**Distributed Systems**
Focus on: Collaborative Platforms, Shared Memory
Distributed Algorithms

**Outline**
I. Computer Supported Collaborative Work
II. Shared Memory Theory
III. Distributed Algorithms for Shared Memory Management
IV. Validations of Distributed Algorithms
V. Implementations on Message Passing Layer
VI. Computer Science Advances in these Domains

**Introduction**

- **CSCW**
- **Shared Memory**
- **Distributed Algorithms**
- **Validation**
- **Message Passing**
- **Advances**
- **Conclusion**

**Collaboration**

- Internet: a new powerful communication vector
- Improvement of performance
  - Processors level (compression treatments...)
  - Networks levels (flow rates, Quality of Services, ...)
- Different domains
  - Remote teaching: teleteaching
  - Tele-Medicine
  - Collaborative editing
  - videoconferencing...

**What are the specificities of this kind of applications?**

- Are images the only one medium?
- Is the text used?
  - Discrete Media
- Does this application use audio?
  - And video?
  - Continuous Media

Two types (groups) of media are distinguished

**Collaboration Domain**

- Besançon Hospital (France)
- Neuroradiology of Lausanne (Switzerland)
- Staff of experts in Geneva (Switzerland)
Before the emergence of the concept of CSCW, we, computer scientists, thought that our developments must manage collaborative work transparently for users.

But, during a collaboration phase (face-to-face or using computer) we need to see what the other users are doing. Or, at least, we need to feel what the other users are doing.

It is the awareness.

For example: during this talk, I can see on your faces if you are interested. If my talk was broadcast using remote teaching platform, I would not be able to know that.

New awareness tools are needed.

Collaborative Work is a part of CSCW

CSCW: a pluridisciplinary domain

Psychology, Linguistic, Sociology, Ethnology...

Groupware

Network, Artificial Intelligence, Distributed systems...

CSCW

Other point of view

From Cooperation

Only Sharing

Interactions

To Collaboration

Outline

I. Computer Supported Collaborative Work
- CSCW Domain,
- Distributed HCI,
- Example of Applications.

II. Shared Memory Theory

III. Distributed Algorithms for Shared Memory Management

IV. Validations of Distributed Algorithms

V. Implementations on Message Passing Layer

VI. Computer Science Advances in these Domains

Centralized Architecture

Only one process manages the consistency of data and actions in the globality of the system.

Advantages:
- Simplicity of implementation for the synchronization and concurrency problems.

Disadvantages:
- The response time is increased, and fault tolerance can not be processed.
Replicated Architecture

One process is associated to each user. Data are replicated on each site (processor) and the consistency will be maintained using communications between sites.

Advantages:
- speed due to local accesses, and fault tolerance due to the data redundancy involved by replications.

Disadvantages:
- difficulty of implementation including data consistency and scheduling of actions.

Hybrid Architecture

A centralized process is in charge of data consistency, and one process per user manages users actions on interface.

Advantages:
- The simplicity and the partial resolution of problems begotten by the centralized management.

Disadvantages:
- difficulty to implement the fault tolerance.

Outline

I. Computer Supported Collaborative Work
- CSCW Domain,
- Distributed Architecture,
- Example of Applications.

II. Shared Memory Theory
- Shared Memory Paradigm,
- Shared Memory Models,

III. Distributed Algorithms for Shared Memory Management
IV. Validations of Distributed Algorithms
V. Implementations on Message Passing Layer
VI. Computer Science Advances in these Domains

Groupware Interfaces

Collaborative spaces model: From 3-leaf clover to 4-leaf and finally 5-leaf clover.

Memory Space Sharing

- A collaborative work
- Shared context between collaborative members
- Message-passing: very heavy to manage, very costly.
- Distributed shared memory: transparent data sharing

?? Replicated Space or Distributed Space ??
Introduction

CSCW
Shared Memory
Distributed Algorithms
Validation
Message Passing
Advances
Conclusion

**Distributed Space Vs Replicated Space**

- **Distributed Space**: An aggregate shared address space.
- **Replicated Space**: Various occurrences of the same space.

**Data consistency**

- Several copies of a same shared memory object (several occurrences of a same shared memory object) are present in the system, but:
  - What is the most recent?
  - What is the most consistent?

**Consistency problem example**

- **Collaborative work (Telediagnostic)**: Consensus
  - **Consensus** (Thresholding + Edge Detection)

**Formalism**

- **$R_p(o)v$**: Reading by a processor $P$ of an object $o$ which return $v$ value.
- **$W_p(o)v$**: Writing by a processor $P$ of a value on an object $o$.

**Consistency: a problem of writing order**

- **On Processor 1**
  - Execution History
  - $R_p(x)1$
  - $W_p(x)2$
- **On the remote Processor 2**
  - Thresholding
  - Edge Detection + Thresholding
Introduction

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Advances

Conclusion

Outline

I. Computer Supported Collaborative Work

II. Shared Memory Theory

• Shared Memory Paradigm

III. Distributed Algorithms for Shared Memory Management

IV. Validations of Distributed Algorithms

V. Implementations on Message Passing Layer

VI. Computer Science Advances in these Domains

Definition using constraints

• Some shared memory need less constraints, to have a management more flexible.

• Weaken the consistency allows better performance in execution

• The different types of consistency are classified following to the constraints: time, order...

Consistency types

• Non-synchronized
  • 5 types,
  • 3 types really in used in applications,
  • 2 theoretical types.

• Synchronized

The Constraints

• C1: The operations are observable simultaneously on all processors.

• C2: The global observed order of the operations is the same on each processor. All processors observe the same sequence but not on the same time.

