Arhitecturi Paralele
Consens în Sisteme Distribuite

Lect. Dr. Ing. Cristian Chilipirea
cristian.chilipirea@mta.ro

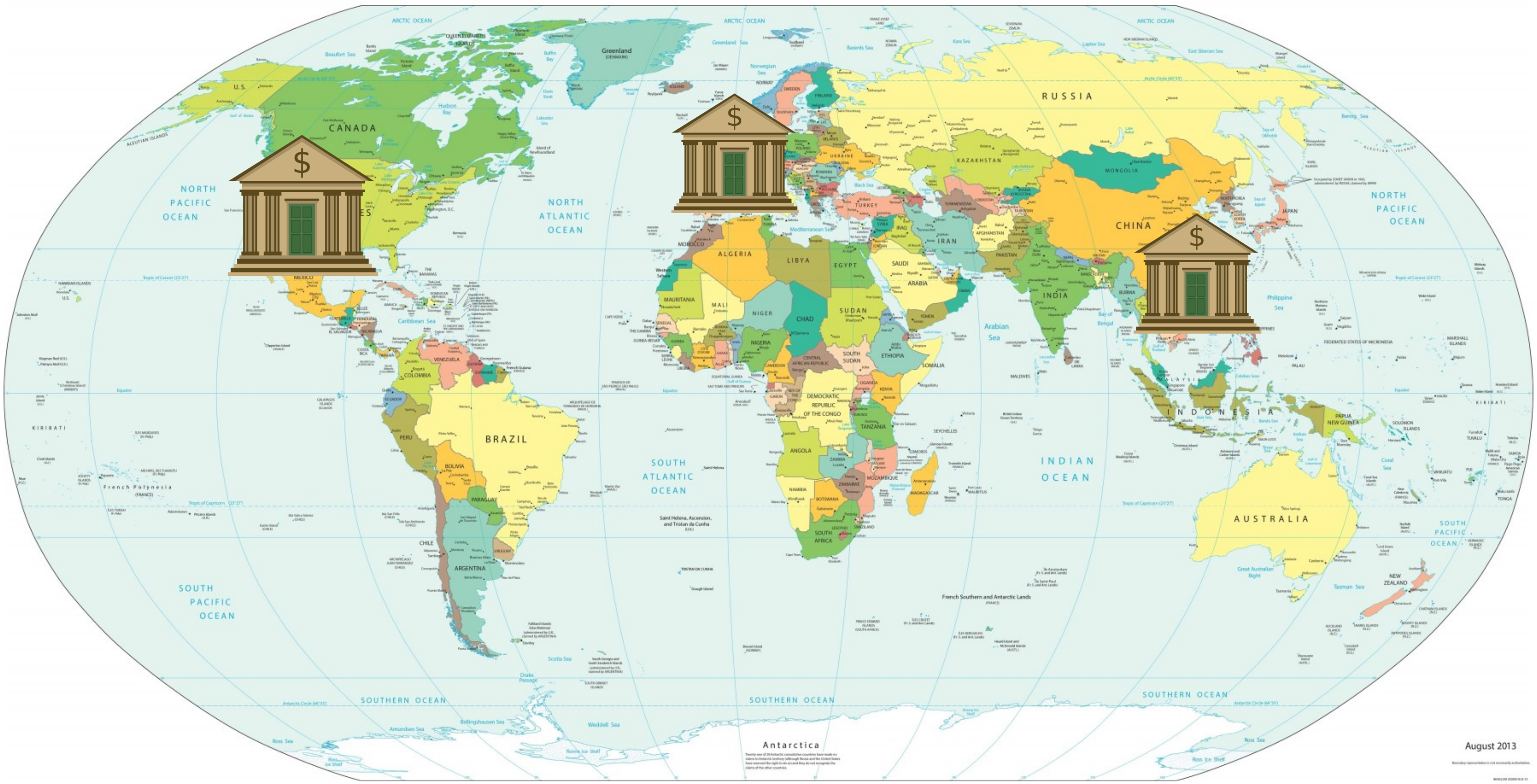
Dorim sisteme distribuite care să

- dea mereu un rezultat corect.
- dea mereu același rezultat având aceleași intrări.
- funcționeze chiar dacă părți din sistem se opresc.
- funcționeze chiar dacă și părți din sistem se comportă defectuos (au bug-uri).
- funcționeze chiar dacă părți din sistem sunt compromise și devin malițioase.

