# RPC y Rendezvous

El Pasaje de mensajes se ajusta bien a problemas de filtros y pares que interactúan ya que se usa la comunicación unidireccional. Para resolver C/S la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas) además que cada cliente necesita un canal de respuesta distinto.

**RPC y Rendezvous** son técnicas de comunicación y/o sincronización entre procesos que suponen un canal bidireccional, que son ideales para programar aplicaciones C/S.

RPC y Rendezvous combinan una interfaz tipo monitor con operaciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados)

Difieren en la forma de servir la invocación de operaciones:

* Un enfoque es declarar un procedure para cada operación y crear un nuevo proceso para manejar cada llamado (RPC porque el llamador y el cuerpo pueden estar en distintas máquinas). Para el cliente, durante la ejecución del servicio es como si tuviera en su sitio el proceso remoto que lo sirve.
* El segundo enfoque es hacer Rendezvous con un proceso existente. Un Rendezvous es servido por una sentencia de Entrada (o accept) que espera una invocación, la procesa y devuelve los resultados.

# RPC (procedimientos remotos)

Los programas en RPC están formados por módulos que pueden residir en espacios de direcciones distintos. Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo. Un proceso en un módulo puede comunicarse con procesos de otro módulo solo invocando procedimientos exportados por este. Los módulos tienen especificación e implementación de procedures.

Texto

Descripción generada automáticamente

Funciona algo así como cuando le ponías métodos públicos y privados a un objeto, pero los procedures están repartidos en distintas máquinas.

Los procesos locales son llamados **background** para distinguirlos de las operaciones exportadas.

Texto

Descripción generada automáticamente

La implementación de un llamado Inter módulo es distinta que para uno local, ya que los módulos pueden estar en distintos espacios, un nuevo proceso sirve el llamado y los argumentos son pasados como mensajes entre el llamador y el proceso server.

El llamador demora mientras el proceso servidor ejecuta el cuerpo del procedure implementa opname. Cuando el server vuelve de opname envía los resultados al llamador y termina. Después de recibir los resultados, el llamador sigue.

Si el proceso llamador y el procedure están en el mismo espacio de direcciones es posible evitar crear un nuevo proceso.

En general un llamado será remoto por lo que se debe crear un proceso server o alocarlo en una pool preexistente.

Es como si tuvieses un monitor que sus funciones están desperdigadas en memorias diferentes, sin exclusión mutua explícita ya que ejecutan de forma concurrente (generalmente) y con un proceso local que corre en segundo plano

**Por si mismo RPC es solo un mecanismo de comunicación**

Aunque un proceso llamador y su server sincronizan el único rol del server es actuar en nombre del llamador.

Necesitamos que los procesos en un módulo se sincronicen. Esto comprende exclusión mutua y sincronización por condición, ya que si bien los procesos se sincronizan las variables dentro de un módulo son compartidas, así que vuelven todos los problemas de la primer parte de la materia.

Existen 2 enfoques para proveer sincronización, dependiendo de si los procesos en un módulo se ejecutan con exclusión mutua o concurrentemente.

**Si se ejecutan con exclusión mutua** las variables compartidas son protegidas automáticamente, pero es necesario programar la sincronización por condición.

**Si ejecutan concurrentemente** necesitamos mecanismo para programar exclusión mutua y sincronización por condición (cada módulo es un programa concurrente). Se puede usar cualquier método ya descripto (monitores, semáforos, incluso Rendezvous)

Es más general asumir que los procesos pueden ejecutar concurrentemente. Asumimos que los procesos en un módulo se ejecutan concurrentemente usando por ejemplo time slicing.

Por lo que entiendo, tenés procedures que los usan las aplicaciones que llaman a tu módulo y un Process (o más) que se está ejecutando todo el tiempo en el módulo que es local (background).

**¿Saben que viene después? ASÍ ES, EJEMPLOS OTRA VEZ yay**

Ejemplo cliente servidor

Un módulo que brinda servicios de timing a procesos cliente en otros módulos, tiene 2 operaciones visibles (get\_time y delay(Interval))

Un proceso interno que continuamente inicia un timer por hardware y luego incrementa el tiempo al ocurrir la interrupción de timer.

