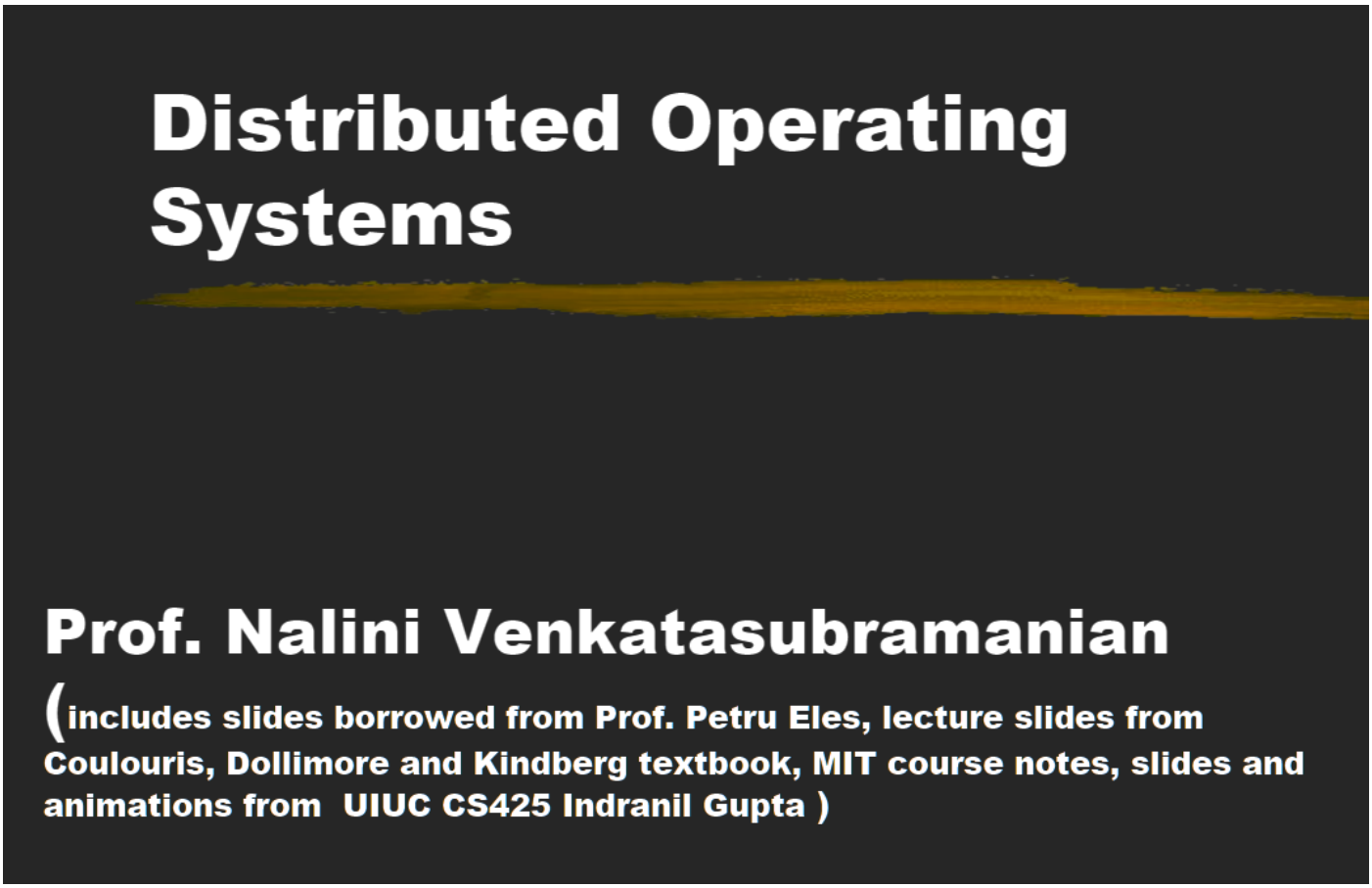


分散式作業 系統

教授。納里尼·文卡塔薩布拉馬尼安

包括從 Petru Eles 教授那裡借來的幻燈片，從 Coulouris、Dollimore 和 Kindberg

教科書、麻省理工學院課程筆記、幻燈片和 UIUC CS425 Indranil Gupta 動畫

OS 做了那些事？

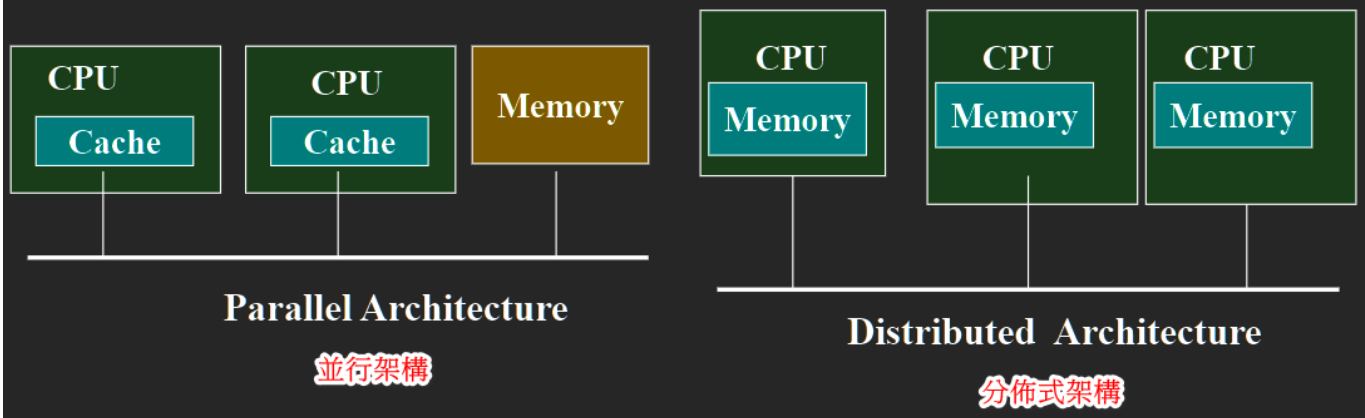
- 進程/線程管理
  - 排程
  - 通訊
  - 同步
- 內存管理
- 存儲管理
- 文件系統管理
- 保護和安全
- 連網

Distributed Operating Systems

Manages a collection of independent computers and makes them appear to the users of the system as if it were a single computer

Multiprocessors
Tightly coupled
Shared memory
多處理器
緊密耦合
共享內存

Multicomputers
Loosely coupled
Private memory
Autonomous
多電腦
鬆散耦合
私有內存
自主性



Parallel Architecture
並行架構

Distributed Architecture
分佈式架構

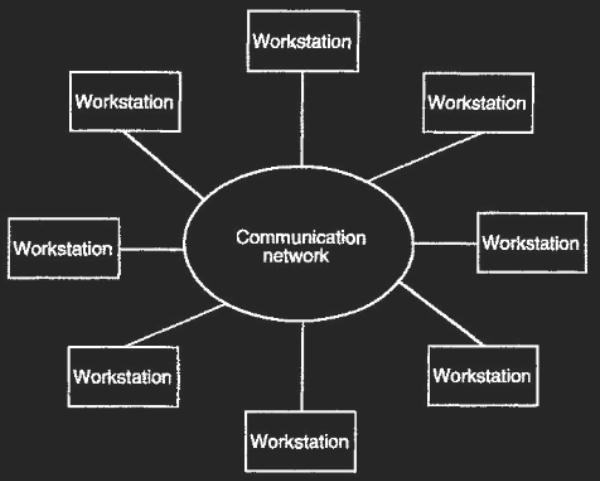
Early Systems - Workstation Model

早期系統 - 工作站模型

怎麼找控閒的工作站？
- 流程如何轉移：從一個工作站到其他？
- 當用戶登錄到空閒的工作站，遠端進程會發生什麼，現在不再變成空閒著嗎？
- 其他狀態：處理器pool，工作站Server...

示例：伯克利NOW項目、威斯康星大學CONDOR系統

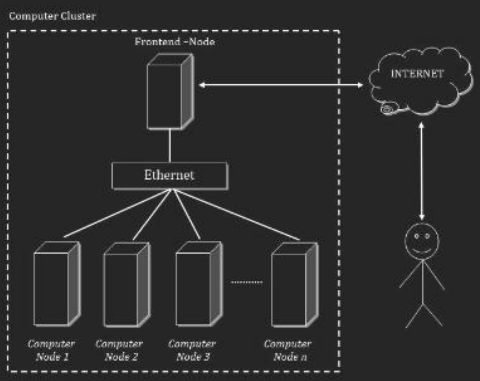
- How to find an idle workstation?
- How is a process transferred from one workstation to another?
- What happens to a remote process if a user logs onto a workstation that was idle, but is no longer idle now?
- Other models - processor pool, workstation server...

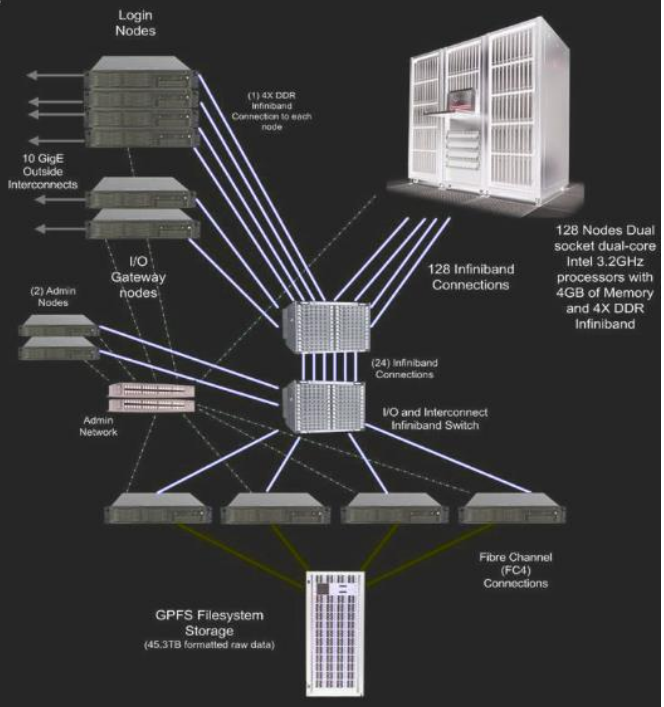


Examples: Berkeley NOW Project, U Wisconsin CONDOR system

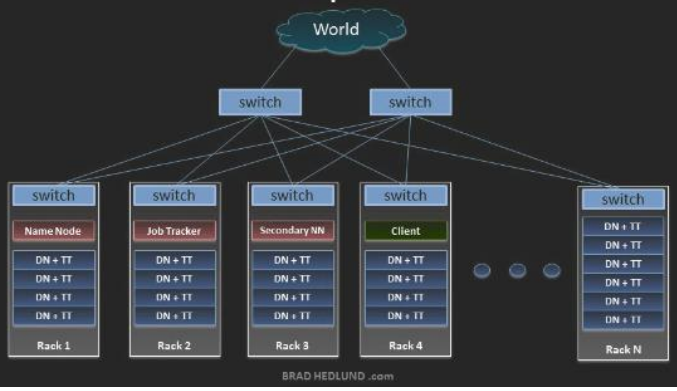
Cluster Computing

Loosely connected set of machines that behave as a unit





Hadoop Cluster



High Performance Computing (HPC) Cluster

Distributed Operating System (DOS) Types

分散式作業系統 (DOS) 類型

- Distributed OSs vary based on
  - System Image
  - Autonomy
  - Fault Tolerance Capability

  系統鏡像
  自主(自治)
  容錯能力
- Multiprocessor OS
  - Looks like a virtual uniprocessor, contains only one copy of the OS, communicates via shared memory, single run queue

  看起來像一個虛擬單處理器，只包含一個操作系統副本，通過共享內存，單運行隊列進行通信
- Network OS
  - Does not look like a virtual uniprocessor, contains n copies of the OS, communicates via shared files, n run queues

