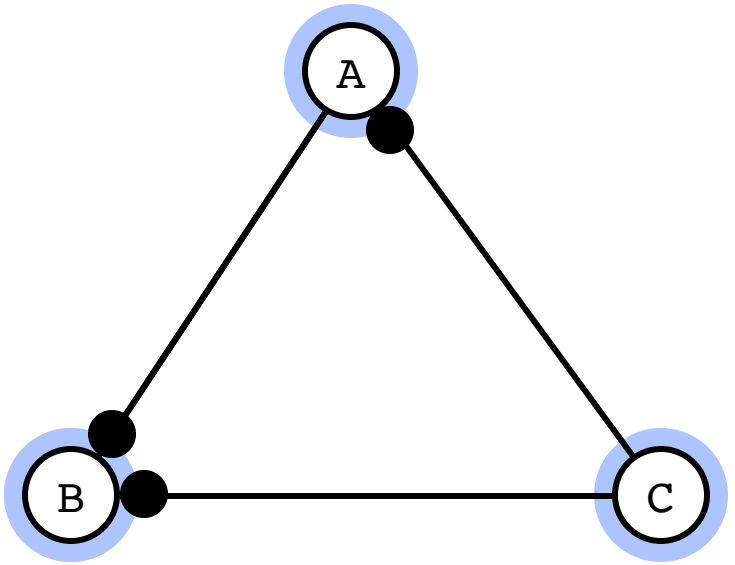
*"J'ai tous les jetons,
je peux entrer en SC"*

Jeton de permission

Exemple 1

3 jetons :

- `A ↔ B : B`
- `A ↔ C : A`
- `B ↔ C : B`



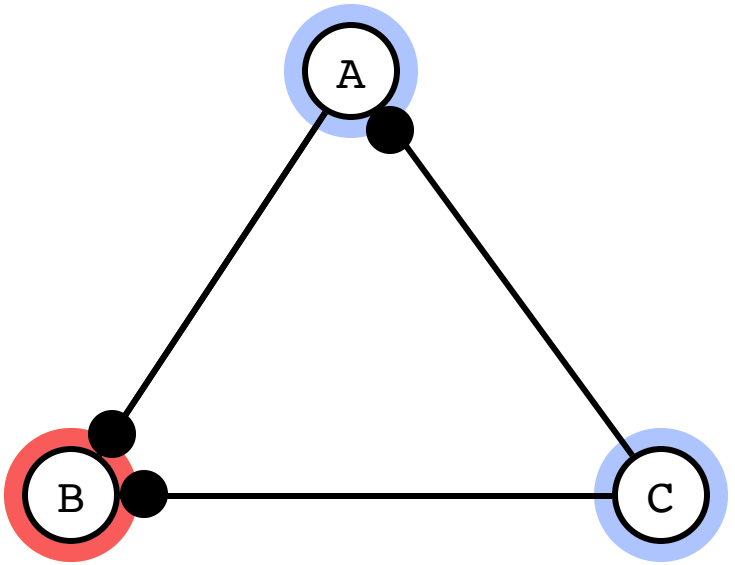
"J'ai fini, personne n'a envie de mes jetons, je les garde."

Jeton de permission

Exemple 1

3 jetons :

- `A ↔ B : B`
- `A ↔ C : A`
- `B ↔ C : B`

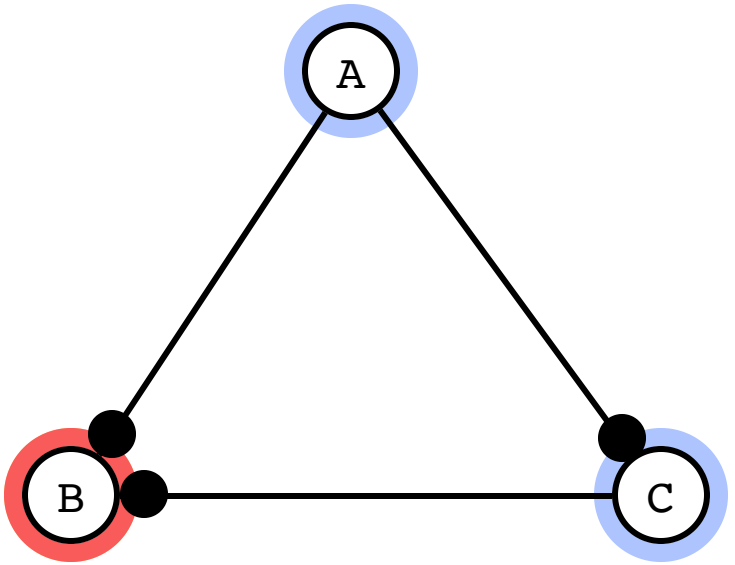


"Je veux entrer en SC,
j'ai encore tous les jetons,
je peux y aller."