Presupunem un sistem bancar

Presupunem un sistem bancar scalabil

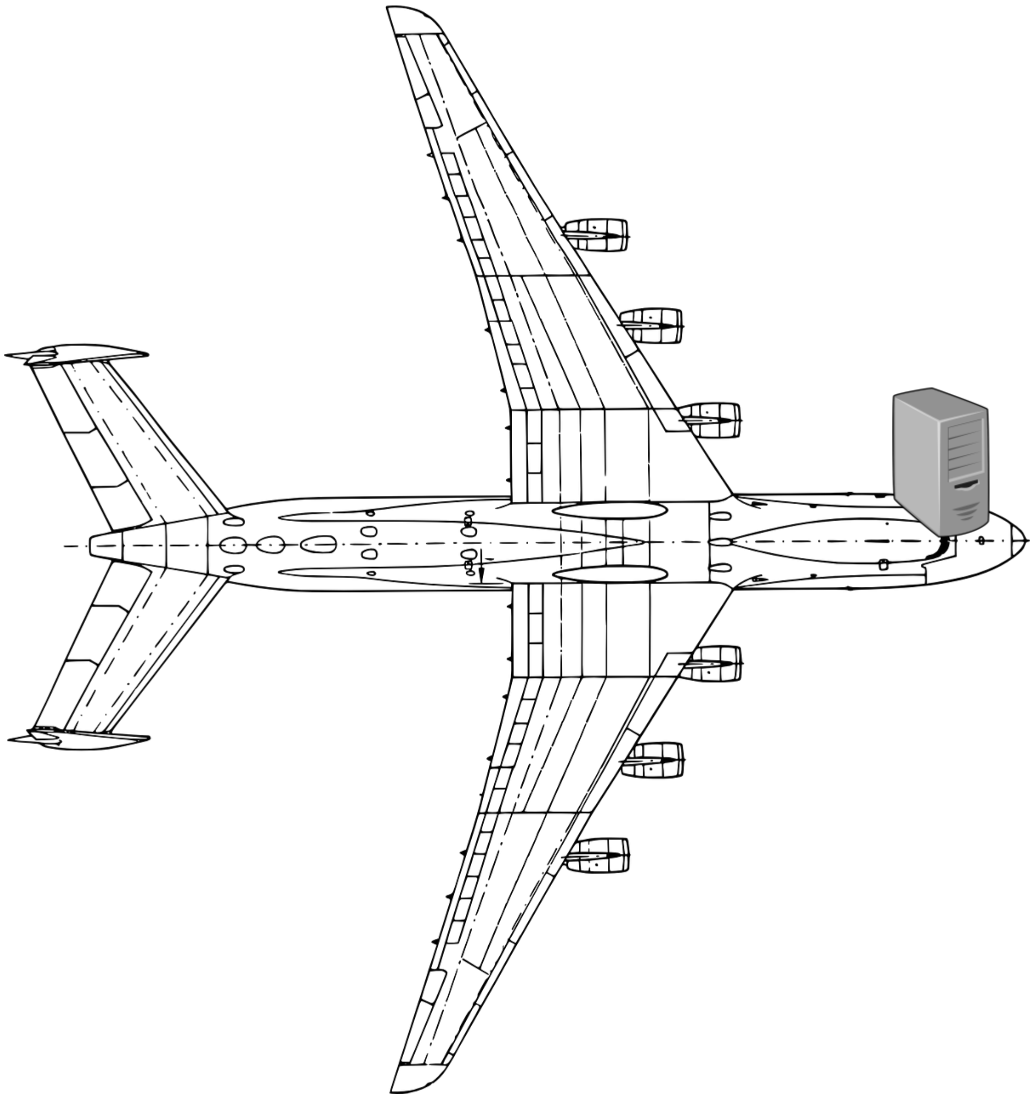
Presupunem un sistem bancar scalabil distribuit global



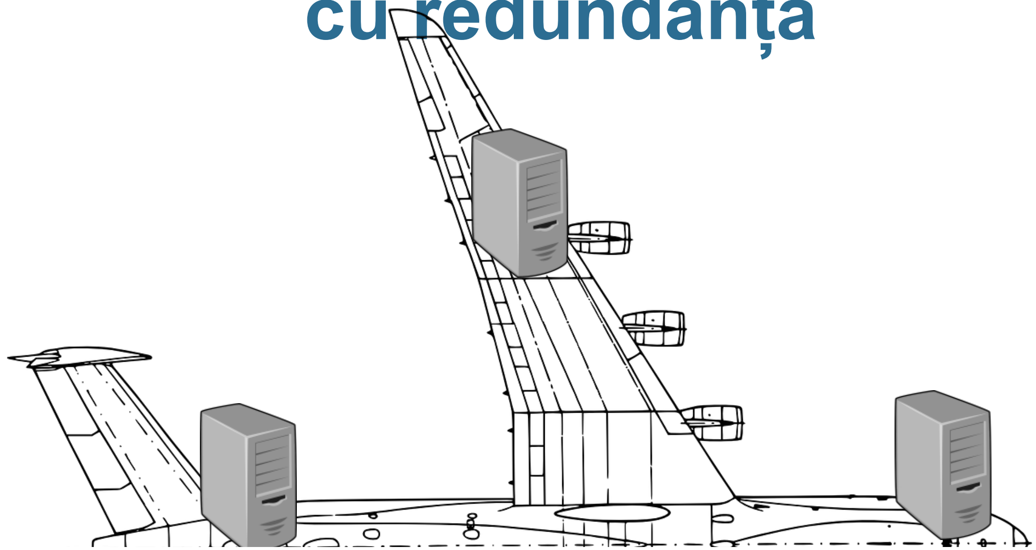
Presupunem un sistem bancar scalabil distribuit global

Dacă pun bani într-o bancă vreau să îi pot scoate din alta.
Dacă scot bani dintr-o bancă când alta calculează o taxă ca procent din banii mei vreau să fie calculat corect.

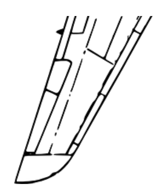
Presupun un sistem ce controlează un avion



Presupun un sistem ce controlează un avion cu redundanță



Dacă un calculator se defectează vreau ca avionul să poată funcționa.
Dacă un calculator dă comenzi greșite vreau ca celelalte să ia decizia corectă.



Generalizare

- Am mai multe sisteme rulând un program distribuit.
- Oricărui sistem îi dau comenzi (sau le generează singur) vreau ca totalitatea lor să vadă comanda ca și executată.
- Ordinea comenzilor (chiar dacă sunt date pe sisteme diferite) este importantă. (A + B) / 2 != A / 2 + B
- Dacă unele sisteme pică, programul distribuit trebuie să funcționeze corect în continuare.
- Dacă unele sisteme se comportă malițios, doresc ca programul distribuit să funcționeze corect în continuare.

- Toate aceste exemple sunt inexistente în programele secvențiale

Generalizare

- Automat avem comunicare asincronă.
- Dacă comunicarea ar fi sincronă și un sistem s-ar opri, un altul ar putea și el rămâne blocat așteptând după acesta.