  看起來不像虛擬單處理器，包含操作系統的 n 個副本，通過共享文件進行通信，n 個運行隊列
- Distributed OS
  - Looks like a virtual uniprocessor (more or less), contains n copies of the OS/runtime, communicates via messages, n run queues

  看起來像一個虛擬單處理器（或多或少），包含 n 個操作系統/運行時，通過消息進行通信，n 個運行隊列

Design Issues (cont.) 設計問題

- *Transparency* 透明度(傳送) 位置傳送 進程、CPU 和其他設備、文件
  - Location transparency
    - processes, cpu's and other devices, files
  - Replication transparency (of files) 複製傳送 (文件)
  - Concurrency transparency
    - (user unaware of the existence of others)
  - Parallelism 並發傳送 (用戶不知道他人的存在)
    - User writes serial program, compiler and OS do the rest 並行性 用戶寫入串行程序、編譯器和作業系統休息
- *Performance*
  - Metrics
    - Throughput, response time
  - Load Balancing (static, dynamic) 效能 指標 吞吐量、響應時間 負載均衡 (靜態、動態的)
  - Communication is slow compared to computation speed
    - fine grain, coarse grain parallelism tradeoff 通訊緩慢與計算相比速度 細粒、粗粒並行權衡

Also : Scalability, Reliability, Flexibility, Heterogeneity, Security

另外：可擴展性、可靠性、靈活性、異構性、安全性

Design Elements 設計元素

- Process Management
  - Task Partitioning, allocation, synchronization, load balancing, migration 流程管理 任務分區、分配、同步、加載、平衡、遷移
- Communication
  - Two basic IPC paradigms used in distributed systems
    - Message Passing (RPC) and Shared Memory
  - synchronous, asynchronous 通訊 分佈式系統中使用的兩種基本 IPC 範例 消息傳遞 (RPC) 和共享內存 同步、異步
- FileSystems
  - Naming of files/directories
  - File sharing semantics
  - Caching/update/replication 文件系統 文件/目錄的命名 文件共享語義 緩存/更新/複製

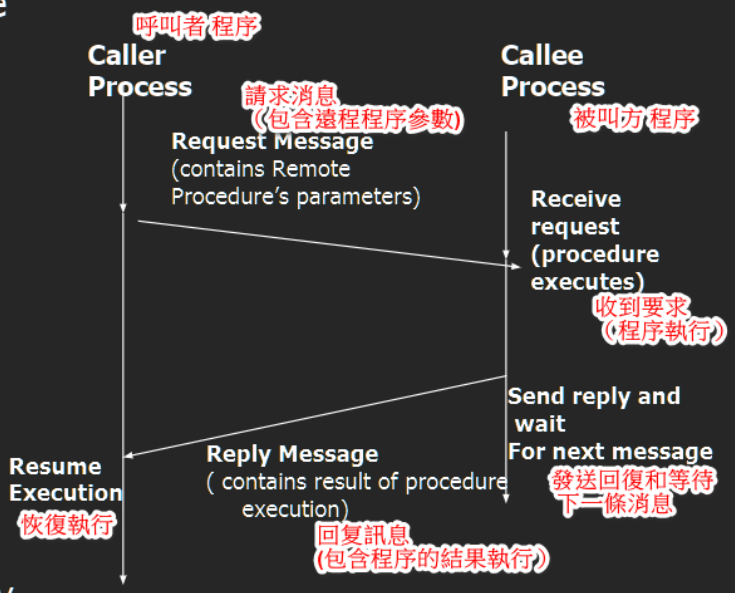
Remote Procedure Call

- Basis of early 2 tier client-server systems
- A convenient way to construct a client-server connection without explicitly writing send/ receive type programs (helps maintain transparency).
- Ideas initiated by RFCs in the 70s
- Landmark paper by Birrell and Nelson in 1980's

遠程過程調用
- 早期(2)層(客戶端-服務器)系統的基礎
- 構建(客戶端-服務器)方式沒有明確寫入(發送/接收)的連接類型程序(有助於保持透明度)
- 70年代由RFCs發起的想法
- Birrell和Nelson在1980年代的論文

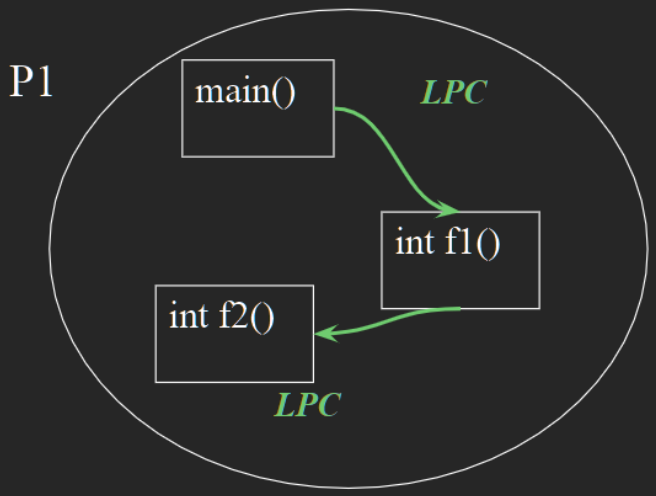
Remote Procedure Calls (RPC)

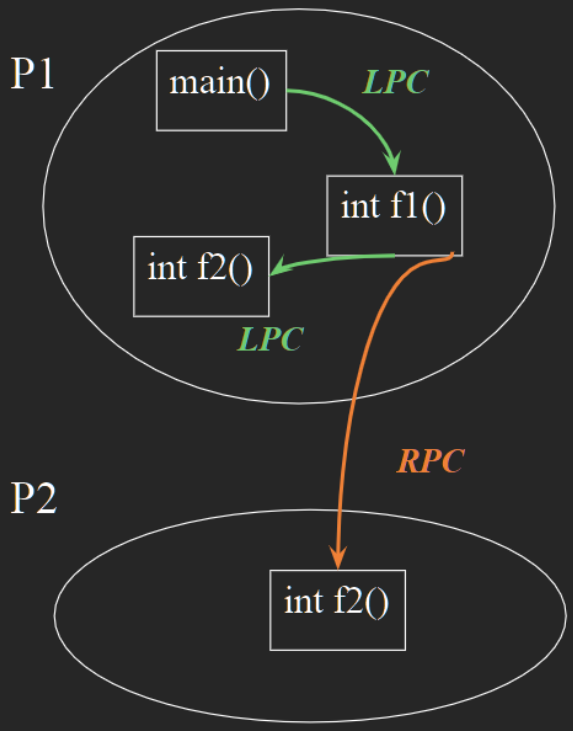
- General message passing model for execution of remote functionality.
  - Provides programmers with a familiar mechanism for building distributed applications/systems
- Familiar semantics (similar to LPC)
  - Simple syntax, well defined interface, ease of use, generality and IPC between processes on same/different machines.
- It is generally synchronous
- Can be made asynchronous by using multi-threading



一般消息傳遞遠程執行模型功能
為程序員提供熟悉的構建機制分佈式應用程序/系統
熟悉的語義(類似於LPC)
語法簡單,定義明確界面,易用性,通用性和IPC之間相同/不同的進程機器
一般是同步的
可以通過以下方式實現異步使用多線程

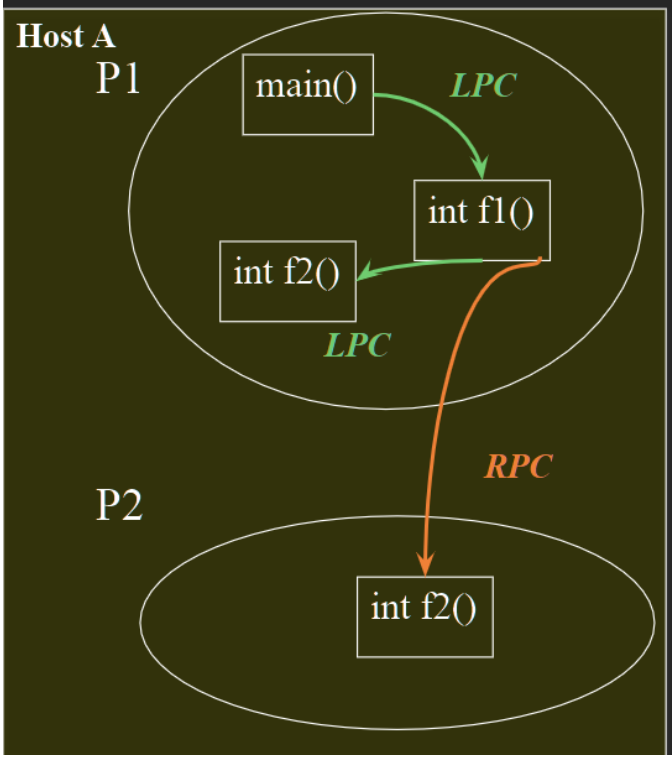
LPC vs. RPC

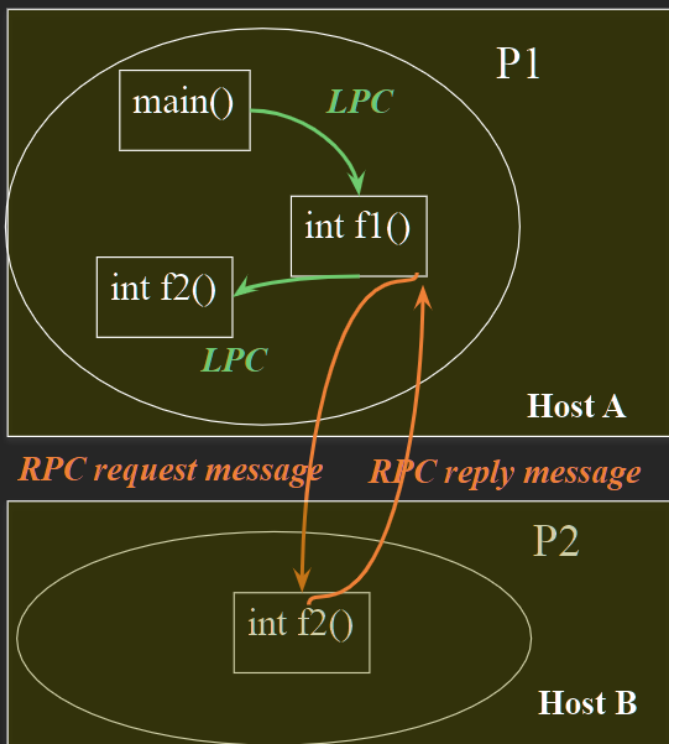




11

LPC vs. RPC





RPC Needs :Syntactic and Semantic Transparency

- Resolve differences in data representation (CDR)
- Support multi-threaded programming
- Provide good reliability
- Provide independence from transport protocols
- Ensure high degree of security
- Locate required services across networks
- Support a variety of execution semantics
  - At most once semantics (e.g., Java RMI)
  - At least once semantics (e.g., Sun RPC)
  - Maybe, i.e., best-effort (e.g., CORBA)

句法和語義透明度(傳送)
- 解決數據表示(CDR)中的差異
- 支持多線程編程
- 提供良好的可靠性
- 獨立於傳輸協議
- 確保高度安全
- 跨網絡定位所需的服務
- 支持多種執行語義
  - 最多一次語義(例如，Java RMI)
  - 至少一次語義(例如，Sun RPC)
  - 也許，即盡力而為(例如，CORBA)

重新執行函數或重新發送回复

Retransmit request 重傳請求	Filter duplicate requests 過濾重複	Re-execute function or retransmit reply	RPC Semantics 重傳請求
Yes	No	Re-execute	At least once
Yes	Yes	Retransmit	At most once
No	NA	NA	Maybe

RPC Challenges

RPC挑戰
為LPC和RPC實現完全相同的語義很困難
- 不相交的地址空間
- 消耗更多時間(由於通信延遲)
- 失敗(很難保證恰好一次語義)