• C3: The only one execution order, which is respected, is the causality (read / write).

• C4: The execution order is only respected for operations made on the same processor.

• C5: The execution order is only respected for operations made on the same processor and on the same object.

In these models all operations can be observed on all processors.
The 5 non-synchronized models

Atomic
Sequential
Causal
PRAM
Slow memory

Theoretical models

Atomic Consistency

This history respects the atomic consistency

Atomic consistency is respected within δt

In this model it is not possible to observe this sequence: Rp3(y)2 after Wp1(x)3

Sequential Consistency

The C1 constraint is no more respected:

C1: The operations are observable simultaneously on all processors.

C2: The global observed order of the operations is the same on each processor. All processors observe the same sequence but not on the same time.

Correction

This history does not respect the atomic consistency

Now, your job: identify and justify histories that respect (or not) the atomic consistency
Introduction

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Advances

Conclusion

Sequential Consistency

This history respects the sequential consistency

The operations sequence is:

\[ R_{p1}(x1) \succ R_{p1}(x2) \succ R_{p1}(x3) \]

Sequential Consistency

This history does not respect the sequential consistency

Because operations sequences are different on P1 and on P2:

\[ R_{p1}(x1) \succ R_{p2}(x1) \succ R_{p3}(x1) \]

Sequential Consistency

Now, your Job: identify and justify histories that respect (or not) the sequential consistency

Causal Consistency

The C2 constraint is no more respected:

- C2: The global observed order of the operations is the same on each processor. All processors observe the same sequence but not on the same time.
- The only one execution order, which is respected, is the causality (read / write).

Correction

These constraints have to be respected:

\[ R_{p1}(x1) \succ R_{p2}(x2) \succ R_{p3}(x3) \]
Now, your Job: to draw all dependencies

Non-exhaustive list of dependencies

Causal Consistency (Theoretical models)

PRAM Consistency (Theoretical models)
The C4 constraint is no more respected:
- C4: The execution order is only respected for operations made on the same processor.
- The execution order is only respected for operations made on the same processor and on the same object.

### Slow Memory Consistency

This history respects the slow memory consistency.

On P2 (as on P3) the reading of value 1 on x proceeds the reading of value 3 on x, so the P1’s writings on x are respected by these readings: $W_p1(x)1 \succ W_p1(x)3$

This history respects the slow memory consistency.

On P2 after $S$, the returned value on an x reading can only be 3.

On P3 there is no synchronization, so there is no constraint on readings for P3.

### Synchronized Consistency Models

In these first five models, all writing operations have to be observable by all processors.
- In these last two models, all operations are not necessary observable. Only operations, which are synchronized, are observable.
- Two models are exposed:
  - The weak ordering
  - The release consistency

### Weak Ordering

- A new operator is defined $S$, used with $R_p(o)$ and $W_p(o)$.
- Remark: it is possible to defined $S$ for each object:
  $S_x$, $S_y$
**Weak Ordering**

- This history does not respect the weak ordering consistency:
  - On P₁, after S, the returned value on an x reading cannot be 1.

**Release consistency**

- It is a refinement of the weak ordering model.
- Indeed in the weak ordering model, during a synchronization phase it is not possible to determine if a processor is writing or is reading.
- Acq (Acquire) operation reports the critical section entry.
- Rel (Release) operation reports the critical section exit.

**Correction**

- These models are theoretical ones, but what are the algorithms that really implement these models?
- Until now, the course has defined only the theory of consistency.
- Now, we need to study the real protocols that allows to respect these models.

**Now, your Job: Identify and justify histories that respect (or not) the release consistency**

- On P₂ after Acq, the returned value on an x reading can only be 3.
- On P₃, there is no synchronization, so there is no constraint on readings for P₃.

**Introduction**

- CSCW
- Shared Memory
- Distributed Algorithms
- Validation
- Message Passing
- Advances
- Conclusion
Outline

I. Computer Supported Collaborative Work
II. Shared Memory Theory
   • Distributed Algorithms for Shared Memory Management
     • Invalidation Protocol,
     • Pilgrim Protocol,
     • Shared Memory guaranteed Models.
III. Distributed Algorithms for Shared Memory Management
     • Invalidation Protocol,
     • Pilgrim Protocol,
     • Shared Memory guaranteed Models.
IV. Validations of Distributed Algorithms
V. Implementations on Message Passing Layer
VI. Computer Science Advances in these Domains

Introduction

Broadcast of Information

- To guarantee that messages arrive ordered, without loss and duplication
- To facilitate the management of shared data consistency

3 types of protocols exist:

- Asymmetric Protocol
- Symmetric Protocol
- Turning Coordinator Protocol

Invalidation Protocol Specifications

- Replicated memory: an object is stored on each processor which uses it.
- The grain of the application is the object.
- This protocol uses the technique of invalidation on writing.
- From an object point of view, this protocol is a simpleWriter/multipleReaders one.
- From the memory point of view, this protocol is a multipleWriters/multipleReaders one.

The owner of an object is the only one which can write on it.

For each object

Invalidation Protocol

This implementation was made using a message passing platform. On each server a daemon is running.

- Writing
  - Then, invalidation broadcasting.
- The non-owners
  - request to recover the ownership,
  - Then, ownership change
    - Writing
    - Then, invalidation broadcasting.
- The non-owners
  - Local reading if valid, if not recover remote value (in the same time the local flag value becomes 1).