Multe din probleme ar avea o rezolvare ușoară dacă am putea atașa fiecărei comenzi timpul curent

Dar timpul în sine e o problemă dificilă în sisteme distribuite

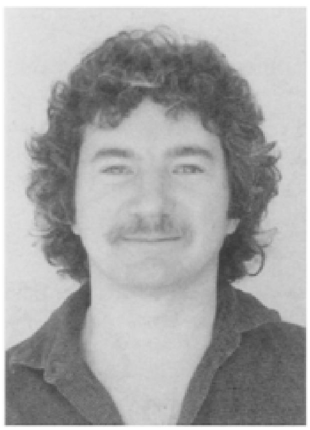
Dacă avem două sisteme avem două “păreri” despre timpul current.

Cu ce protocol se sincronizează ceasul calculatoarelor?

Sincronizarea timpilor între 2 sisteme

Probabilistic clock synchronization

Flaviu Cristian
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA



Flaviu Cristian is a computer scientist at the IBM Almaden Research Center in San Jose, California. He received his PhD from the University of Grenoble, France, in 1979. After carrying out research in operating systems and programming methodology in France, and working on the specification, design, and verification of fault-tolerant programs in England, he joined IBM in 1982. Since then he has worked in the area of fault-tolerant distributed protocols and systems. He has participated in the design and implementation of a highly available system prototype at the Almaden Research Center and has reviewed and consulted for several fault-tolerant distributed system designs, both in Europe and in the American divisions of IBM. He is now a technical leader in the design of a new U.S. Air Traffic Control System which must satisfy very stringent availability requirements.

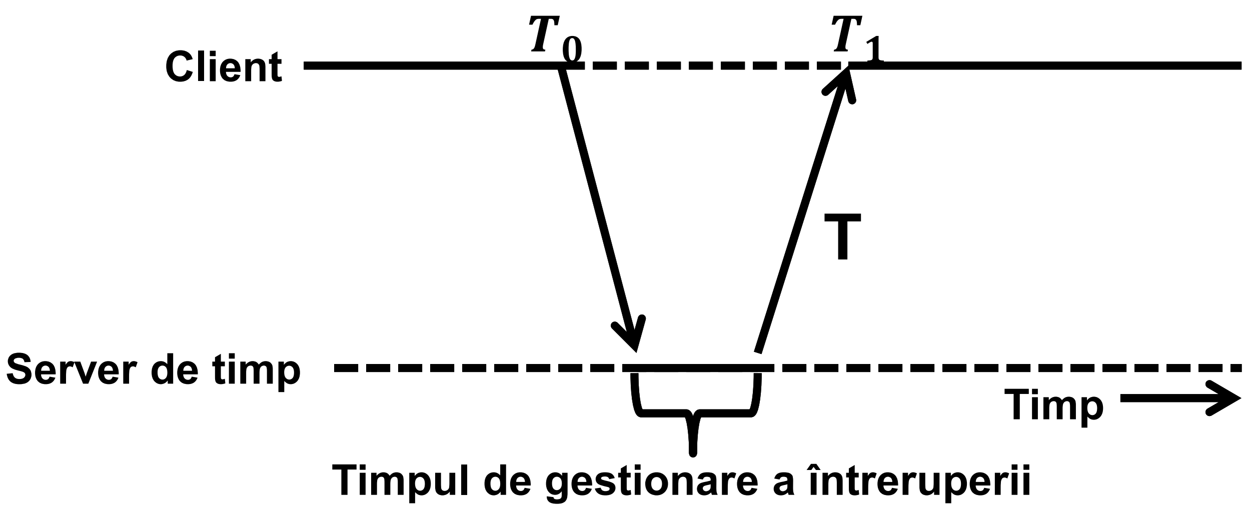
Abstract. A probabilistic method is proposed for reading remote clocks in distributed systems subject to unbounded random communication delays.

in some given maximum derivation from a time reference external to the system. *Internal* clock synchronization keeps processor clocks within some maximum relative deviation of each other. Externally synchronized clocks are also internally synchronized. The converse is not true: as time passes internally synchronized clocks can drift arbitrarily far from external time.

Clock synchronization is needed in many distributed systems. Internal clock synchronization enables one to measure the duration of distributed activities that start on one processor and terminate on another processor and to totally order distributed events in a manner that closely approximates their real time precedence. To allow exchange of information about the timing of events with other systems and users, many systems require external clock synchronization. For example external time can be used to record the occurrence of events for later analysis by humans, to instruct a system to take certain actions when certain specified (external) time deadlines occur, and to order the occurrence of related events observed by distinct sys-

Sincronizarea timpilor între 2 sisteme

T + RTT/2



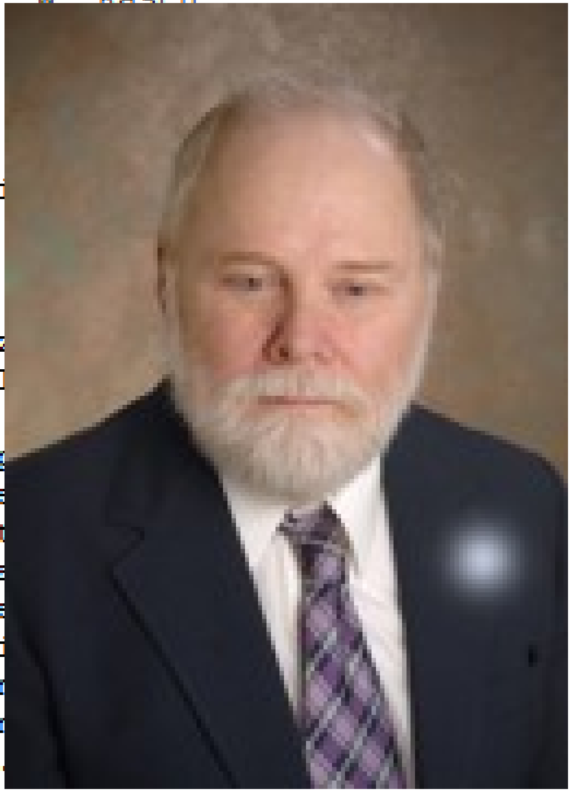
Cu ce protocol se sincronizează ceasul calculatoarelor?

```
Internet Engineering Task Force (IETF)                          D. Mills
Request for Comments: 5905                                   U. Delaware
Obsoletes: 1305, 4330                                     J. Martin, Ed.
Category: Standards Track                                            ISC
ISSN: 2070-1721                                               J. Burbank
                                                                W. Kasch

 Network Time Protocol Version 4: Protocol and Algorithms Speci

Abstract

   The Network Time Protocol (NTP) is widely used to synchroni
   computer clocks in the Internet.  This document describes NT
   4 (NTPv4), which is backwards compatible with NTP version 3
   described in RFC 1305, as well as previous versions of the p
   NTPv4 includes a modified protocol header to accommodate the
   Protocol version 6 address family.  NTPv4 includes fundament
   improvements in the mitigation and discipline algorithms tha
   the potential accuracy to the tens of microseconds with mode
   workstations and fast LANs.  It includes a dynamic server di
   scheme, so that in many cases, specific server configuratio
   required.  It corrects certain errors in the NTPv3 design an
   implementation and includes an optional extension mechanism.
```