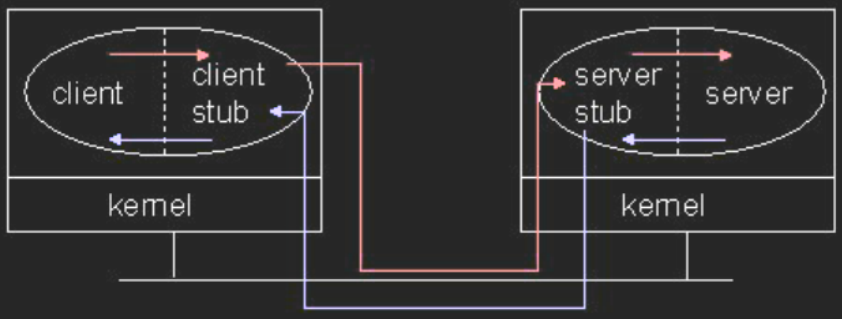
Achieving exactly the same semantics for LPC and RPC is hard

- Disjoint address spaces
- Consumes more time (due to communication delays)
- Failures (hard to guarantee exactly-once semantics)
  - Function may not be executed if
    - Request (call) message is dropped
    - Reply (return) message is dropped
    - Called process fails before executing called function
    - Called process fails after executing called function
    - Hard for caller to distinguish these cases
  - Function may be executed multiple times if request (call) message is duplicated

- 如果出現以下情況，可能無法執行函數
  - 請求(呼叫)消息被丟棄
  - 回复(返回)消息被丟棄
  - 被調用進程在執行被調用函數之前失敗
  - 執行被調用函數後被調用進程失敗
  - 呼叫者難以區分這些情況
- 如果請求(調用)，函數可能會被執行多次消息重複

Implementing RPC - Mechanism

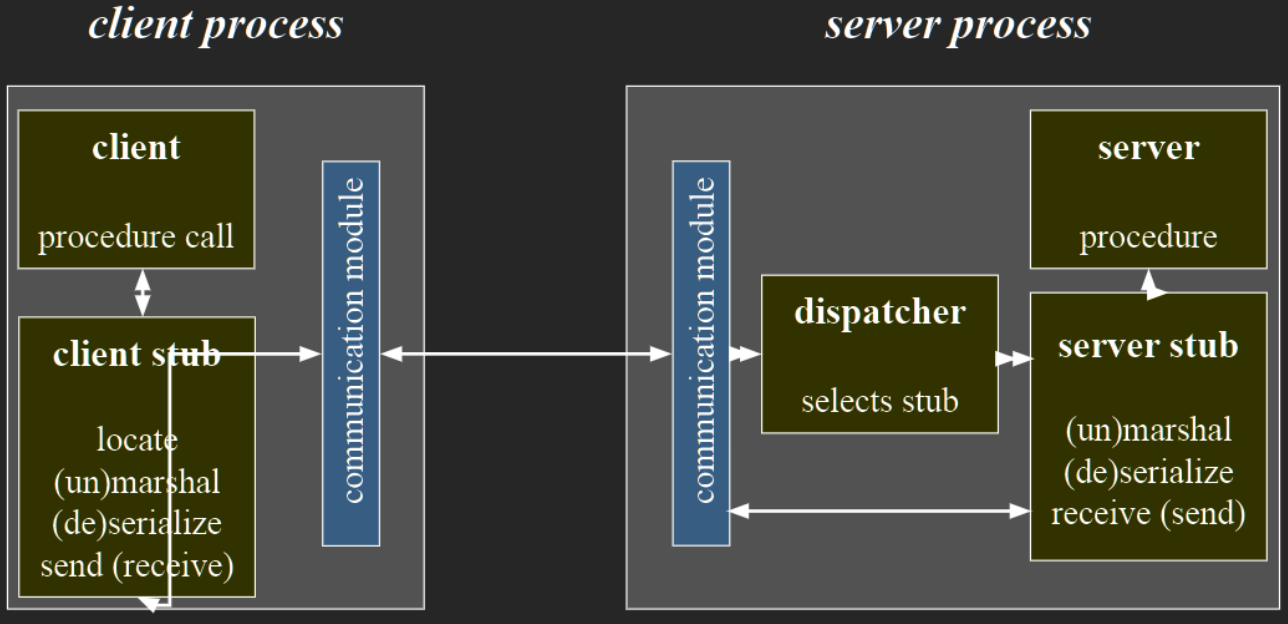
實現 RPC - 機制



- Uses the concept of stubs; A perfectly normal LPC abstraction by concealing from programs the interface to the underlying RPC
- Involves the following elements
  - The client, The client stub
  - The RPC runtime
  - The server stub, The server

- 使用存根的概念：一個完全正常的 LPC 通過向程序隱藏接口來抽象，到底層 RPC
- 涉及以下要素
- 客戶端，客戶端存根
- RPC 運行時
- 服務器存根，服務器

RPC – How it works II



RPC - Steps

RPC - 步驟
- 客戶端過程以正常方式調用客戶端存根
- 客戶端存根構建消息並陷阱到內核
- 內核向遠程內核發送消息
- 遠程內核將消息發送給服務器存根
- 服務器存根解包參數並調用服務器
- 服務器計算結果並返回給服務器存根
- 服務器存根包導致消息並陷阱到內核
- 遠程內核向客戶端內核發送消息
- 客戶端內核向客戶端存根發送消息
- 客戶端存根解包結果並返回給客戶端

- Client procedure **calls** the client stub in a normal way
- Client stub **builds** a message and **traps** to the kernel
- Kernel **sends** the message to remote kernel
- Remote kernel **gives** the message to server stub
- Server stub **unpacks** parameters and **calls** the server
- Server **computes** results and **returns** it to server stub
- Server stub **packs** results in a message and **traps** to kernel
- Remote kernel **sends** message to client kernel
- Client kernel **gives** message to client stub
- Client stub **unpacks** results and **returns** to client

RPC - Marshalling and Unmarshalling

RPC - 編組和解組

- Different architectures use different ways of representing data
  - **Big endian**: Hex 12-AC-33 stored with 12 in lowest address, then AC in next higher address, then 33 in highest address
    - IBM z, System 360
  - **Little endian**: Hex 12-AC-33 stored with 33 in lowest address, then AC in next higher address, then 12
    - Intel
- Caller (and callee) process uses its own *platform-dependent* way of storing data

- 不同的架構使用不同的方法表示數據的
- 大端：十六進制12-AC-33存儲，12在最低地址，然後在AC下一個較高的地址，然後在33，最地址
  - IBM Z，系統360
- 小尾：十六進制12-AC-33存儲最低地址為33，然後為AC下一個較高的地址，然後12
  - 英特爾
- 呼叫者（方和被叫方過程中使用它自己的存儲數據的依賴於平台的方式

- Middleware has a common data representation (CDR)
  - Platform-independent
- Caller process converts arguments into CDR format
  - Called "Marshalling"
- Callee process extracts arguments from message into its own platform-dependent format
  - Called "Unmarshalling"
- Return values are marshalled on callee process and unmarshalled at caller process

- 中間件具有共同的數據代表（CDR）－獨立於平台
- 調用者進程轉換參數轉換為CDR格式－稱為"編組"
- 被調用者進程提取從消息到其的參數自己的平台相關格式－稱為"解組"
- 返回值編組在被調用進程和解組在調用者進程

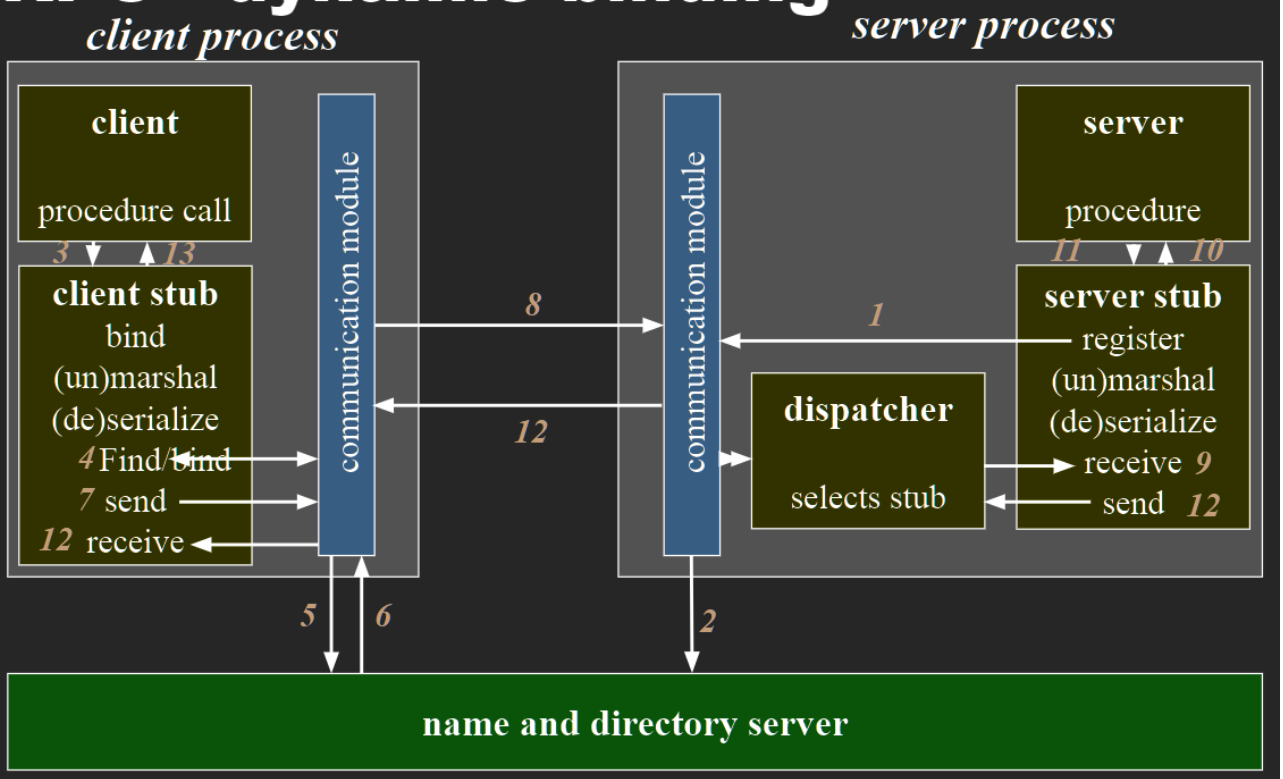
RPC - binding

- Static binding
  - hard coded stub
    - Simple, efficient
    - not flexible
  - stub recompilation necessary if the location of the server changes
  - use of redundant servers not possible
- Dynamic binding
  - name and directory server
    - load balancing
  - IDL used for binding
  - flexible
  - redundant servers possible

RPC - 綁定

- 靜態綁定
  - 硬編碼存根
    - 簡單
    - 高效
    - 不靈活
  - stub重新編譯如果服務器的位置動態綁定
  - 無法使用冗餘服務器
- 動態綁定
  - 名稱和目錄服務器
    - 負載均衡
  - 用於綁定的IDL
  - 靈活
  - 可以使用冗餘服務器

RPC - dynamic binding



Wolfgang Gassler, Eva Zangerle

RPC - Extensions

- conventional RPC: sequential execution of routines
- client blocked until response of server
- asynchronous RPC – non blocking
  - client has two entry points(request and response)
  - server stores result in shared memory
  - client picks it up from there

RPC - 擴展
- 常規 RPC 是順序例行程序的執行
- 客戶端被阻塞，直到服務器響應
- 異步 RPC 是非阻塞
  - 客戶端有兩個入口點（請求和回复）
  - 服務器將結果存儲在共享內存中
  - 客戶端從那裡取

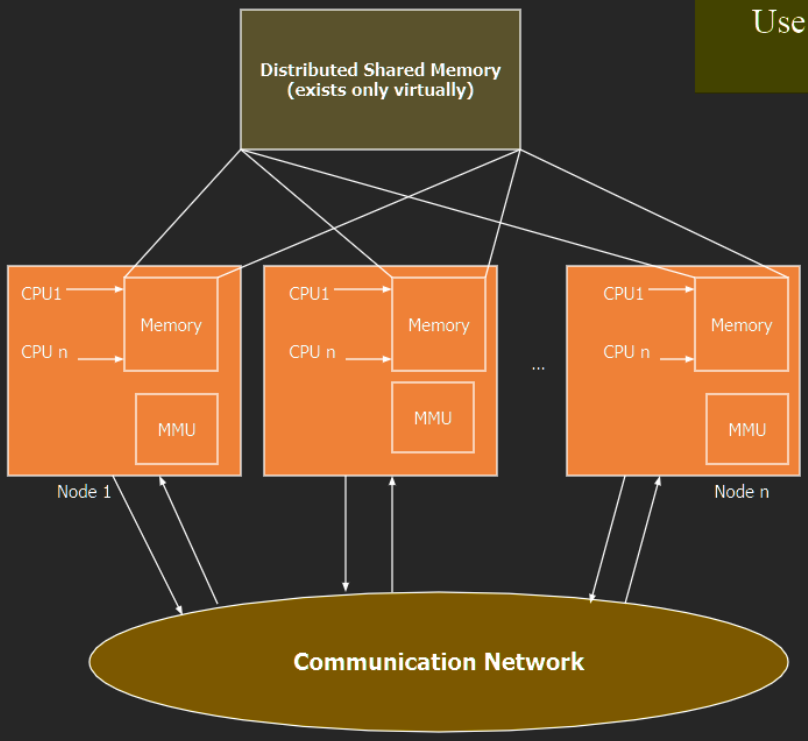
RPC servers and protocols...