Writer

Reader

After each writing an invalidation must be broadcast
Outline

I. Computer Supported Collaborative Work
II. Shared Memory Theory
   a) Replicated Algorithms for Shared Memory Management
   i. Invalidation Protocol
   ii. Shared Memory guaranteed Models
III. Distributed Algorithms for Shared Memory Management
IV. Validations of Distributed Algorithms
V. Implementations on Message Passing Layer
VI. Computer Science Advances in these Domains

The Token or Pilgrim

- Node 1: No object on node 1, objects list and garbage list
- Node 2: Objects broadcasts
- Node 3: Messages for objects ownership management

Structure of an object:
- Command: the command can have one value among the following: null, destroy objects, Change object ownership, Invalidation to earn object ownership, Accept to give up object ownership
- Parameter: the parameter is optional, for example it indicates the number of the asking site
- Rank: the rank is the identifier of the object o_i
- Data: n octets, Data size depends on the cooperative application developed
Introduction

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Advances

Conclusion

Illustration of the Pilgrim Execution

Do you remember: The Pilgrim algorithm?
Together we will think about the automata of an object ownership (single writer, multiple readers)

Writer: The owner
  + writing

  The non-owners
  + request to recover the ownership,
  + ownership change
  + writing

Reader: The owner
  + Local reading.

  The non-owners
  + Local reading

Do you remember: The Pilgrim algorithm?
Together we will think about the automata of an object ownership (simple writer, multiple readers)

Outline

I. Computer Supported Collaborative Work

II. Shared Memory Theory

- Invalidation Algorithms for Shared Memory Management
  - Invalidation Protocol,
  - Pilgrim Protocol,

- Shared Memory Guarantee Models

III. Distributed Algorithms for Shared Memory Management

- Model checking,
- Petri Net,
- Finite State Automata.

IV. Validations of Distributed Algorithms

V. Implementations on Message Passing Layer

VI. Computer Science Advances in these Domains

Your Job: to Find the Consistency Model Allowed by the Use of the Pilgrim Protocol

Your Job: to Find the Consistency Model Allowed by the Use of the Invalidation Protocol

Outline

I. Computer Supported Collaborative Work

II. Shared Memory Theory

III. Distributed Algorithms for Shared Memory Management

- Petri Net,
- Finite State Automata

IV. Validations of Distributed Algorithms

V. Implementations on Message Passing Layer

VI. Computer Science Advances in these Domains

Consistency Model Allowed by the Use of the Invalidation Protocol
Introduction
Distributed Algorithms
Message Passing
Advancements
Conclusions

Validation with model checking tools (average 40 existing MCTs)  

The most famous SPIN overview
- open-source software tool (http://spinroot.com),
- freely available since 1991,
- one of the most prominent tools for formal verification of distributed software systems,
- developed by Gerald Holzmann at Bell Labs (beginning in 1980),
- awarded the prestigious System Software Award 2001 by the ACM,
- Primer and Reference Manual [Hol03].

To use SPIN, you need to implement your distributed software systems,  

ProMeLa (Process Meta Language) → the name SPIN stands for Simple ProMeLa Language

Conclusion
Advances
Message
Validation
Distributed
Shared
CSCW

# SPIN

Channel declarations:  
\[ \text{chan ChannelName} = \{ \text{mtype} \} \text{of \{mtype,byte,byte,byte,int,byte,byte,byte,int,byte,byte,byte,int \}} \]

Channel name: name of the channel,  
capacity: capacity of the FIFO channel (0 in Pilgrim),  
\(|m| \leq k-1\): type of transmittable data (tuples),  
Example: \(\{mtype,byte,byte,byte,int,byte,byte,byte,int,byte,byte,byte,int \}}\)

Mtype is the Label: example mtype = \{pilgrim,ackn,premier\}

Communication:  
synchronous message passing → capacity 0,  
 asynchronous message passing → capacity k-1,  

Communication actions:  
sending: ChannelName expr1, expr2, . . . , exprk;  
Example: \(\text{token[node1]}!pilgrim(c1,f1,p1,d1,c2,f2,p2,d2,c3,f3,p3,d3)\)  
receiving: ChannelName? x1, x2, . . . , xk;  
Example: \(\text{ctrl[1]}?\text{ctrl}[1]\)  

Communication:  
non deterministic decisions  
Example:  
\(\text{IF (g) THEN statement1 ELSE statement 2 FI}\)  

SPIN: Classic Presented Case  

If several parts are kept activable simultaneously when each of them has an equal probability chance being selected: non deterministic decisions. This means that during checking all solutions will be compared.

# SPIN: Extracts of Pilgrim ProMeLa Implementation

```
#define N 100
#define R 102
#define A 103
#define Q 104
#define FALSE 0
#define PROCESSEUR 3
```

```
chan token[PROCESSEUR] = [0] of {mtype,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte}
chan ack[PROCESSEUR] = [0] of {mtype};
```

```
proctype receiver(){
  proctype transmitter(){
    /* Creation of SUCCESS and ERROR lists */
    do :: (\text{true}) \to \text{assert! control,info} \to \text{printf("received control message: \text{\%d}, info")}
    :: (\text{true}) \to \text{assert! data,info} \to \text{printf("received data message: \text{\%d}, info")}
    od
  }  
}
```

```
atomic { /* execution of an indivisible sequence */
  /* Parallel processes instantiation */
  run transmitter(); run receiver();
}
```

```
byte info = 1
```

```
:: (\text{true}) -> \text{assert! control,info} \to \text{printf("received control message: \text{\%d}, info")}
:: (\text{true}) -> \text{assert! data,info} \to \text{printf("received data message: \text{\%d}, info")}
```

```
run receiver();
```

```
run transmitter();
```

```
 Chan token[PROCESSEUR] = [0] of {mtype,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte,byte}
 chan ack[PROCESSEUR] = [0] of {mtype};
```
Introduction