Ce “viteză” au calculatorele?

- Reminder: viteză e un cuvânt extrem de nepotrivit

În cât timp va executa următorul program?

```
int main() {
    int i;

    for(i=0; i < 1000 * 1000 * 1000; i++);

    return 0;
}
```

În cât timp va executa următorul program?

```c
int main() {
    int i;

    for(i=0; i < 1000 * 1000 * 1000; i++);

    return 0;
}
```

```
real    0m1.783s
user    0m1.766s
sys     0m0.000s
```

Timp transmitere pachet?

Timp transmitere pachet?

```
root@Nyx:/mnt/d/Dropbox/backupServer/apd-homework/rezults# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=120 time=15.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=120 time=13.0 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=120 time=15.3 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=120 time=14.5 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=120 time=32.5 ms
```

Scala

Termen	Not	Multiplu	10^x	
Tera	T	1 000 000 000 000	12	
Giga	G	1 000 000 000	9	
Mega	M	1 000 000	6	
kilo	k	1 000	3	
UNITATE		1	0	
mili	m	0. 001	-3	Trimitere pachet
micro	μ	0. 000 001	-6	
nano	n	0. 000 000 001	-9	Executare operație
pico	p	0. 000 000 000 001	-12	

Pentru a avea încrede în obținerea proprietăților dorite trebuie să validăm matematic sistemele noastre

Pentru a valida un program matematic acesta trebuie modelat

Modelarea unui sistem simplu/secvențial?

Modelarea unui sistem simplu/secvențial?

First Draft of a Report
on the EDVAC

by

John von Neumann

Modelarea unui sistem simplu/secvențial?

Un set de stări.

Operațiile sau acțiunile modifică starea

Modelarea unui sistem distribuit?

Modelarea unui sistem distribuit?

Obligatoriu trebuie să ia în calcul comunicarea

Se pot executa milioane de operații până la primirea unui mesaj de la un alt sistem

Modelarea unui sistem distribuit: CSP

Programming Techniques

S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays

CR Categories: 4.20, 4.22, 4.32

grams, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (C
(CLU [13]), forms (ALPH

The traditional stored
been designed primarily f
single sequential program
speed has led to the intr
attempt has been made
programmer, either by ha
function units of the CD
in an I/O control packag
erating system). Howeve
technology suggest that a
structed from a number
essors (each with its ow
powerful, capacious, reli
machine which is disguise

In order to use such a
task, the component pro
municate and to synchr
methods of achieving this
adopted method of comm
updating of a common st
and many machine code
severe problems in the co

Modelarea unui sistem distribuit: Rețele Petri

K o m m u n i k a t i o n
m i t
A u t o m a t e n

Von der Fakultät für Mathematik und Physik
der Technischen Hochschule Darmstadt

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer.nat.)

genehmigte
Dissertation

vorgelegt von
C a r l A d a m P e t r i

Modelarea unui sistem distribuit: Mașini de stări

Operating Systems

R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

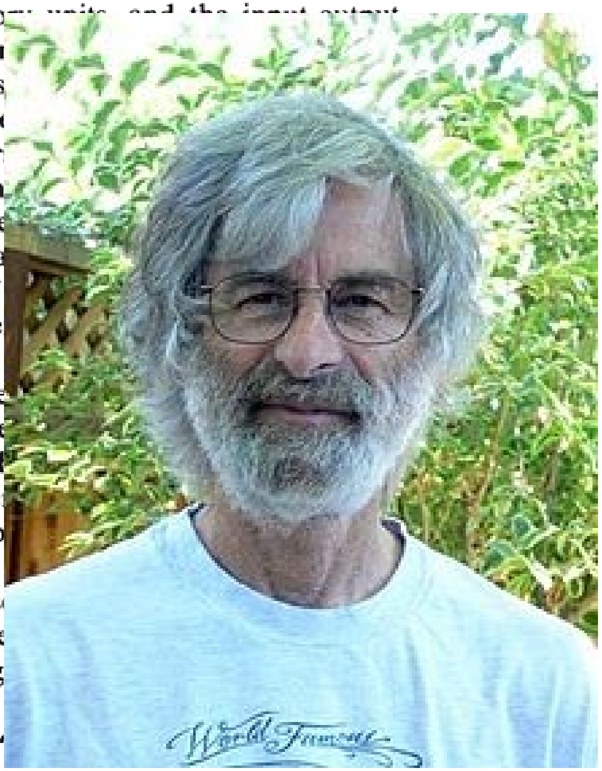
The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memo
channels are separate pr
if the message transmiss
pared to the time betwe

We will concern our
spatially separated com
remarks will apply more
tiprocessing system on a
lems similar to those of
the unpredictable orde
occur.