- RPC Messages (call and reply messages)
- Server Implementation
  - Stateful servers
  - Stateless servers
- Communication Protocols
  - Request(R)Protocol
  - Request/Reply(RR) Protocol
  - Request/Reply/Ack(RRA) Protocol

RPC 服務器和協議...
- RPC Messages（調用和回复消息）
- 服務器實現
  - 有狀態的服務器
  - 無狀態服務器
- 通訊協議
  - 請求(R)協議
  - 請求/回复（(RR）協議
  - 請求/回复/確認（(RRA）協議
- 冪等操作 - 可以重複多次，沒有任何副作用
- 範例（(x 是服務器端變量）
  - x=1；
- 非範例
  - x=x+1；
  - x=x*2

- Idempotent operations - can be repeated multiple times, without any side effects
- Examples (x is server-side variable)
  - x=1;
- Non-examples
  - x=x+1;
  - x=x*2

Distributed Shared Memory (DSM)



Tightly coupled systems
Use of shared memory for IPC is natural

緊耦合系統
為IPC使用共享內存是很自然的

Loosely coupled distributed-memory processors
Use DSM – distributed shared memory
A middleware solution that provides a shared-memory abstraction.

鬆散耦合
分佈式內存處理器
使用DSM=分佈式共享記憶
=個中間件解決方案
提供抽象共享內存

Issues in designing DSM

- *Synchronization*
- Granularity of the block size
- Memory Coherence (Consistency models)
- Data Location and Access
- Replacement Strategies
- Thrashing
- Heterogeneity

設計DSM中的問題
- 同步
- 塊大小的粒度
- 內存一致性(一致性模型)
- 數據定位和訪問
- 更換策略
- 顛簸
- 異質性

Synchronization

- Inevitable in Distributed Systems where distinct processes are running concurrently and sharing resources.
- Synchronization related issues
  - Clock synchronization/Event Ordering (recall happened before relation)
  - Mutual exclusion
  - Deadlocks
  - Election Algorithms

同步
- 在分佈式系統中不可避免進程同時運行並共享資源。
- 同步相關問題
  - 時鐘同步/事件排序（召回發生在關係）
  - 互斥
  - 死鎖
  - 選舉算法

Distributed Mutual Exclusion

分佈式互惠排除

相互排斥
- 確保並發進程可以序列化訪問共享資源 - 臨界區問題
- 共享變量（信號量）不能用於分佈式系統
  - 互斥必須基於消息傳遞，
    在不可預測的延遲和不完整的知識背景
- 在某些應用程序（例如事務處理）中，
  資源由實現自己的服務器管理
  鎖定以及同步訪問的機制資源。

- Mutual exclusion
  - ensures that concurrent processes have serialized access to shared resources - the critical section problem
  - Shared variables (semaphores) cannot be used in a distributed system
    - Mutual exclusion must be based on message passing, in the context of unpredictable delays and incomplete knowledge
  - In some applications (e.g. transaction processing) the resource is managed by a server which implements its own lock along with mechanisms to synchronize access to the resource.

Distributed Mutual Exclusion

分佈式互惠排除

- 基本要求
  - 安全
    - 至多一個進程可以在危急時刻執行(CS)一次
  - 活力
    - 請求進入CS的進程最終是授予它(只要任何進程在其CS最終離開了它)。
  - 意味著免於死鎖和飢餓

- Basic requirements
  - Safety
    - At most one process may execute in the critical section (CS) at a time
  - Liveness
    - A process requesting entry to the CS is eventually granted it (as long as any process executing in its CS eventually leaves it.
    - Implies freedom from deadlock and starvation

Mutual Exclusion Techniques

互斥技術

- **Non-token Based Approaches** 非token的方法
  - Each process freely and equally competes for the right to use the shared resource; requests are arbitrated by a central control suite or by distributed agreement

    每個進程都自由平等地爭奪使用權
    共享資源；請求由中央控制仲裁
    套件或通過分佈式協議
    - Central Coordinator Algorithm
    - Ricart-Agrawala Algorithm

    中央協調器算法
    Ricart-Agrawala 算法
- **Token-based approaches** token的方法
  - A logical token representing the access right to the shared resource is passed in a regulated fashion among processes; whoever holds the token is allowed to enter the critical section.

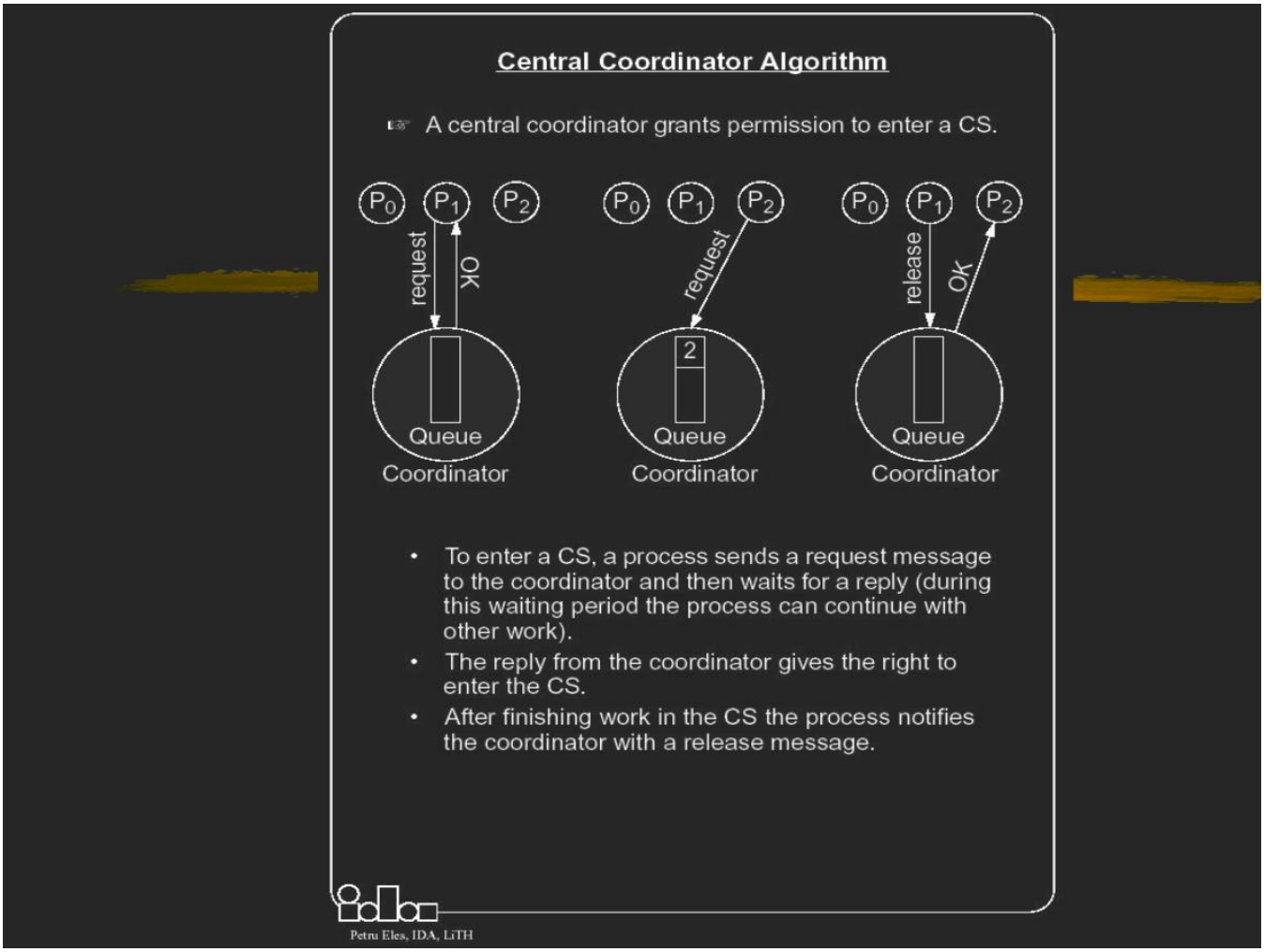
    表示對共享訪問權的邏輯標記
    資源在過程中的調節的方式通過;
    凡持有TOKEN被允許進入臨界區
    - Token Ring Algorithm
    - Ricart-Agrawala Second Algorithm

    • Token Ring 算法
    • Ricart-Agrawala 第二算法
- **Quorum-based approaches**

  基於群體的方法
  - Need to coordinate with only some members in a group (quorum).

    只需要與組（法定人數）中的一些成員進行協調。
    - Maekawa's Algorithm -- voting sets

    • Maekawa 的算法——投票集



Distributed Operating Systems

Prof. Nalini Venkatasubramanian

(includes slides borrowed from Prof. Petru Eles, lecture slides from Coulouris, Dollimore and Kindberg textbook, MIT course notes, slides and animations from UIUC CS425 Indranil Gupta)