CSCW

Shared Memory

Distributed Algorithms

Message Passing

Advances

Conclusion

**SPIN:** Extracts of Pilgrim ProMeLa Implementation

```plaintext
int token[AnneauPred[s]]

if (msg == pilgrim) -> ack[AnneauPred[s]]!ackn;
/* sending of Ack */

if (s == p1) 
  do 
    if (TRUE) 
      break;
    if (TRUE) 
      break;
  od;

if (s == p2) 
  do 
    if (TRUE) 
      break;
    if (TRUE) 
      break; 
  od;
```

/* pilgrim sending and ack waiting */

token[AnneauSucc[s]]! pilgrim(c1,f1,p1,d1,c2,f2,p2,d2,c3,f3,p3,d3);

ack[s]? Ackn;

```plaintext
::(msg==premier) ->token[AnneauSucc[s]]!pilgrim(c1,f1,p1,d1,c2,f2,p2,d2,c3,f3,p3,d3);
ack[s]?ackn;
```

XSPIN Visualization (token with 3 nodes)

XSPIN Visualization (token with 4 nodes)

Execution Results on XSPIN

After one night of calculation...

- With a depth of 9999 in the execution tree
- No error were found

SMV from Carnegie Mellon University...

Model Checking @CMU

Cadence SMV is a symbolic model checking tool that allows you to formally verify temporal logic properties of finite state systems, such as computer hardware designs

**Outline**

I. Computer Supported Collaborative Work

II. Shared Memory Theory

III. Distributed Algorithms for Shared Memory Management

  - Model checking
  - Petri Net
  - Finite State Automata

IV. Implementations on Message Passing Layer

V. Computer Science Advances in these Domains
A Petri Net Specification

- PN consists of three types of components: places (circles), transitions (rectangles) and arcs (arrows).
- Places represent possible states of the system.
- Transitions are events or actions which cause the change of states.
- Every arc simply connects a place with a transition or a transition with a place.
- Tokens are dots (or integers) associated with places; a place containing tokens indicates that the corresponding condition holds.

Places represent possible states of the system;
Transitions are events or actions which cause the change of states;
Every arc simply connects a place with a transition or a transition with a place;
Tokens are dots (or integers) associated with places; a place containing tokens indicates that the corresponding condition holds.

A Change of State in Petri Net: the Firing of a transition

- It is denoted by a movement of token(s) (yellow dots) from place(s) to place(s), and is caused by the firing of a transition.
- The firing represents an occurrence of the event or an action taken.
- The firing is subject to the input conditions, denoted by token availability.

Several Distributed Algorithms Phases Modelized by Petri Net.

- Parallelism
- Synchronization for Meeting (e.g., barrier)
- Synchronization by Semaphore
- Resources Sharing
- Memorization
- Capacity Limitation

Concurrency

Independent inputs permit “concurrent” firing of transitions.
**Conflict**

Overlapping (sharing) inputs put transitions in conflict

Only one of a or b may fire

**Mutual Exclusion**

The two subnets are forced to synchronize

**Introduction**

CSCW

Shared Memory

Distributed Algorithms

Message Passing

Advantages

Conclusion

**Mutual Exclusion**

Your job: The two subnets are forced to synchronize with equity

**Mutual Exclusion**

Another solution: student’s one

**Mutual Exclusion**

Another solution: a second student’s one

**Your Job:** to Create the Petri Net (PN) for the Following Execution

Fork one place to 3 places, work parallel, and then join 3 places to one.
Your Job: to Create the Petri Net (PN) for the Following Execution
A Cycle of Producer with Buffer (also called WaitingPlace: Place with 3 possible tokens) and a Cycle of Consumer

(matrix gather with 3 treatment lines)

Animation
A token ring with 3 nodes, on each node a task to realize on each round

Possible Declinations of Petri Net
- Inhibitor arcs:
  - Inhibitor arcs are represented with a circle-headed arc.
  - The transition can fire if the inhibitor place does not contain tokens.
- Weight & Capacity:
  - It is possible to define every place has a capacity and every arc has a weight.
- Timed Petri Net:
  - Time delays associated with transitions and/or places.
  - Fixed delays or interval delays.
  - Stochastic Petri net: exponentially distributed random variables as delays.
- Coloured Petri Net:
  - Tokens or places have "colours", holding complex information.
Introduction

A Token Ring with 3 Nodes
With Inhibition Arcs and Only One Active Node

Behavioural Properties: a Way to Prove the Protocols

- Reachability (atteignabilité)
  - “Can we reach one particular state from another?”

- Boundedness (est-il borné)
  - “Will a storage place overflow?”

- Liveness (vivacité)
  - “Will the system die in a particular state?”

- Softwares and tools: TINA (INRIA), PEP, HiQPN, Design/CPN

Softwares and tools: TINA (INRIA), PEP, HiQPN, Design/CPN

Behavioural Properties: a Way to Prove the Protocols

- Reachability (atteignabilité)
  - “Can we reach one particular state from another?”

- Boundedness (est-il borné)
  - “Will a storage place overflow?”

- Liveness (vivacité)
  - “Will the system die in a particular state?”