In a distributed syste
say that one of two eve
"happened before" is th
of the events in the syste
often arise because peop
and its implications.

In this paper, we dis
by the "happened before
algorithm for extending
of all the events. This
mechanism for impleme



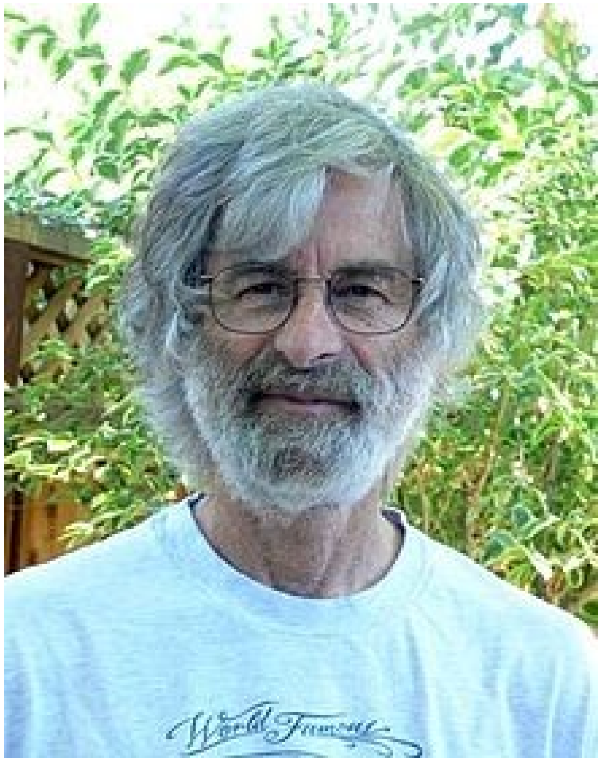
Leslie Lamport

- Majoritatea problemelor/soluțiilor prezentate în acest curs sunt construite de Leslie Lamport.
- Câștigător premiul **Turing 2013**
- Are una din cele mai citate lucrări
- Ne este contemporan și are

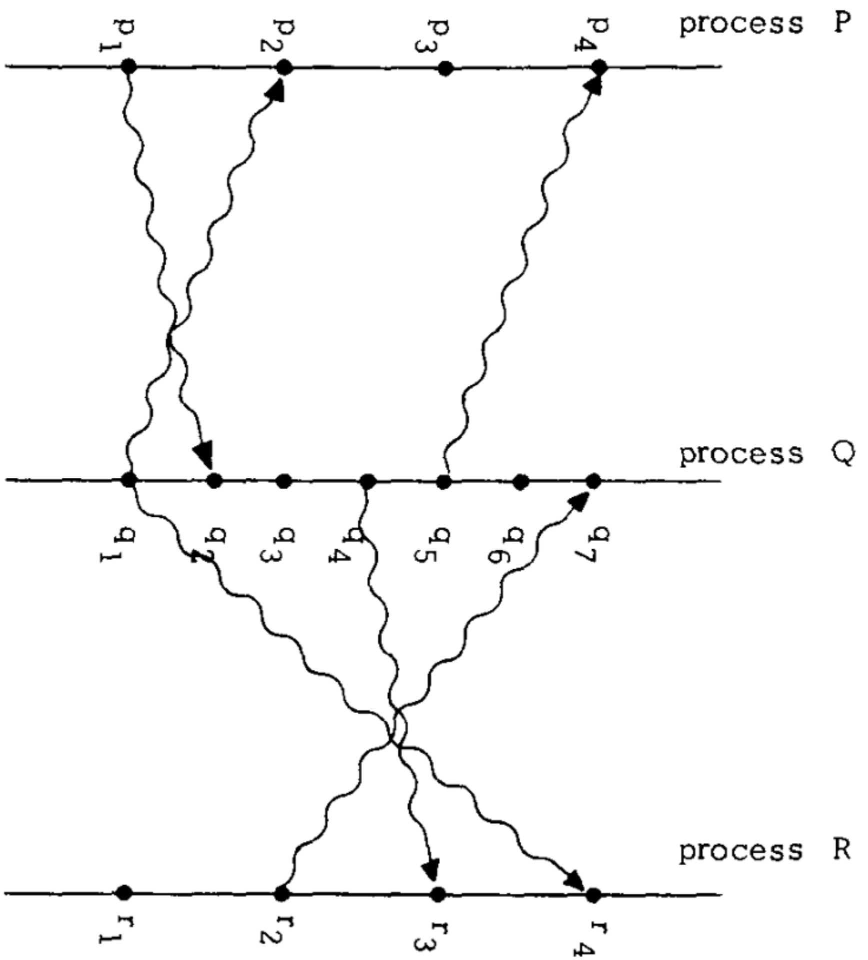
multiple înregistrări pe Youtube scurte:

[1](#) [2](#) [3](#) [4](#) [5](#)

- Și un interviu lung în două părți



Lamport – ceasuri logice, ordonare parțială



Lamport – ceasuri logice, ordonare parțială

IR1. Each process P_i increments C_i between any two successive events.

IR2. (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i\langle a \rangle$. (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

Clock Condition. For any events a, b:

$$\text{if } a \rightarrow b \text{ then } C\langle a \rangle < C\langle b \rangle.$$

Invers NU este adevărat

Lamport – ordonare totală ⇒

- Putem să definim o relație deordine totală folosindu-ne de id-ul proceslui.
- Dacă două evenimente au același C atunci le comparăm și dupa id-ul procesului.

Lamport – mutex distribuit

1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.

2. When process P_j receives the message T_m:P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.[5]

3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.

Lamport – mutex distribuit

4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.