分散式作業 系統

教授。納里尼·文卡塔薩布拉馬尼安

包括從 Petru Eles 教授那裡借來的幻燈片，從 Coulouris、Dollimore 和 Kindberg

教科書、麻省理工學院課程筆記、幻燈片和 UIUC CS425 Indranil Gupta 動畫

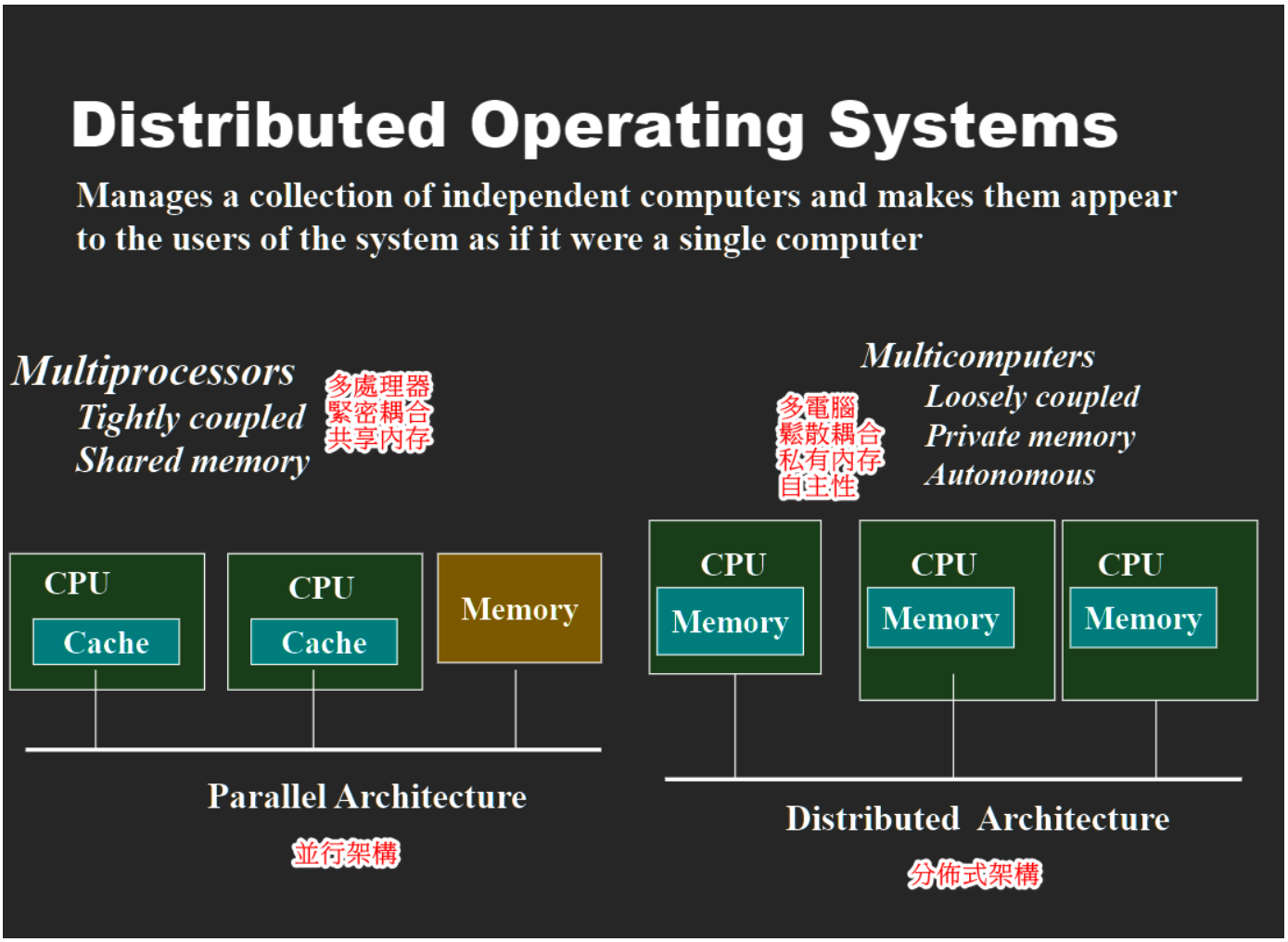
OS 做了那些事？

●進程/線程管理

- 排程
- 通訊
- 同步

- 內存管理
- 存儲管理

- 文件系統管理
- 保護和安全
- 連網

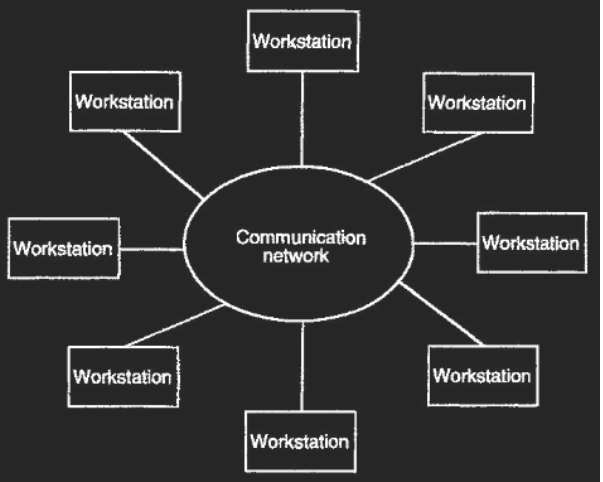


Early Systems - Workstation Model

早期系統 - 工作站模型

怎麼找控閒的工作站？
●流程如何轉移：從一個工作站到其他？
●當用戶登錄到空閒的工作站，遠端進程會發生什麼，現在不再變成空閒著嗎？
●其他狀態：處理器pool，工作站Server...
示例：伯克利NOW項目、威斯康星大學CONDOR系統

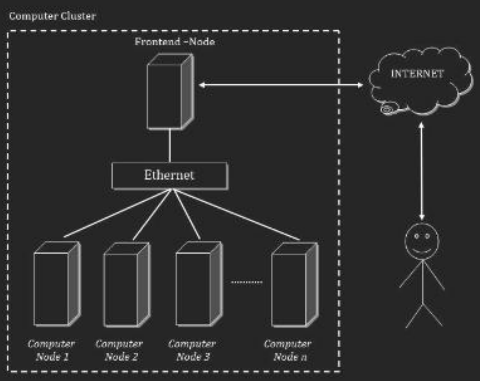
- How to find an idle workstation?
- How is a process transferred from one workstation to another?
- What happens to a remote process if a user logs onto a workstation that was idle, but is no longer idle now?
- Other models - processor pool, workstation server...

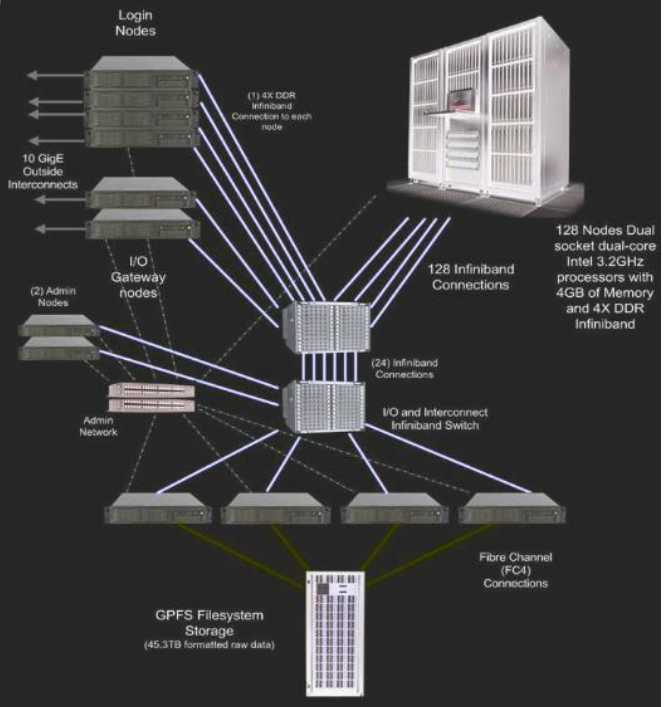


Examples: Berkeley NOW Project, U Wisconsin CONDOR system

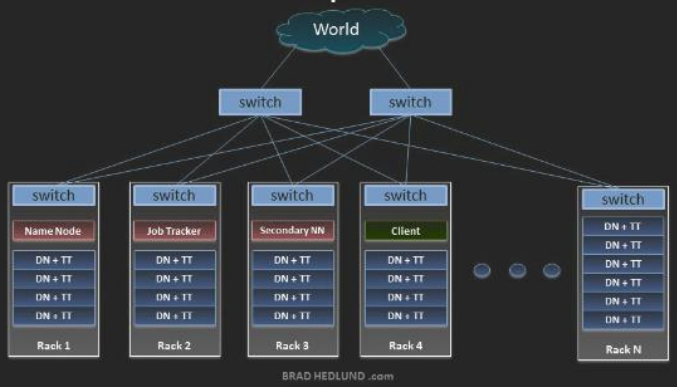
Cluster Computing

Loosely connected set of machines that behave as a unit





Hadoop Cluster



High Performance Computing (HPC) Cluster

Distributed Operating System (DOS) Types

分散式作業系統 (DOS) 類型

- Distributed OSs vary based on
  - System Image
  - Autonomy
  - Fault Tolerance Capability

  系統鏡像
  自主(自治)
  容錯能力
- Multiprocessor OS
  - Looks like a virtual uniprocessor, contains only one copy of the OS, communicates via shared memory, single run queue

  看起來像一個虛擬單處理器，只包含一個操作系統副本，通過共享內存，單運行隊列進行通信
- Network OS
  - Does not look like a virtual uniprocessor, contains n copies of the OS, communicates via shared files, n run queues

  看起來不像虛擬單處理器，包含操作系統的 n 個副本，通過共享文件進行通信，n 個運行隊列
- Distributed OS
  - Looks like a virtual uniprocessor (more or less), contains n copies of the OS/runtime, communicates via messages, n run queues

  看起來像一個虛擬單處理器（或多或少），包含 n 個操作系統/運行時，通過消息進行通信，n 個運行隊列

Design Issues (cont.) 設計問題

- *Transparency* 透明度(傳送) 位置傳送 進程、CPU 和其他設備、文件
  - Location transparency
    - processes, cpu's and other devices, files
  - Replication transparency (of files) 複製傳送 (文件)
  - Concurrency transparency
    - (user unaware of the existence of others)
  - Parallelism 並發傳送 (用戶不知道他人的存在)
    - User writes serial program, compiler and OS do the rest 並行性 用戶寫入串行程序、編譯器和作業系統休息
- *Performance*
  - Metrics
    - Throughput, response time
  - Load Balancing (static, dynamic) 效能 指標 吞吐量、響應時間 負載均衡 (靜態、動態的)
  - Communication is slow compared to computation speed
    - fine grain, coarse grain parallelism tradeoff 通訊緩慢與計算相比速度 細粒、粗粒並行權衡

Also : Scalability, Reliability, Flexibility, Heterogeneity, Security

另外：可擴展性、可靠性、靈活性、異構性、安全性

Design Elements 設計元素

- Process Management
  - Task Partitioning, allocation, synchronization, load balancing, migration 流程管理 任務分區、分配、同步、加載、平衡、遷移
- Communication
  - Two basic IPC paradigms used in distributed systems
    - Message Passing (RPC) and Shared Memory
  - synchronous, asynchronous 通訊 分佈式系統中使用的兩種基本 IPC 範例 消息傳遞 (RPC) 和共享內存 同步、異步
- FileSystems
  - Naming of files/directories
  - File sharing semantics
  - Caching/update/replication 文件系統 文件/目錄的命名 文件共享語義 緩存/更新/複製

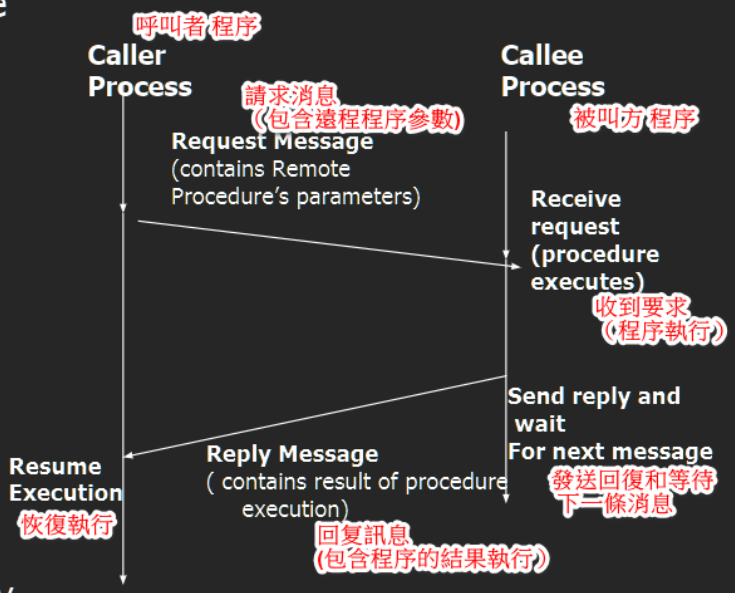
Remote Procedure Call

- Basis of early 2 tier client-server systems
- A convenient way to construct a client-server connection without explicitly writing send/ receive type programs (helps maintain transparency).
- Ideas initiated by RFCs in the 70s
- Landmark paper by Birrell and Nelson in 1980's

遠程過程調用
- 早期(2)層(客戶端-服務器)系統的基礎
- 構建(客戶端-服務器)方式沒有明確寫入(發送/接收)的連接類型程序(有助於保持透明度)
- 70年代由RFCs發起的想法
- Birrell和Nelson在1980年代的論文