- Softwares and tools: TINA (INRIA), PEP, HiQPN, Design/CPN

Softwares and tools: TINA (INRIA), PEP, HiQPN, Design/CPN

A Token Ring with 3 Nodes
With Inhibition Arcs and Only One Active Node

In 1965, Dijkstra posed and solved a synchronisation problem called the dining philosophers problem. The problem can be stated as follows:

- Five philosophers are sitting around a table. Each philosopher has a plate of spaghetti.
- A philosopher needs two chopsticks to eat it. There is one chopstick between each plates.
- The life of a philosopher consists of alternate periods of eating and thinking.
- When a philosopher gets hungry, he tries to acquire his left and right chopsticks, one at a time.
- If successful in acquiring two chopsticks, a philosopher eats for a while, then puts down the chopsticks and continues to think.

Your Job to conclude this Part on Petri Net:

The five Philosophers Wellknown Problem

Your Job: to Add Equity Between Philosophers?

A Philosopher Can Eat for the Second Time Only if his Other Colleagues Have Already Ate

- The first modelization allows one philosopher to eat many times, and his neighbors cannot eat
- It should be interesting to propose a mecanism to allows to share fairly the possibility to eat.
Introduction

I. Computer Supported Collaborative Work
II. Shared Memory Theory
III. Distributed Algorithms for Shared Memory Management
   - Validation of Distributed Algorithms
     - Model checking
     - Petri Net
   - Message Passing
   - Advancements
IV. Implementations on Message Passing Layer
V. Computer Science Advances in these Domains

Outline

Other Possible Modelization of Distributed Protocols
- Finite state automata are classically used to modelize distributed protocols.
- These automata are used to prove these protocols.
- A finite state automaton is composed of states and transitions.

Behaviors Modelization
- To establish a system, it is first necessary to define the behavior of the actors:
  - First Automata: the behavior of the students,
  - Second Automata: the behavior of the teacher.

Student Behavior

Teacher Behavior
Introduction

Do you remember *The five Philosophers*?

**Your job, this time:** to design the automata of a philosopher behavior

---

**Philosopher Behavior**

State: Initial state: philosopher thinking

State: Philosopher eating

---

**Philosopher Behavior (one first simple solution)**

State: Initial state: philosopher thinking

State: Philosopher eating

---

**Philosopher Behavior (a solution with separate chopsticks)**

State: Initial state: philosopher thinking with one chopstick

State: Philosopher eating

---

**Philosopher Behavior**

State: Initial state: philosopher thinking

State: Philosopher eating

---

**Philosopher Behavior**

State: Initial state: philosopher thinking

State: Philosopher eating

---

**Philosopher Behavior**

State: Initial state: philosopher thinking

State: Philosopher eating

---

**Philosopher Behavior**

State: Initial state: philosopher thinking

State: Philosopher eating
Introduction
CSCW
Shared Memory
Distributed Algorithms
Languages
Message Passing
Advantages
Conclusion

One Solution

Do you remember The Pilgrim algorithm?
Together we will think about the automata of an object ownership (simple writer, multiple readers)

Let a whiteboard:
- It is shared by 4 nodes (4 processors)
- and it contains one object: a circle.
The owner of this object is the node 2.

Do you remember The Pilgrim algorithm?
Together we will think about the automata of an object ownership (simple writer, multiple readers)

Node 2
Node 4
Node 3
Node 1

Let a whiteboard:
- It is shared by 4 nodes (4 processors)
- and it contains one object: a circle.
The owner of this object is the node 2.

Two nodes try to become the owner of the circle

Adds his request on the token

Cannot add his request because one other request is already on the token

Adds his request on the token

Is not working, and accepts to lose the ownership

Becomes the new owner of the circle and accepts to lose the ownership on the token

A First Simple Finite State Automaton

Pilgrim’s Finite State Automaton

Statuses
- Owner
- Non-Owner

Transitions
- (reading) + (writing)
- (reading) + (writing) + (reading)
- (reading) + (writing) + (reading)
- (reading) + (writing) + (reading) + (reading)

Writer
- The owner
- writing
- The non-owners
- respect to recover the ownership,
- ownership change
- Writing

Reader
- The owner
- Local reading
- The non-owners
- Local reading

OLO: Owner which Loses the Ownership
AUO: Active Unlocked Owner
IO: Inactive Owner
OLO: Owner which Loses the Ownership
NO: Non-Owner
NOCA: Non-Owner which Could Ask for the Ownership
NOA: Non-Owner which Asks
NOW: Non-Owner which Wins the Ownership
This Automaton Was Used to Prove these Theorems

- **Mutual Exclusion Theorem:** The Pilgrim protocol guarantees mutual exclusion for writing on a shared object.
- **Liveliness Theorem:** An object always has an owner or has one after finite time.

### Pilgrim's Proof

For each state we also state the 10 rules defining the automaton, and especially the impossible events.

**Pilgrim's Proof: 10 rules**

- **Rule 1:** The (reading) event does not change the state of a site.
- **Rule 2:** A site can write on an object only if it is in one of the following states: ALO (Active Locked Owner), and ALO (Active Unlocked Owner) or OLO (Inactive Owner).
- **Rule 3:** A site can send an ownership request for an object only if it is in the NO (Non-Owner) state.
- **Rule 4:** At time t one site at most can be in the NOA state.
- **Rule 5:** A site in the NOA state cannot receive a message with an accept or refuse command.
- **Rule 6:** A site in the NOA state cannot receive a message with an accept or refuse command.
- **Rule 7:** A site in the NOA state cannot receive a message with an accept or refuse command.
- **Rule 8:** A site in the NOA state cannot receive a message with an accept or refuse command.
- **Rule 9:** A site in the NOA state cannot receive a message with an accept or refuse command.
- **Rule 10:** A site in the NOA state cannot receive a message with an accept or refuse command.