↓ Exemple 2

Jeton de permission

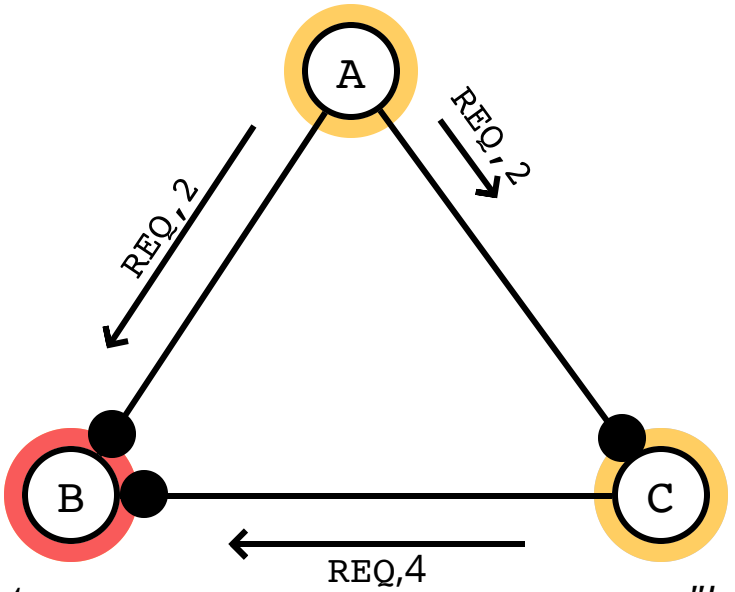
Exemple 2



Jeton de permission

Exemple 2

"Je veux entrer en SC,
j'ai besoin des deux jetons"

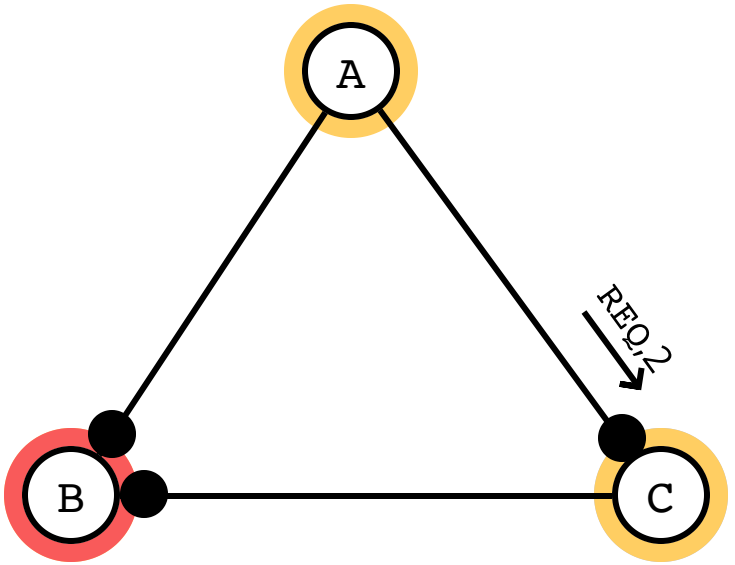


"J'ai encore besoin des jetons,
ils attendront."

"Je veux entrer en SC,
j'ai besoin du jeton de B"

Jeton de permission

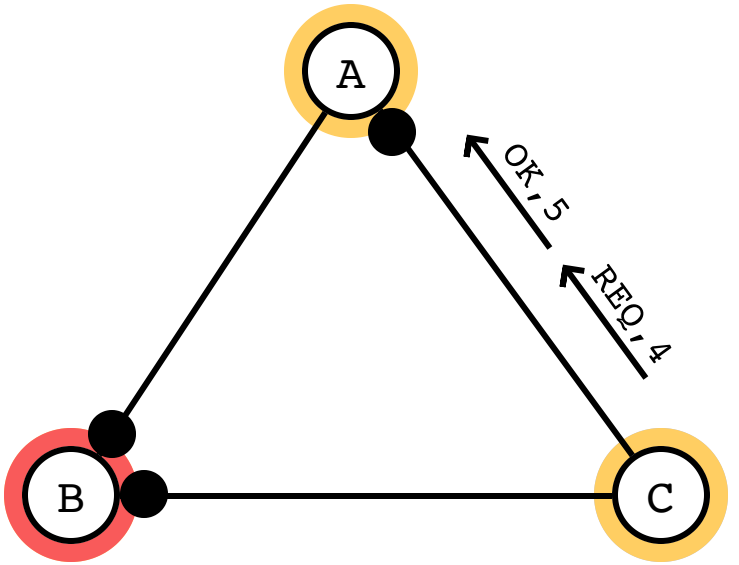
Exemple 2



"Tu as la priorité sur moi, donc d'accord, prends"

Jeton de permission

Exemple 2



"Tu as la priorité sur moi, donc d'accord, prends"

"Mais je le re-veux après"

Amélioration 2

Algorithme & Propriétés

Algorithme

Avec 1 jeton par voisin

App veut entrer en SC

J'envoie un `REQ` à ceux qui ont le jeton.