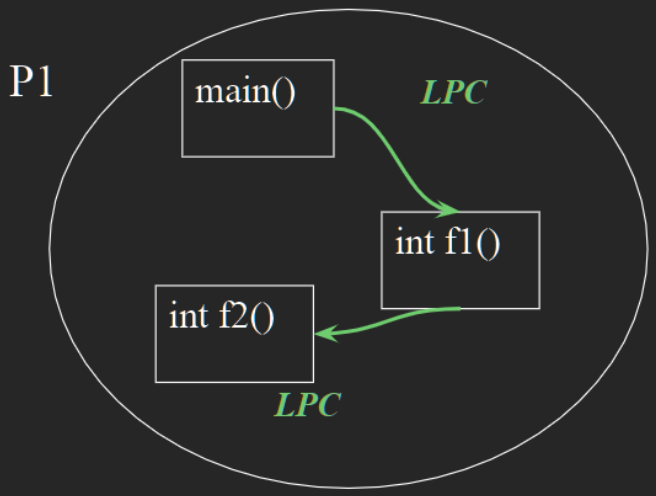
Remote Procedure Calls (RPC)

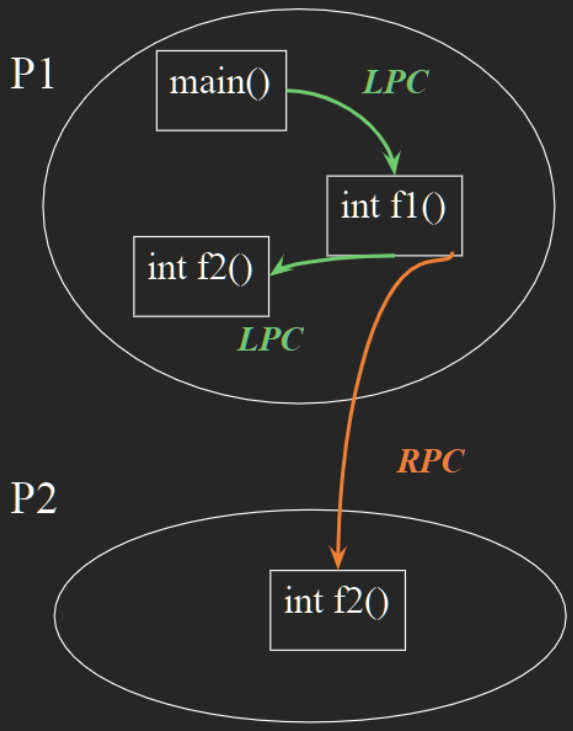
- General message passing model for execution of remote functionality.
  - Provides programmers with a familiar mechanism for building distributed applications/systems
- Familiar semantics (similar to LPC)
  - Simple syntax, well defined interface, ease of use, generality and IPC between processes on same/different machines.
- It is generally synchronous
- Can be made asynchronous by using multi-threading



一般消息傳遞遠程執行模型功能
為程序員提供熟悉的構建機制分佈式應用程序/系統
熟悉的語義(類似於LPC)
語法簡單,定義明確界面,易用性,通用性和IPC之間相同/不同的進程機器
一般是同步的
可以通過以下方式實現異步使用多線程

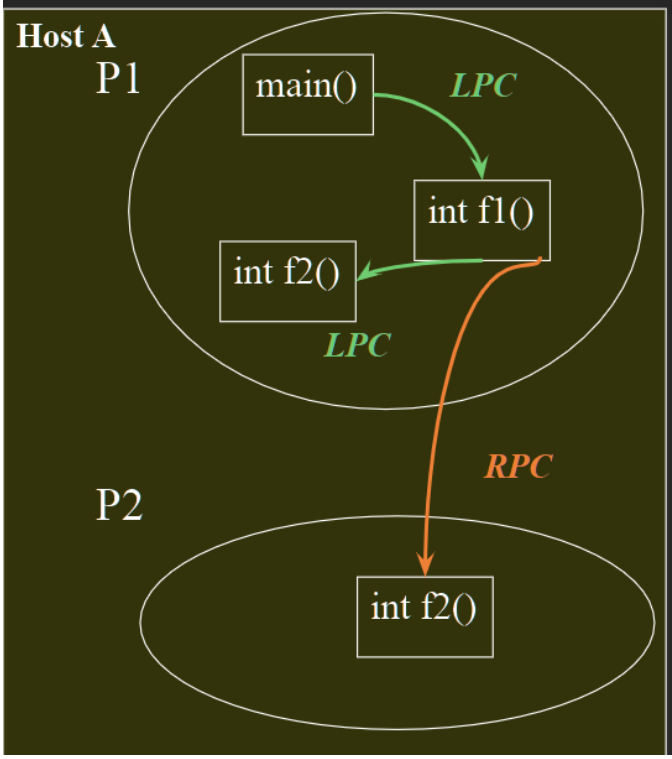
LPC vs. RPC

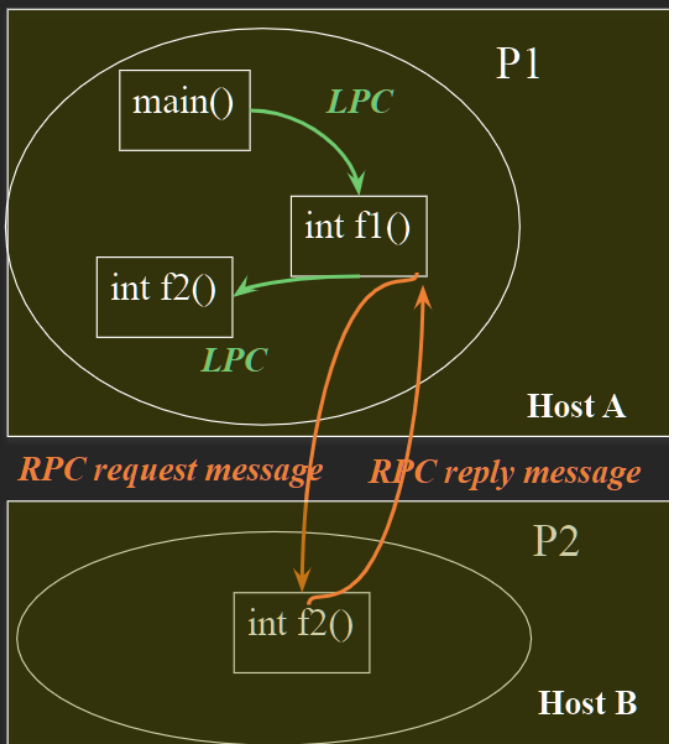




11

LPC vs. RPC





RPC Needs :Syntactic and Semantic Transparency

- Resolve differences in data representation (CDR)
- Support multi-threaded programming
- Provide good reliability
- Provide independence from transport protocols
- Ensure high degree of security
- Locate required services across networks
- Support a variety of execution semantics
  - At most once semantics (e.g., Java RMI)
  - At least once semantics (e.g., Sun RPC)
  - Maybe, i.e., best-effort (e.g., CORBA)

句法和語義透明度(傳送)
- 解決數據表示(CDR)中的差異
- 支持多線程編程
- 提供良好的可靠性
- 獨立於傳輸協議
- 確保高度安全
- 跨網絡定位所需的服務
- 支持多種執行語義
  - 最多一次語義(例如，Java RMI)
  - 至少一次語義(例如，Sun RPC)
  - 也許，即盡力而為(例如，CORBA)

重新執行函數或重新發送回复

Retransmit request 重傳請求	Filter duplicate requests 過濾重複	Re-execute function or retransmit reply	RPC Semantics 重傳請求
Yes	No	Re-execute	At least once
Yes	Yes	Retransmit	At most once
No	NA	NA	Maybe

RPC Challenges

RPC挑戰
為LPC和RPC實現完全相同的語義很困難
- 不相交的地址空間
- 消耗更多時間(由於通信延遲)
- 失敗(很難保證恰好一次語義)

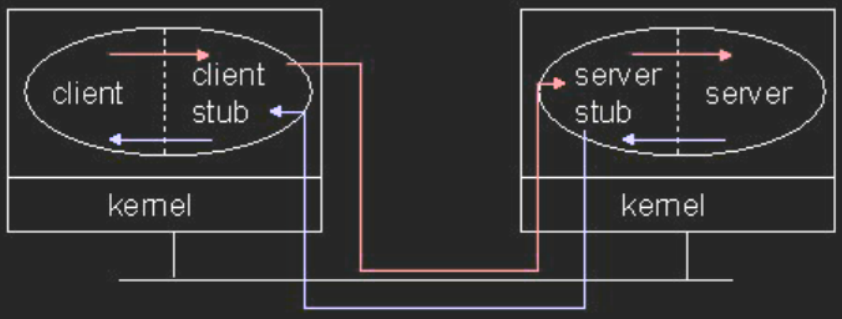
Achieving exactly the same semantics for LPC and RPC is hard

- Disjoint address spaces
- Consumes more time (due to communication delays)
- Failures (hard to guarantee exactly-once semantics)
  - Function may not be executed if
    - Request (call) message is dropped
    - Reply (return) message is dropped
    - Called process fails before executing called function
    - Called process fails after executing called function
    - Hard for caller to distinguish these cases
  - Function may be executed multiple times if request (call) message is duplicated

- 如果出現以下情況，可能無法執行函數
  - 請求(呼叫)消息被丟棄
  - 回复(返回)消息被丟棄
  - 被調用進程在執行被調用函數之前失敗
  - 執行被調用函數後被調用進程失敗
  - 呼叫者難以區分這些情況
- 如果請求(調用)，函數可能會被執行多次消息重複

Implementing RPC - Mechanism

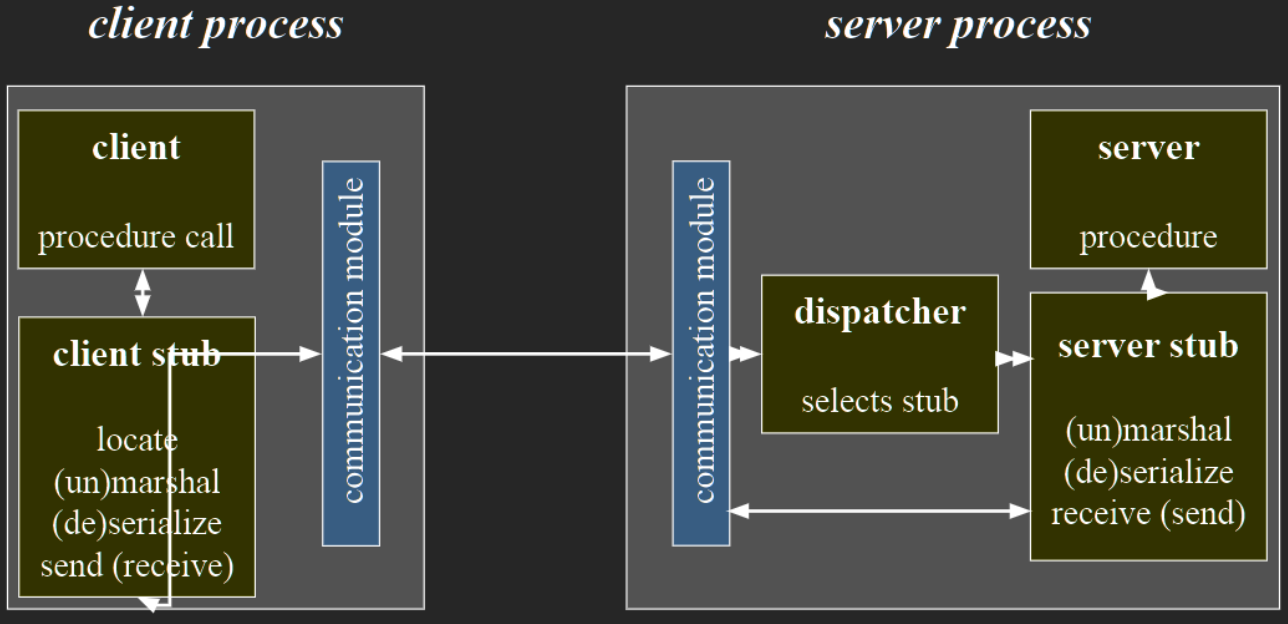
實現 RPC - 機制



- Uses the concept of stubs; A perfectly normal LPC abstraction by concealing from programs the interface to the underlying RPC
- Involves the following elements
  - The client, The client stub
  - The RPC runtime
  - The server stub, The server