**Pilgrim's Proof: examples of States Justifications**

**NOE (Non-Owner which Asks the ownership State):**

Rule 5 states that a site in NOE cannot receive a message with an accept or refuse command simultaneously with the two events (rta) and (rtr).

**AUE (Active Unlocked Owner) State:**

Rules 2 and 3 show that in AUE the events (request) and (accept) are impossible. A (reading) event does not change the state of the state (rule 1).

**AOO (Active Locked Owner State):**

Rule 3 shows that in AOO the event (request) is impossible. A (reading) event does not change the state of the state (rule 1). If it receives a request from an owner, it changes to LLO (Locked Owner).

We call this state Active Locked Owner, because it is receiving a token with a request (event (rta)). It has to accept the request. Then, it changes to LLO when receiving a token with no command, or to change to LLO (Locked Owner).

**LLO (Locked Owner State):**

If the ownership is accepted, the state changes to NOW (Non-Owner which Wins the ownership), and will have to manage the owner change. Otherwise it will return to NO and make another request.
Pilgrim: Proof of Mutual Exclusion Theorem

The Pilgrim protocol guarantees mutual exclusion for writing on a shared object.

Lemmas 1 and 2:

Lemma 1: When a site in the OLO (Owner which Looses the Ownership) state changes to the ALO (Active Locked Owner) state, the site which was the owner changes to the NO (Non-Owner) state.

Lemma 2: When a site changes from the OLO (Owner which Looses the Ownership) state to the NO (Non-Owner) state, one and only one site becomes the owner of the object. As soon as the winner receives the token, it changes to ALO (Active Locked Owner).

Conclusion: The mutual exclusion theorem is true at time $t_0$ and remains true following various events which modify the system. Therefore, the Pilgrim protocol guarantees mutual exclusion for writing on a shared object.

Pilgrim: Proof of Liveliness Theorem

An object always has an owner or has one after time $t_0$.

Using the same approach, we have to show that the liveliness theorem is true at time $t_0$. Theorem is initially true, because the ALO site is the only one which can write on the object. We have to prove that this theorem still remains true for each event that modifies the system.

Conclusion: The liveliness theorem is true at time $t_0$ and remains true following various events which modify the system. Therefore, the Pilgrim protocol guarantees that an object always has an owner or has one after time $t_0$.

Outline

I. Computer Supported Collaborative Work

II. Shared Memory Theory

III. Distributed Algorithms for Shared Memory Management

IV. Validations of Distributed Algorithms

IV. Computer Science Advances in these Domains

PVM: Parallel Virtual Machine

Version 1: 1989-90 Oak Ridge National Laboratory (first a research project),
Version 2: 1991; more simple; more robust,
Version 3: 1993; more suitable for parallel architectures,
Version 3.4: Major extensions (introduction of communicative contexts and static process groups),

PVM provides an unified framework for developing parallel programs with the existing infrastructure. PVM enables a collection of heterogeneous computer systems as a single parallel virtual machine. PVM supports functional and data parallelism. A well defined library of PVM interface routines are used for programming.

Many processors virtual or not

PVM is used for the development of parallel programs or parallel algorithms:

- Several virtual processors can be simulated on only one physical processor.
- Or for real parallelism: each virtual processor is associated to a physical processor.
- Mixed solutions with more or less physical and logical processors.
19/05/2017

PVM: Parallel Matrix Algebra
Parallel Matrix Multiplication

Example
On Node A

\[
\begin{pmatrix}
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\begin{pmatrix}
2 & 3 \\
5 & 6 \\
8 & 9
\end{pmatrix}
= 
\begin{pmatrix}
? & ? \\
? & ? \\
? & ?
\end{pmatrix}
\]

PVM’s Specifications

- Simple (hiding technical issues from the users),
- Portability: under Linux, FreeBSD, SunOS, Mips, SG IRIS...,
- Heterogeneity: machines, architectures, OS, network
  - with the use of external Data Representation
- Users oriented,
- Use the available power of a set of machines: a cluster,
- Collaboration of a set of machines.
- PVM is composed of:
  - A PVMd daemon resides on all computers making the virtual machine,
  - A PVMd is owned by one user and has no interaction with daemons from another user,
  - A PVMd routes and controls messages,
  - The first running is the « master » and the other are « slaves »
  - A PVMd owns a table of hosts and tasks under his control,
  - A PVMd owns also queue of packets and a queue of messages,
  - When a daemon PVMd is running, the file « /tmp/pvmd.<uid> » is created.

Deamons: PVMd

- A PVMd daemon resides on all computers making the virtual machine,
- A PVMd is owned by one user and has no interaction with daemons from another user,
- A PVMd routes and controls messages,
- The first running is the « master » and the other are « slaves »
- A PVMd owns a table of hosts and tasks under his control,
- A PVMd owns also queue of packets and a queue of messages,
- When a daemon PVMd is running, the file « /tmp/pvmd.<uid> » is created.

Virtual Machine Creation

- A PVMd daemon resides on all computers making the virtual machine
- It has to be registered to participate to the Virtual Machine

Tasks Connections

- Each task is connected to the daemon running on the processor where it has been launched.
Introduction
CSCW
Shared Memory
Distributed Algorithms
Validation
Message Passing
Advances
Conclusion

Tasks Connections

- A connected task can create other task (with pvm_spawn)

Deamon 1
Deamon 2
Deamon 3

task1 & task2

Deamon 3

task3

Tasks Connections

A connected task can create other task (with pvm_spawn)

Deamon 1
Deamon 2
Deamon 3

task1 & task2

Deamon 3

task3

The configuration of the virtual Machine is defined in the hostfile:

- pvm3 hostfile
- pvm to run the console
- pvm> conf gives the configuration of the Virtual Machine

The configuration of the virtual Machine is defined in the hostfile:

- pvm3 hostfile
- pvm to run the console
- pvm> conf gives the configuration of the Virtual Machine

It Contains both parent and child code.