5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m:P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $\Rightarrow$. (To define the relation "$\Rightarrow$" for messages, we identify a message with the event of sending it.) (ii) P_i has received a message from every other process timestamped later than T_m.[6]

Vector Clocks

Timestamps in Message-Passing Systems That Preserve the Partial Ordering

Colin J. Fidge
Department of Computer Science, Australian National University, Canberra, ACT.

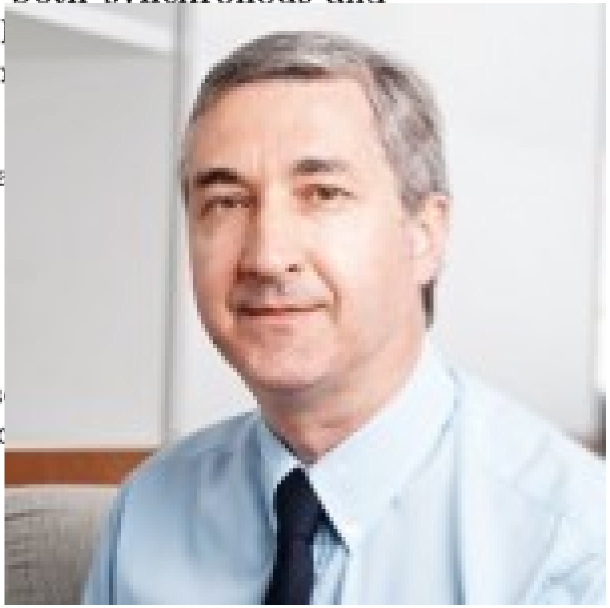
ABSTRACT

Timestamping is a common method of totally ordering events in concurrent programs. However, for applications requiring access to the global state, a total ordering is inappropriate. This paper presents algorithms for timestamping events in both synchronous and asynchronous message-passing programs that allow for access to t
herent in a parallel system. The algorithms do not change the con
require a central timestamp issuing authority.

Keywords and phrases: concurrent programming, message-passing, timesta
CR categories: D.1.3

INTRODUCTION

A fundamental problem in concurrent programming is determining the
different processes occurred. An obvious solution is to attach a number repres
a permanent record of the execution of each event. This assumes that each proc
clock, but practical parallel systems, by their very nature, make it difficult to



Vector Clocks

Virtual Time and Global States of Distributed Systems *

Friedemann Mattern †

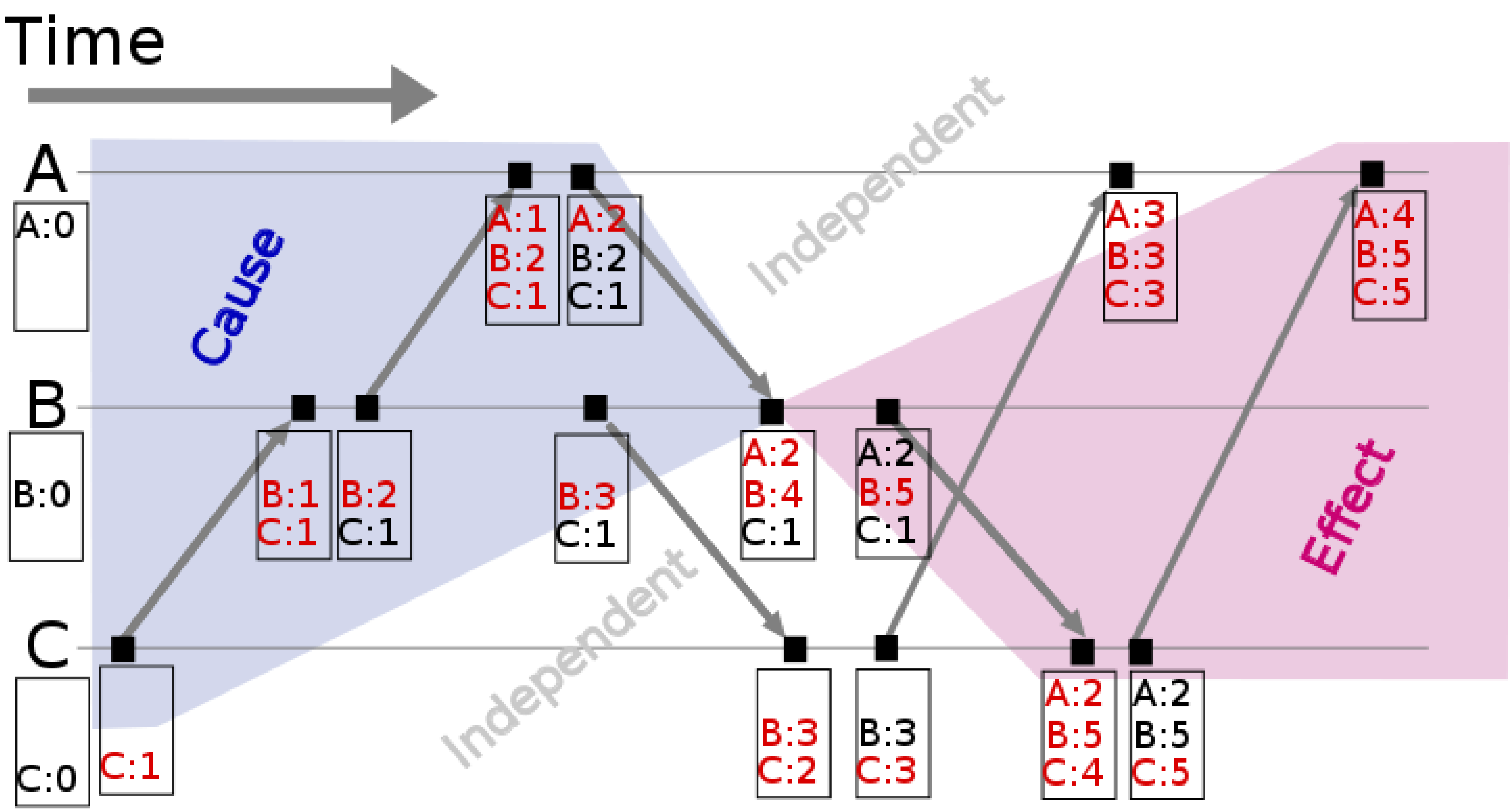
Department of Computer Science, University of Kaiserslautem
D 6750 Kaiserslautern, Germany

Abstract

A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. However, the notion of time is an important concept in every day life of our decentralized "real world" and helps to solve problems like getting a consistent population census or determining the potential causality between events. We argue that a linearly ordered structure of time is not (always) adequate for distributed systems and propose a generalized non-standard model of time which consists of vectors

view of an idealized
access to all process
The fact that *a*
view of the global s
not exist is the cau
tributed systems.
and database syste
detection, and conc
cult to solve in a dis
sical centralized env
of distributed contr
found to be wrong.