- 使用存根的概念：一個完全正常的 LPC 通過向程序隱藏接口來抽象，到底層 RPC
- 涉及以下要素
- 客戶端，客戶端存根
- RPC 運行時
- 服務器存根，服務器

RPC – How it works II



RPC - Steps

RPC - 步驟
- 客戶端過程以正常方式調用客戶端存根
- 客戶端存根構建消息並陷阱到內核
- 內核向遠程內核發送消息
- 遠程內核將消息發送給服務器存根
- 服務器存根解包參數並調用服務器
- 服務器計算結果並返回給服務器存根
- 服務器存根包導致消息並陷阱到內核
- 遠程內核向客戶端內核發送消息
- 客戶端內核向客戶端存根發送消息
- 客戶端存根解包結果並返回給客戶端

- Client procedure **calls** the client stub in a normal way
- Client stub **builds** a message and **traps** to the kernel
- Kernel **sends** the message to remote kernel
- Remote kernel **gives** the message to server stub
- Server stub **unpacks** parameters and **calls** the server
- Server **computes** results and **returns** it to server stub
- Server stub **packs** results in a message and **traps** to kernel
- Remote kernel **sends** message to client kernel
- Client kernel **gives** message to client stub
- Client stub **unpacks** results and **returns** to client

RPC - Marshalling and Unmarshalling

RPC - 編組和解組

- Different architectures use different ways of representing data
  - **Big endian**: Hex 12-AC-33 stored with 12 in lowest address, then AC in next higher address, then 33 in highest address
    - IBM z, System 360
  - **Little endian**: Hex 12-AC-33 stored with 33 in lowest address, then AC in next higher address, then 12
    - Intel
- Caller (and callee) process uses its own *platform-dependent* way of storing data

- 不同的架構使用不同的方法表示數據的
- 大端：十六進制12-AC-33存儲，12在最低地址，然後在AC下一個較高的地址，然後在33，最地址
  - IBM Z，系統360
- 小尾：十六進制12-AC-33存儲最低地址為33，然後為AC下一個較高的地址，然後12
  - 英特爾
- 呼叫者（方和被叫方過程中使用它自己的存儲數據的依賴於平台的方式

- Middleware has a common data representation (CDR)
  - Platform-independent
- Caller process converts arguments into CDR format
  - Called "Marshalling"
- Callee process extracts arguments from message into its own platform-dependent format
  - Called "Unmarshalling"
- Return values are marshalled on callee process and unmarshalled at caller process

- 中間件具有共同的數據代表（CDR）－獨立於平台
- 調用者進程轉換參數轉換為CDR格式－稱為"編組"
- 被調用者進程提取從消息到其的參數自己的平台相關格式－稱為"解組"
- 返回值編組在被調用進程和解組在調用者進程

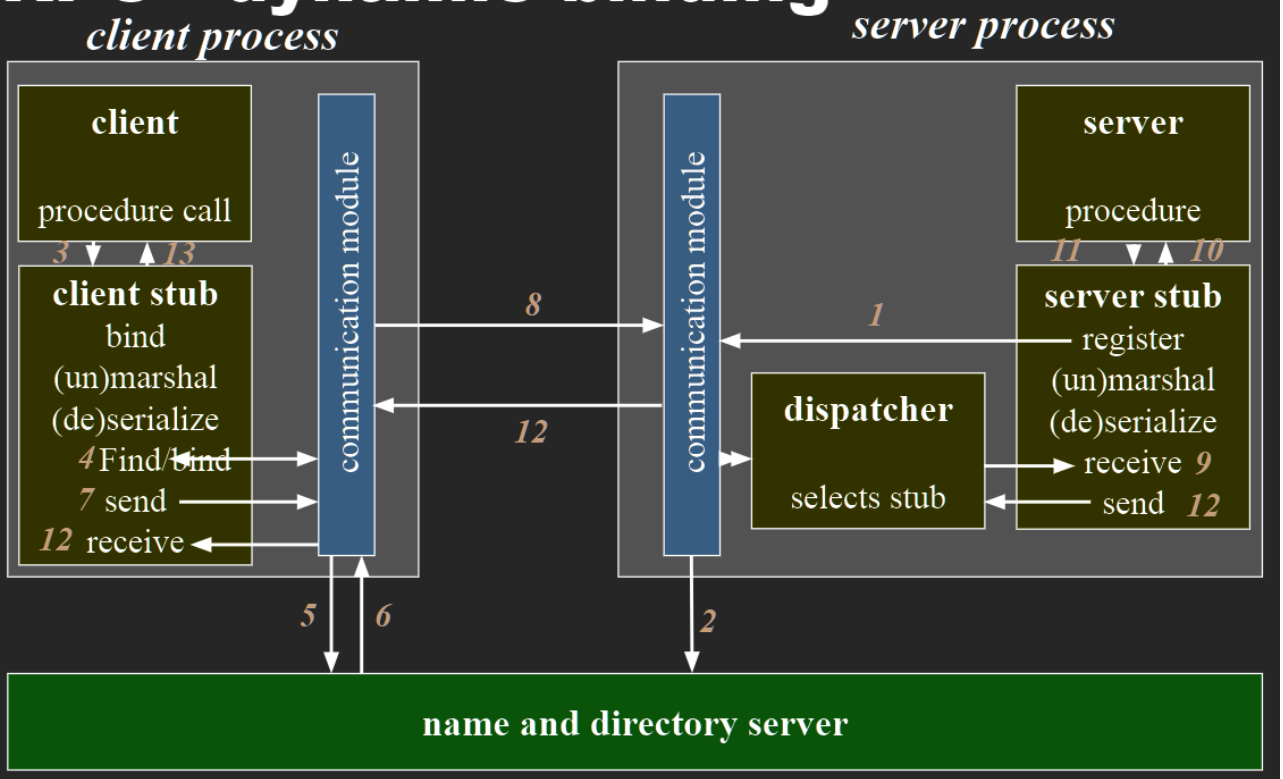
RPC - binding

- Static binding
  - hard coded stub
    - Simple, efficient
    - not flexible
  - stub recompilation necessary if the location of the server changes
  - use of redundant servers not possible
- Dynamic binding
  - name and directory server
    - load balancing
  - IDL used for binding
  - flexible
  - redundant servers possible

RPC - 綁定

- 靜態綁定
  - 硬編碼存根
    - 簡單
    - 高效
    - 不靈活
  - stub重新編譯如果服務器的位置動態綁定
  - 無法使用冗餘服務器
- 動態綁定
  - 名稱和目錄服務器
    - 負載均衡
  - 用於綁定的IDL
  - 靈活
  - 可以使用冗餘服務器

RPC - dynamic binding



Wolfgang Gassler, Eva Zangerle

RPC - Extensions

- conventional RPC: sequential execution of routines
- client blocked until response of server
- asynchronous RPC – non blocking
  - client has two entry points(request and response)
  - server stores result in shared memory
  - client picks it up from there

RPC - 擴展
- 常規 RPC 是順序例行程序的執行
- 客戶端被阻塞，直到服務器響應
- 異步 RPC 是非阻塞
  - 客戶端有兩個入口點（請求和回复）
  - 服務器將結果存儲在共享內存中
  - 客戶端從那裡取

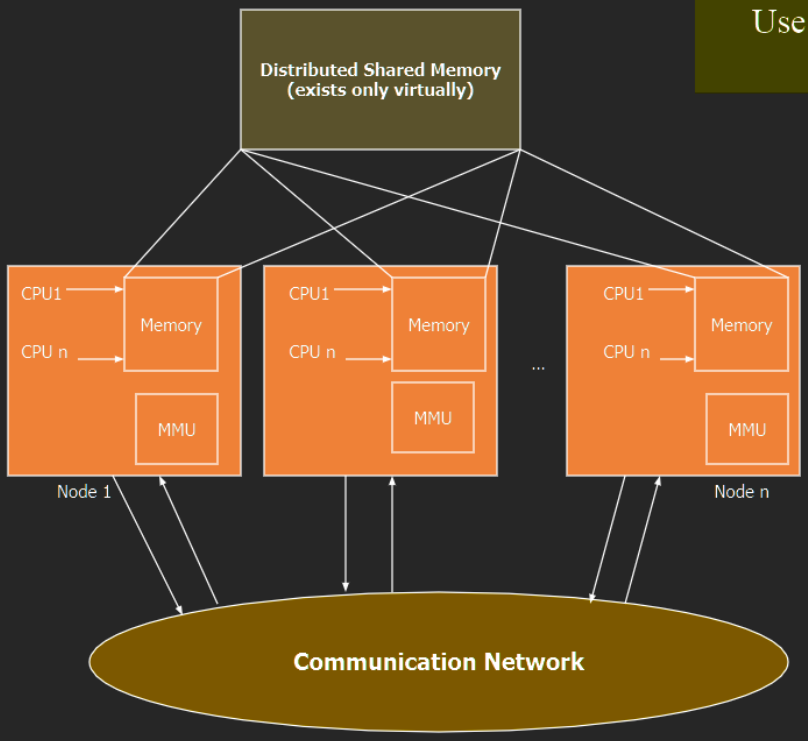
RPC servers and protocols...

- RPC Messages (call and reply messages)
- Server Implementation
  - Stateful servers
  - Stateless servers
- Communication Protocols
  - Request(R)Protocol
  - Request/Reply(RR) Protocol
  - Request/Reply/Ack(RRA) Protocol

RPC 服務器和協議...
- RPC Messages（調用和回复消息）
- 服務器實現
  - 有狀態的服務器
  - 無狀態服務器
- 通訊協議
  - 請求(R)協議
  - 請求/回复（(RR）協議
  - 請求/回复/確認（(RRA）協議
- 冪等操作 - 可以重複多次，沒有任何副作用
- 範例（(x 是服務器端變量）
  - x=1；
- 非範例
  - x=x+1；
  - x=x*2

- Idempotent operations - can be repeated multiple times, without any side effects
- Examples (x is server-side variable)
  - x=1;
- Non-examples
  - x=x+1;
  - x=x*2

Distributed Shared Memory (DSM)



Tightly coupled systems
Use of shared memory for IPC is natural

緊耦合系統
為IPC使用共享內存是很自然的

Loosely coupled distributed-memory processors
Use DSM – distributed shared memory
A middleware solution that provides a shared-memory abstraction.

鬆散耦合
分佈式內存處理器
使用DSM=分佈式共享記憶
=個中間件解決方案
提供抽象共享內存

Issues in designing DSM

- *Synchronization*
- Granularity of the block size
- Memory Coherence (Consistency models)
- Data Location and Access
- Replacement Strategies
- Thrashing
- Heterogeneity

設計DSM中的問題
- 同步
- 塊大小的粒度
- 內存一致性(一致性模型)
- 數據定位和訪問
- 更換策略
- 顛簸
- 異質性

Synchronization

- Inevitable in Distributed Systems where distinct processes are running concurrently and sharing resources.
- Synchronization related issues
  - Clock synchronization/Event Ordering (recall happened before relation)
  - Mutual exclusion
  - Deadlocks
  - Election Algorithms

同步
- 在分佈式系統中不可避免進程同時運行並共享資源。
- 同步相關問題
  - 時鐘同步/事件排序（召回發生在關係）
  - 互斥
  - 死鎖
  - 選舉算法

Distributed Mutual Exclusion

分佈式互惠排除

相互排斥
- 確保並發進程可以序列化訪問共享資源 - 臨界區問題
- 共享變量（信號量）不能用於分佈式系統
  - 互斥必須基於消息傳遞，
    在不可預測的延遲和不完整的知識背景
- 在某些應用程序（例如事務處理）中，
  資源由實現自己的服務器管理
  鎖定以及同步訪問的機制資源。

- Mutual exclusion
  - ensures that concurrent processes have serialized access to shared resources - the critical section problem
  - Shared variables (semaphores) cannot be used in a distributed system
    - Mutual exclusion must be based on message passing, in the context of unpredictable delays and incomplete knowledge
  - In some applications (e.g. transaction processing) the resource is managed by a server which implements its own lock along with mechanisms to synchronize access to the resource.