/* find out my task id number */
mytid = pvm_mytid();

This function must be called before any other call. This call must be positive number.

pvm_perror(argv[0]);

This call tells what goes wrong with the last call.

/* spawn the parent task */
myparent = pvm_parent();

Returns the Parent ID or else will return Error code

"PvmNoParent" says that the process is spawned by user and not by any other process.

Following code differenciate the parent of all childs.

/* find my parent's task id number */

if (myparent == PvmNoParent) {
    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    Following code differenciate the parent of all childs.

    /* find my parent's task id number */

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

        This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvmspawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* find out my task id number */
    mytid = pvm_mytid();

    This function must be called before any other call. This call must be positive number.

    pvm_perror(argv[0]);

    This call tells what goes wrong with the last call.

    /* spawn the parent task */
    myparent = pvm_parent();

    Returns the Parent ID or else will return Error code

    "PvmNoParent" says that the process is spawned by user and not by any other process.

    Following code differenciate the parent of all childs.

    /* find my parent's task id number */

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

        This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvmspawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* find out my task id number */
    mytid = pvm_mytid();

    This function must be called before any other call. This call must be positive number.

    pvm_perror(argv[0]);

    This call tells what goes wrong with the last call.

    /* spawn the parent task */
    myparent = pvm_parent();

    Returns the Parent ID or else will return Error code

    "PvmNoParent" says that the process is spawned by user and not by any other process.

    Following code differenciate the parent of all childs.

    /* find my parent's task id number */

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

        This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvmspawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* find out my task id number */
    mytid = pvm_mytid();

    This function must be called before any other call. This call must be positive number.

    pvm_perror(argv[0]);

    This call tells what goes wrong with the last call.

    /* spawn the parent task */
    myparent = pvm_parent();

    Returns the Parent ID or else will return Error code

    "PvmNoParent" says that the process is spawned by user and not by any other process.

    Following code differenciate the parent of all childs.

    /* find my parent's task id number */

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

        This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvmspawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* find out my task id number */
    mytid = pvm_mytid();

    This function must be called before any other call. This call must be positive number.

    pvm_perror(argv[0]);

    This call tells what goes wrong with the last call.

    /* spawn the parent task */
    myparent = pvm_parent();

    Returns the Parent ID or else will return Error code

    "PvmNoParent" says that the process is spawned by user and not by any other process.

    Following code differenciate the parent of all childs.

    /* find my parent's task id number */

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

        This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvmspawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* find out my task id number */
    mytid = pvm_mytid();

    This function must be called before any other call. This call must be positive number.

    pvm_perror(argv[0]);

    This call tells what goes wrong with the last call.

    /* spawn the parent task */
    myparent = pvm_parent();

    Returns the Parent ID or else will return Error code

    "PvmNoParent" says that the process is spawned by user and not by any other process.

    Following code differenciate the parent of all childs.

    /* find my parent's task id number */

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

        This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvmspawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* find out my task id number */
    mytid = pvm_mytid();

    This function must be called before any other call. This call must be positive number.

    pvm_perror(argv[0]);

    This call tells what goes wrong with the last call.

    /* spawn the parent task */
    myparent = pvm_parent();

    Returns the Parent ID or else will return Error code

    "PvmNoParent" says that the process is spawned by user and not by any other process.

    Following code differenciate the parent of all childs.

    /* find my parent's task id number */

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

        This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvm Spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.

    /* spawn the child tasks */
    info = pvmspawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, ntask, child);

    This is the code which spawns childs.
Following code is run by child.

```c
info = pvm_initsend(PvmDataDefault);
info = pvm_pkint(&mytid, 1, 1);
info = pvm_send(myparent, JOINTAG);
pvm_exit();
```

First line - For message to be sent first we need to create the buffer to send the data.
Second line – We are adding the child’s id to the message.
Third line sending the join call to the parent with the message attached.
Fourth line calls off the child from pvm and destroy its memory references.

Communications

- Communications are based on TCP and UDP/IP: UDP inside between daemons, possibility of TCP mode between tasks.
- Several communication modes are allowed:
  - send: non-blocking
  - send: task
  - recv: blocking
  - direct routing
  - multicasting and broadcasting

Messages Exchanges

- All connected tasks can communicate

Blocking Receipt

- `pvm_send`: when this function is called the task is blocked until the reception of the message.
- `pvm_recv`: when this function is called task is blocked until the reception of the message.
Non-Blocking Receipt
- `pvm_nrecv`: This function is non-blocking. You just make sure that the message is not used before the final reception.

Multicasting and Broadcasting
- `pvm_bcast`: Sends to all nodes in a group, whether that group is predefined.
- `Pvm_mcast`: Sends to all nodes with node numbers that appear in an predefined array. These functions are more efficient than successive sendings.

XPVM: a graphical interface for PVM

Outline
I. Computer Supported Collaborative Work
II. Shared Memory Theory
III. Distributed Algorithms for Shared Memory Management
IV. Validations of Distributed Algorithms
V. Computer Science Advances in these Domains
You will work on MPI during practical exercises with Violeta Feloa.