Vector Clocks



Problema Generalilor Bizantini

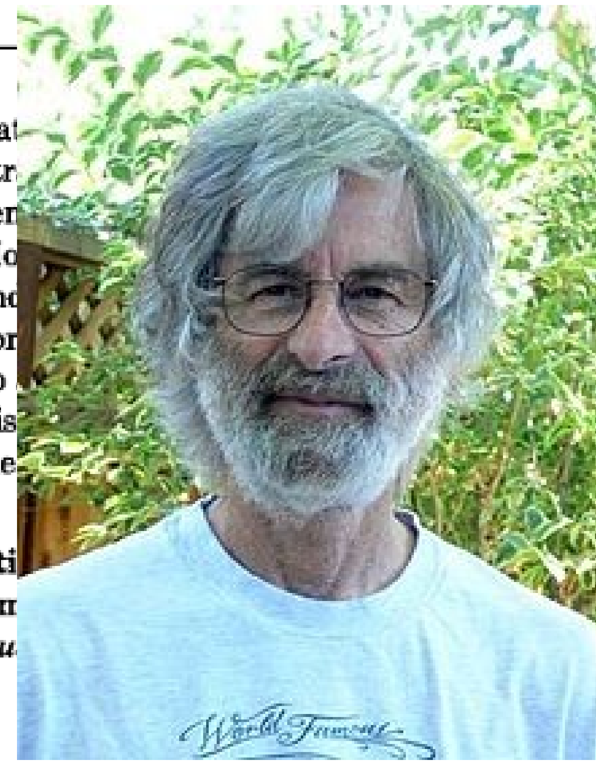
The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that
to different parts of the system. This situation can be expressed abstr
generals of the Byzantine army camped with their troops around an ener
by messenger, the generals must agree upon a common battle plan. Ho
may be traitors who will try to confuse the others. The problem is to finc
the loyal generals will reach agreement. It is shown that, using only or
solvable if and only if more than two-thirds of the generals are loyal; so
two loyal generals. With unforgeable written messages, the problem is
generals and possible traitors. Applications of the solutions to reliable
discussed.

Categories and Subject Descriptors: C.2.4. [**Computer-Communicati**
Systems—*network operating systems*; D.4.4 [**Operating Systems**]: Con
network communication; D.4.5 [**Operating Systems**]: Reliability—*fau*

General Terms: Algorithms, Reliability



Soluție Generali Bizantini – fără semnături

- Există doar dacă mai mult de două treimi din generali sunt onești.
  - **NU** există soluție pentru 3 sau mai puțini generali.
- Necesită multă comunicație, mesajele de la un general sunt transmise la toți ceilalți într-o formă recursivă.

Soluție Generali Bizantini – cu semnătură

(a) A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
(b) Anyone can verify the authenticity of a general's signature.

Funcționează dacă sunt minim doi generali corecți.
Înseamnă că există soluție pentru 3 generali.

Our algorithm assumes a function *choice* which is applied to a set of orders to obtain a single one. The only requirements we make for this function are

1. If the set V consists of the single element v, then $choice(V) = v$.
2. $choice(\varnothing)$ = RETREAT, where $\varnothing$ is the empty set.

Soluție Generali Bizantini – cu semnătură

Algorithm SM(m).

Initially $V_i = \emptyset$.

(1) The commander signs and sends his value to every lieutenant.
(2) For each i:

(A) If Lieutenant i receives a message of the form $v:0$ from the commander and he has not yet received any order, then