App sort de SC

Je passe mon jeton à tous les demandeurs.

Réception d'un `REQ`

Si je suis en SC

- J'attends d'avoir fini, puis je passe le jeton.

Si je ne suis pas en SC

- Si je veux entrer en SC
  - Si j'ai la priorité, j'attends d'avoir fini, puis je passe le jeton.
  - Si je n'ai pas la priorité, je passe le jeton, **et je renvoie mon `REQ`.**
- Si je ne veux pas entrer en SC
  - Je passe le jeton.

Réception d'un `OK`

Je note avoir reçu le jeton.

Si j'ai tous les jetons

Je peux entrer en SC quand je veux.

← Le req *initial* est renvoyé, avec son timestamp d'origine.

Amélioration 2

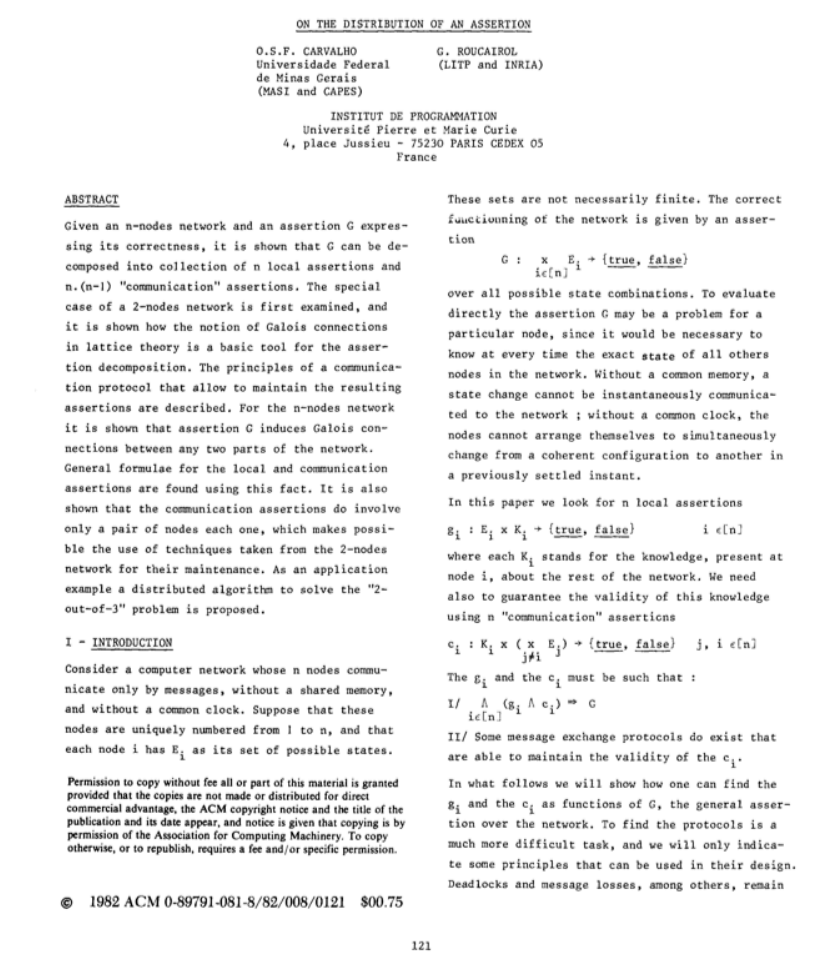
a.k.a. Carvalho et Roucairol, 1982

Propriétés

Correctness Jamais plus d'un processus ne sera en SC à un instant T.

Progrès Toute demande d'entrée en SC sera autorisée un jour.

Complexité de communication
Entre 0 et $2(n-1)$ messages par processus souhaitant entrer en SC.