**Introduction**

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Advances

Conclusion

**PVM Versus MPI**

**PVM**

- A distributed operating system
- Portability
- Heterogeneity
- Handling communication failures

**MPI**

- A library for writing application program, not a distributed operating system
- Portability
- High Performance
- Heterogeneity
- Well-defined behavior

**What is PVM?**

PVM - Parallel Virtual Machine

- A software package that allows an heterogeneous collection of workstations (host pool) to run as a single high performance parallel machine (virtual).
- PVM, through its virtual machine provides a simple yet useful distributed operating system.
- It has daemons running on all computers making up the virtual machine.

**What is MPI?**

MPI - Message Passing Interface

- A fixed set of processes is created at program initialization, one process is created per processor: mpirun –np 5 program
- Each process knows its personal number (rank)
- Each process knows number of all processes
- Each process can communicate with other processes
- Process cannot create new processes (in MPI-1)

**What is Not Different?**

- Portability - source code written for one architecture can be copied to a second architecture, compiled and executed without modification (to some extent).
- Support MPMD (Multiple Program Multiple Data) programs as well as SPMD (Single Program Multiple Data).
- Interoperability - the ability of different implementations of the same specification to exchange messages.
- Heterogeneity (to some extent)

PVM & MPI are systems designed to provide users with libraries for writing portable, heterogeneous, MPMD programs.
Process control: PVM and MPI

- Ability to start and stop tasks, to find out which tasks are running, and possibly where they are running.
- PVM contains all of these capabilities - it can spawn/kill tasks dynamically.
- But MPI-1 has no defined method to start new task.
- But MPI-2 contains functions to start a group of tasks and to send a kill signal to a group of tasks.

Message Passing operations: MPI Vs PVM:
- PVM: Simple message passing
- MPI: Rich message support.

Conclusion: PVM Versus MPI
Each API Has Its Unique Strengths

- PVM:
  - No such abstraction
  - Rich message support
  - Supports large communication topologies
  - Some realizations do not interoperate across architectural boundaries
  - Performance axioms for.
Example of a the FLC Consensus Protocol

- Principal idea: a phase of leader election
  - Reinforces the choice made by G
  - Ensures the existence of one leader per round.

- est: The estimation sent of the decision value

Example of the FLC Consensus Protocol

- Execution with No Crashes, No Wrong Suspicions

Example of the FLC Consensus Protocol

- Execution with Crash of the Leader, No Wrong Suspicions

Example of the FLC Consensus Protocol

- Execution with No Crashes, Wrong Suspicions

Outline

I. Computer Supported Collaborative Work
II. Shared Memory Theory
III. Distributed Algorithms for Shared Memory Management
IV. Validations of Distributed Algorithms
V. Implementations on Message Passing Layer

Adaptability at the Protocol Level From the Pilgrim Algorithm to the Chameleon Algorithm
Introduction

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Advances

Conclusion

Outline

I. Computer Supported Collaborative Work

II. Shared Memory Theory

III. Distributed Algorithms for Shared Memory Management

IV. Validations of Distributed Algorithms

V. Implementations on Message Passing Layer

6. Computer Science Advances in these Domains

• Consensus,

• Adaptability,

• Active Networks,

• Components Platforms,

• Ontologies and Traceability

Network adaptation: the use of Active Network (intelligent networks)

3 possibilities to adapt the network

6. APPAT: an Adaptability Dedicated Platform

PA4
Introduction

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Conclusion

Outline

I. Computer Supported Collaborative Work

II. Shared Memory Theory

III. Distributed Algorithms for Shared Memory Management

IV. Validations of Distributed Algorithms

V. Implementations on Message Passing Layer

- Computer Science Advances in these Domain:
  - Consensus,
  - Adaptability,
  - Active Networks,
  - Components Platforms,
  - Ontologies and Traceability

- Example UbiCore: the functional core of the application

- Development based on components

- Generic tools

- New components can be plugged following the targeted application
Introduction

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Ontologies

Myocardial infarction

Cardiac infarction

heart attack

infarction

These terms represent the same pathology. They must be linked by the ontologies. Like that, everybody speaks about the same concept.

Ontologies: COOVADIS Platform (Vascular Diseases)

Our objectives:

- Providing information sharing between physicians and improving the diagnoses in DICOM.
- Traceability chain also includes the diagnosis using ontology technology, which is the result of physician interpretation of the initial study and further studies made by collaborative diagnosis communities.
- It is crucial that the professionals involved in collaborative diagnosis have total access to medical information and up-to-date reports in the traceability chain.

Ontologies for Traceability in Telemedicine

- In medical systems, the most important objective is to identify, classify, and protect the information using relational databases.
- It allows to track the activities of medical procedures and treatments made by different departments and different professionals.

3 Ontologies as Support Tool for Physicians

- Vascular Diseases Ontology
- Traceability of Diagnosis Ontology

These ontologies provide the required distributed information to the medical collaborative community for enabling a better patient monitoring and diagnosis support in COOVADIS.

An Ontology: What Does It Look Like

Vascular Diseases Ontology: Defined with PROTEGE Software

COOVADIS Global Architecture
Introduction

CSCW

Shared Memory

Distributed Algorithms

Validation

Message Passing

Advances

Conclusion

Ontologies for Tourism in Phuket

Transliterated Words

If a stranger hears: « Namtok Kathu »

Namtork Kathu Numtoc Katu Numtok Kratuu Narmtog Kathu

Ontologies for Tourism in Phuket

Transliterated Words

q

e-Health & Telemedicine is THE WAY of the future for medicine.

q In this context, in computer science, the main domain for CSCW is the « Distributed algorithm for collaborative work ».

It is a very large domain in which you find distributed algorithm, distributed systems, webservice, a touch of network, adaptability, ontologies...

Conclusion

An Open Door for Many Research Applicative Subjects

Questions ?