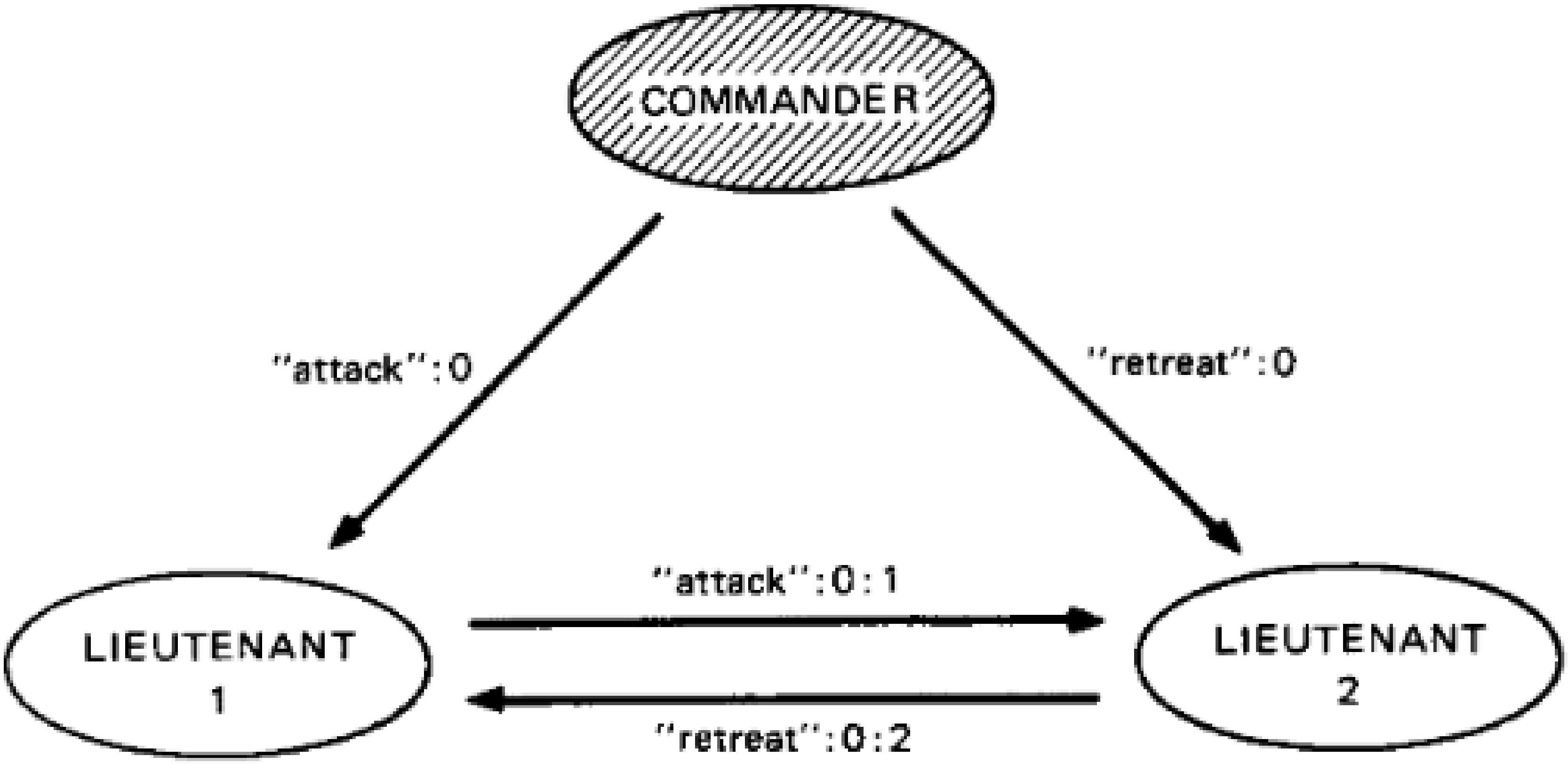
(i) he lets V_i equal $\{v\}$;
(ii) he sends the message $v:0:i$ to every other lieutenant.

(B) If Lieutenant i receives a message of the form $v:0:j_1:\cdots:j_k$ and v is not in the set V_i, then

(i) he adds v to V_i;
(ii) if $k < m$, then he sends the message $v:0:j_1:\cdots:j_k:i$ to every lieutenant other than $j_1, \ldots, j_k$.

(3) For each i: When Lieutenant i will receive no more messages, he obeys the order $choice(V_i)$.

Soluție Generali Bizantini – cu semnătură



Consens rezistent la Faults - Paxos

The Part-Time Parliament

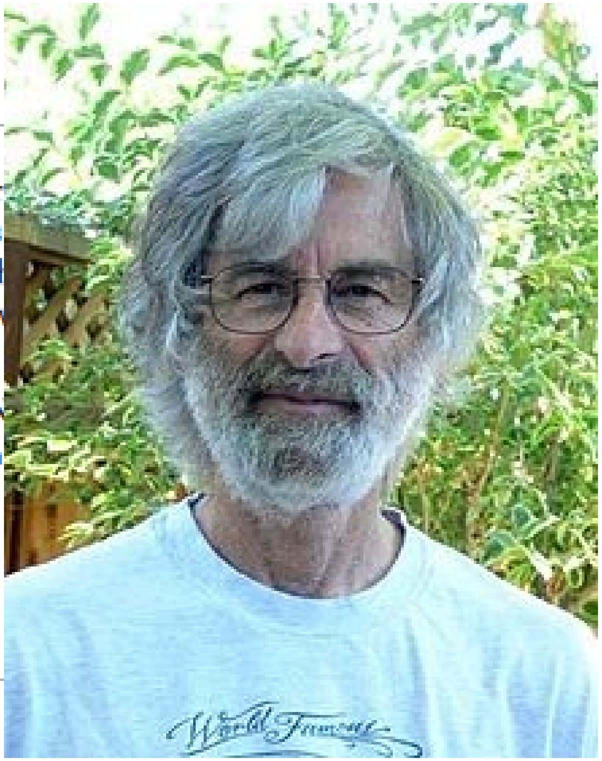
LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the par
spite the peripatetic propensity of its part-time legislators. The legislators
copies of the parliamentary record, despite their frequent forays from the ch
fulness of their messengers. The Paxon parliament's protocol provides a nev
the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications N**
Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Relia
J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit,



Paxos

Phase 1. (a) A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors.

(b) If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Paxos

Phase 2. (a) If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.

Paxos

2.3 Learning a Chosen Value

To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors. The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal. This allows learners to find out about a chosen value as soon as possible, but it requires each acceptor to respond to each learner—a number of responses equal to the product of the number of acceptors and the number of learners.

Paxos

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |<---------X--X--X       |  |  Promise(1,{Va,Vb,Vc})
   |         X--------->|->|->|       |  |  Accept!(1,V)
   |         |<---------X--X--X------>|->|  Accepted(1,V)
   |<---------------------------------X--X  Response
   |         |          |  |  |       |  |
```

Paxos

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |          |  |  !       |  |  !! FAIL !!
   |         |<---------X--X          |  |  Promise(1,{Va, Vb, null})
   |         X--------->|->|          |  |  Accept!(1,V)
   |         |<---------X--X--------->|->|  Accepted(1,V)
   |<---------------------------------X--X  Response
   |         |          |  |          |  |
```