ON THE DISTRIBUTION OF AN ASSERTION

O.S.F. CARVALHO
Universidade Federal de Minas Gerais
(MASI and CAPES)

G. ROUCAIROL
(LITP and INRIA)

INSTITUT DE PROGRAMMATION
Université Pierre et Marie Curie
4, place Jussieu - 75230 PARIS CEDEX 05
France

ABSTRACT

Given an n-nodes network and an assertion G expressing its correctness, it is shown that G can be decomposed into collection of n local assertions and n.(n-1) "communication" assertions. The special case of a 2-nodes network is first examined, and it is shown how the notion of Galois connections in lattice theory is a basic tool for the assertion decomposition. The principles of a communication protocol that allow to maintain the resulting assertions are described. For the n-nodes network it is shown that assertion G induces Galois connections between any two parts of the network. General formulae for the local and communication assertions are found using this fact. It is also shown that the communication assertions do involve only a pair of nodes each one, which makes possible the use of techniques taken from the 2-nodes network for their maintenance. As an application example a distributed algorithm to solve the "2-out-of-3" problem is proposed.

I - INTRODUCTION

Consider a computer network whose n nodes communicate only by messages, without a shared memory, and without a common clock. Suppose that these nodes are uniquely numbered from 1 to n, and that each node i has E_i as its set of possible states.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-081-8/82/008/0121 $00.75

These sets are not necessarily finite. The correct functionning of the network is given by an assertion

$$G : \underset{i\in[n]}{\times} E_i \to \{\underline{true}, \underline{false}\}$$

over all possible state combinations. To evaluate directly the assertion G may be a problem for a particular node, since it would be necessary to know at every time the exact state of all others nodes in the network. Without a common memory, a state change cannot be instantaneously communicated to the network ; without a common clock, the nodes cannot arrange themselves to simultaneously change from a coherent configuration to another in a previously settled instant.

In this paper we look for n local assertions

$$g_i : E_i \times K_i \to \{\underline{true}, \underline{false}\} \qquad i \in[n]$$

where each K_i stands for the knowledge, present at node i, about the rest of the network. We need also to guarantee the validity of this knowledge using n "communication" assertions

$$c_i : K_i \times (\underset{j\neq i}{\times} E_j) \to \{\underline{true}, \underline{false}\} \quad j, i \in[n]$$

The g_i and the c_i must be such that :

I/ $\underset{i\in[n]}{\wedge} (g_i \wedge c_i) \Rightarrow G$

II/ Some message exchange protocols do exist that are able to maintain the validity of the c_i.

In what follows we will show how one can find the g_i and the c_i as functions of G, the general assertion over the network. To find the protocols is a much more difficult task, and we will only indicate some principles that can be used in their design. Deadlocks and message losses, among others, remain

Carvalho et Roucairol

Pseudocode

Pseudocode

Variables

`n`	*entier, constant*	nombre de processus
`self`	*entier, entre 0 et n-1*	mon numéro de processus
`ts`	*entier, init 0*	mon timestamp Lamport actuel
`hasRequested`	*booléen, init false*	ssi j'ai fait une demande de SC
`selfRequestTs`	*entier*	timestamp de cette demande
`requesters`	*ensemble d'entiers*	numéros des processus qui attendent mon jeton.
`jetons`	*ensemble d'entiers*	numéros des processus dont j'ai le jeton. Initialement arbitraire.

Pseudocode

Initialisation

Écouter infiniment les événements suivants:

- Demande de SC de la couche applicative
- Sortie de SC dans la couche applicative
- Réception de `{REQ, tsi}` du processus `i`
- Réception de `{OK, tsi}` du processus `i`

(Avec traitement séquentiel de chaque événement)

Pseudocode

Traitement : Demande de SC de la couche applicative.

```
ts += 1
hasRequested ← true
selfRequestTs ← ts
Pour chaque processus i qui n'est pas dans jetons :
    Envoi de {REQ, ts}
Si jetons a une taille n-1, entrer en SC
```

Traitement : Sortie de SC par la couche applicative.

```
ts += 1
hasRequested ← false
Pour tout i dans requesters :
    Envoi de {OK, ts}
    jetons.remove(i)
Réinitialisation de requesters
```

Pseudocode

Traitement : Réception `{REQ, tsi}` du processus `i`.

```
ts ← max(tsi, ts) + 1
Si hasRequested et
   (selfRequestTs < tsi ou
   (selfRequestTs == tsi et self < i))
     Ajout de i dans requesters
Sinon
     Envoi de {OK, ts} à i
     jetons.remove(i)
     Si hasRequested
          Envoi de {REQ, selfRequestTs} à i
```

Pseudocode

Traitement : Réception `{OK, tsi}` du processus `i`.

```
ts ← max(tsi, ts) + 1
jetons.add(i)
Si jetons a une taille n-1 et hasRequested :
    Entrer en SC
```