Distributed Mutual Exclusion

分佈式互惠排除

- 基本要求
  - 安全
    - 至多一個進程可以在危急時刻執行(CS)一次
  - 活力
    - 請求進入CS的進程最終是授予它(只要任何進程在其CS最終離開了它)。
  - 意味著免於死鎖和飢餓

- Basic requirements
  - Safety
    - At most one process may execute in the critical section (CS) at a time
  - Liveness
    - A process requesting entry to the CS is eventually granted it (as long as any process executing in its CS eventually leaves it.
    - Implies freedom from deadlock and starvation

Mutual Exclusion Techniques

互斥技術

- **Non-token Based Approaches** 非token的方法
  - Each process freely and equally competes for the right to use the shared resource; requests are arbitrated by a central control suite or by distributed agreement

    每個進程都自由平等地爭奪使用權
    共享資源；請求由中央控制仲裁
    套件或通過分佈式協議
    - Central Coordinator Algorithm
    - Ricart-Agrawala Algorithm

    中央協調器算法
    Ricart-Agrawala 算法
- **Token-based approaches** token的方法
  - A logical token representing the access right to the shared resource is passed in a regulated fashion among processes; whoever holds the token is allowed to enter the critical section.

    表示對共享訪問權的邏輯標記
    資源在過程中的調節的方式通過;
    凡持有TOKEN被允許進入臨界區
    - Token Ring Algorithm
    - Ricart-Agrawala Second Algorithm

    • Token Ring 算法
    • Ricart-Agrawala 第二算法
- **Quorum-based approaches**

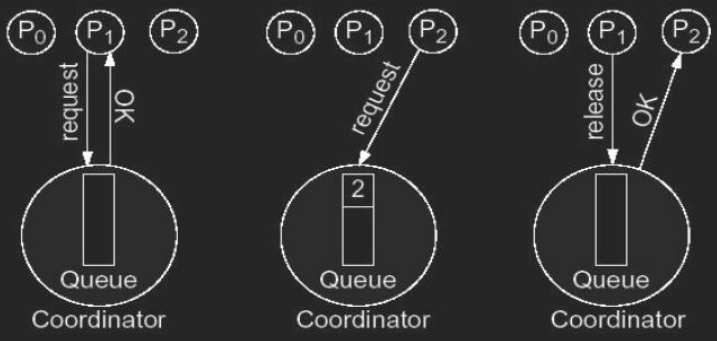
  基於群體的方法
  - Need to coordinate with only some members in a group (quorum).

    只需要與組（法定人數）中的一些成員進行協調。
    - Maekawa's Algorithm -- voting sets

    • Maekawa 的算法——投票集

Central Coordinator Algorithm

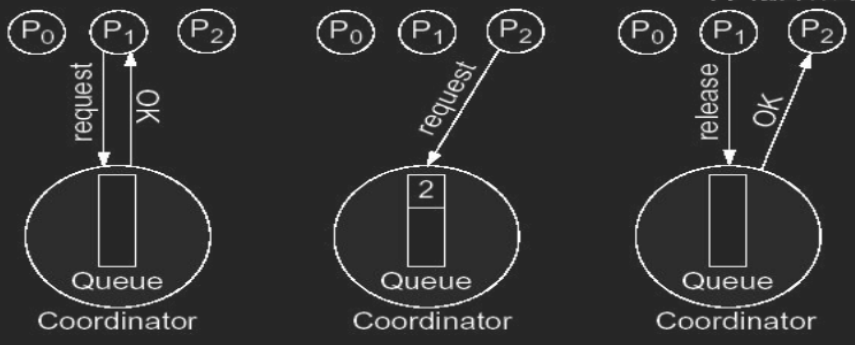
☞ A central coordinator grants permission to enter a CS.



- To enter a CS, a process sends a request message to the coordinator and then waits for a reply (during this waiting period the process can continue with other work).
- The reply from the coordinator gives the right to enter the CS.
- After finishing work in the CS the process notifies the coordinator with a release message.

Central Coordinator Algorithm

中央協調器算法



- A central coordinator grants permission to enter the critical section (CS). 中央協調器授予進入臨界區的權限（CS）
- To enter the CS, a process 進入CS，一個流程
  - sends a ***request*** message to the coordinator and 將請求消息發送到協調和
  - waits for a ***reply*** (during this waiting period the process can continue work) 等待回復（在此等待期間該過程可以繼續工作）
- The reply from the coordinator gives a process the right to enter the CS. 協調員的回復賦予進程進入的權利CS。
- After completing work in the CS, the process notifies the coordinator with a ***release*** message. 在CS中完成工作後，該過程通知帶有發布消息的協調器。

Central Coordinator Algorithm

中央協調器算法

- Advantages
  - Scheme is easy to implement
  - Requires only 3 messages to get access to a critical section (request, OK, release)
- 優勢
  - 方案易於實施
  - 只需要3條消息即可訪問臨界區（請求，確定，釋放）
- Issues with centralized coordinator
  - A performance bottleneck
  - Critical point of Failure
    - New coordinator must be elected when existing coordinator fails
    - New coordinator can be one of the processes competing for access to the CS
    - A *leader election* algorithm must select one and only one coordinator.
- 集中協調器的問題
  - 性能瓶頸
  - 故障臨界點
    - 新協調員存在時必須選舉協調器失敗
    - 新協調器可以是競爭進程之一用於訪問CS
    - 一個leader選舉算法必須選擇一個並且只有一個協調員。

Ricart-Agrawara 算法 1

Ricart-Agrawala Algorithm 1

在分佈式環境中，
基於分佈式協議而不是
中央協調器來實現互斥似乎更自然。

- In a distributed environment it seems more natural to implement mutual exclusion, based upon distributed agreement - not on a central coordinator.

假設所有進程都保留一個（Lamport的）邏輯時鐘，該時鐘根據時鐘規則進行更新。

- It is assumed that all processes keep a (Lamport's) logical clock which is updated according to the clock rules.
  - The algorithm requires a total ordering of requests. Requests are ordered according to their global logical timestamps; if timestamps are equal, process identifiers are compared to order them.

該算法需要對請求進行總排序。請求根據其全局邏輯時間戳排序；
如果時間戳相等，則比較進程標識符以對它們進行排序。

- The process that requires entry to a CS multicasts the request message to all other processes competing for the same resource.
  - Process is allowed to enter the CS when all processes have replied to this message.
  - The request message consists of the requesting process' timestamp (logical clock) and its identifier.
- Each process keeps its state with respect to the CS: released, requested, or held.

需要進入CS的進程將請求消息多播給所有其他競爭相同資源的進程。

- 當所有進程都回復了這個消息時，進程才被允許進入CS。
- 請求消息由請求進程的時間戳（邏輯時鐘）及其標識符組成。

每個進程都保持其與CS相關的狀態：已釋放、已請求或已保留。

Ricart-Agrawala Algorithm 1

☞ Rule for process initialization

/* performed by each process P_i at initialization */

[RI1]: $state_{Pi}$:= RELEASED.

☞ Rule for access request to CS

/* performed whenever process P_i requests an access to the CS */

[RA1]: $state_{Pi}$:= REQUESTED.
T_{Pi} := the value of the local logical clock corresponding to this request.

[RA2]: P_i sends a request message to all processes; the message is of the form (T_{Pi}, i), where i is an identifier of P_i.

[RA3]: P_i waits until it has received replies from all other n-1 processes.

☞ Rule for executing the CS

/* performed by P_i after it received the n-1 replies */

[RE1]: $state_{Pi}$:= HELD.
P_i enters the CS.

☞ Rule for handling incoming requests

/* performed by P_i whenever it received a request (T_{Pj}, j) from P_j */

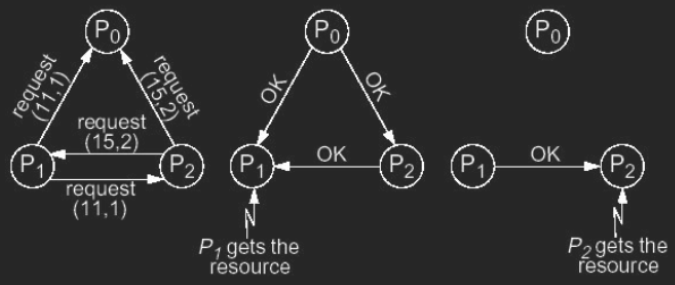
[RH1]: **if** $state_{Pi}$ = HELD **or** (($state_{Pi}$ = REQUESTED) **and** ((T_{Pi}, i) < (T_{Pj}, j))) **then**
Queue the request from P_j without replying.
else
Reply immediately to P_j.
end if.

☞ Rule for releasing a CS

/* performed by P_i after it finished work in a CS */

[RR1]: $state_{Pi}$:= RELEASED.
P_i replies to all queued requests.

Ricart-Agrawala Algorithm1



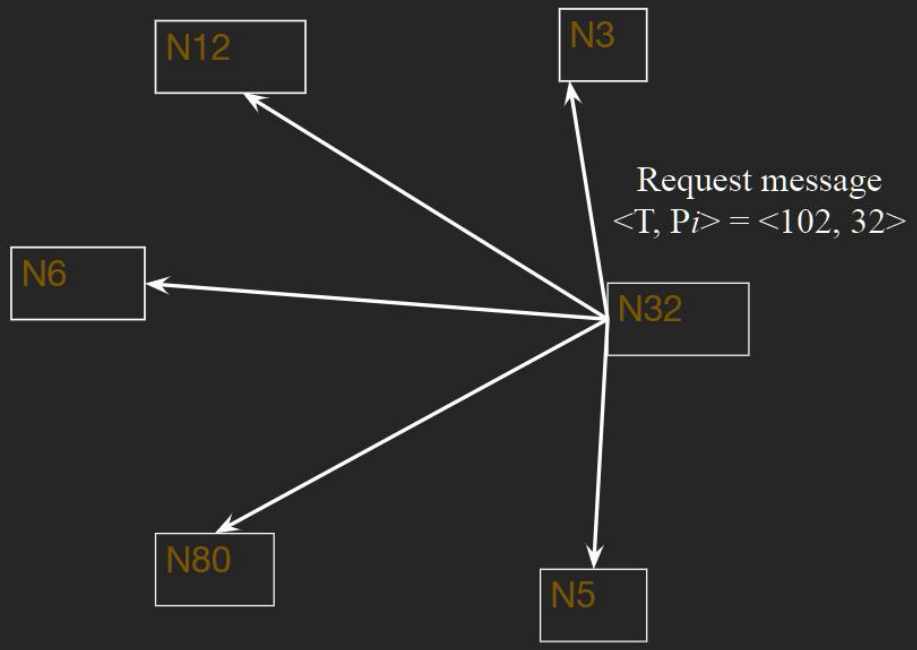
一個進程Pj發出的請求只有在Pi持有時才會被另一個進程Pi阻塞資源，或者如果它請求具有更高優先級的資源（即較小的時間戳）比Pj。

- A request issued by a process Pj is blocked by another process Pi only if Pi is holding the resource or if it is requesting the resource with a higher priority (i.e. smaller timestamp) than Pj.

- Issues 問題
  - Expensive in terms of message traffic.
    - requires 2(n-1) messages for entering the CS; (n-1) requests and (n-1) replies.
  - The failure of any process involved makes progress impossible.
    - Special recovery measures must be taken.

○消息流量昂貴。
■需要2(n-1)條消息才能進入CS：(n-1)請求和(n-1)回复。
○所涉及的任何過程的失敗都會使進展變得不可能。
■必須採取特殊的恢復措施。

Example: Ricart-Agrawala Algorithm1



Example: Ricart-Agrawala Algorithm1