Código bonito:

Texto

Descripción generada automáticamente

La cola napQ es compartida, así que se usa un semáforo para acceder a ella con exclusión mutua, el semáforo d[id] se usa para despertar al proceso que toca despertar o dormirlo.

Para simplificar, suponemos que min(napQ) devuelve un valor mucho muy grande si la cola está vacía.

Los pedidos get\_time se pueden atender concurrentemente porque solo hay que leer la variable tod.

Ejemplo: Manejo de caches en un servidor distribuido

Hay procesos que se ejecutan en una WS y archivos de datos almacenados en un File Server (FS). Los programas de aplicación quieren acceder a datos del FS, llaman a procedimientos red y write del módulo local FileCache. Leen o escriben arreglos de caracteres.

Los archivos se almacenan en el FileServer en bloques de 1024 bytes fijos. El módulo FS maneja el acceso a bloques del disco, provee los procedimientos ReadBlk y writeBlk.

El módulo FileCache mantiene en caché los bloques recientemente leídos. Al recibir un pedido de read el FileCache primero chequea si los bytes están en caché. Sino llama al procedimiento readblock del FileServer. Algo similar ocurre con los write.

**módulo Filecache:**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Esto es un manejo muy por arriba de como funciona RPC (la concha de tu madre ejemplos de mierda), no de como funciona un sistema de archivos.

Los llamados de los programas de aplicación de las WS son locales a su FileCache, pero desde estos módulos se invocan procesos remotos de FileServer.

Filecache es un server para procesos de aplicación; FileServer es un servidor para múltiples clientes FileCache, uno por WS.

Si existe un FileCache por programa de aplicación no se requiere sincronización interna entre los read y write, porque solo uno puede estar activo. Si múltiples programas de aplicación usarán el mismo FileCache tendríamos que usar semáforos para implementar la EM en el acceso a FileCache.

En cambio, FileServer si requiere sincronización interna porque atiende múltiples FileCache y contiene un proceso DiskDriver.

**módulo File Server:**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Ejemplo: pares que interactúan

Si dos procesos de diferentes módulos deben intercambiar valores, cada módulo debe exportar un procedimiento que el otro módulo llamará.

Texto

Descripción generada automáticamente con confianza media

Es más complejo usar RPC para intercambio de valores que PMA por ejemplo que era una estupidez.

RPC en java

Java soporta RPC en programas distribuidos mediante RMI.

Una aplicación RMI tiene 3 componentes: Una interfaz, una clase server que la implementa y uno o más clientes que llaman a los métodos remotos. El server y los clientes pueden residir en máquinas diferentes.

# Rendezvous

RPC por sí mismo sólo brinda un mecanismo de comunicación intermódulo. Dentro de un módulo es necesario programar la sincronización. Además, a veces son necesarios procesos extra solo para manipular los datos comunicados por medio de RPC, por ejemplo, un merge.

**Rendezvous combina comunicación y sincronización (god no?)**

Como con RPC, un proceso cliente invoca una operación por medio de un call pero esta operación es servida por un proceso existente en lugar de por uno nuevo. Un proceso servidor usa una sentencia de entrada par esperar por un call y actuar y las operaciones se atienden una por vez más que concurrentemente.

Los módulos son procesos comunes, pero su especificación contiene declaraciones de los headers de las operaciones exportadas.

Si un módulo exporta opname, el proceso server en el módulo realiza Rendezvous con un llamador de opname ejecutando una sentencia de entrada.



Las partes entre in y ni se llaman operación guardada. Una sentencia de entrada demora al proceso server hasta que haya al menos un llamado pendiente de opname, luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta S y finalmente retorna los parámetros de resultado al llamador.

Luego, ambos procesos pueden continuar.

A diferencia de RPC el server es un proceso activo, y también se puede combinar comunicación guardada con Rendezvous (el do múltiple, en caso de ADA un SELECT).

Esto es lo que hace realmente útil el Rendezvous para que funcione completamente.

Sintaxis sin ADA:

Texto

Descripción generada automáticamente

La expresión de scheduling se usa para darle prioridad a una comunicación particular, en Rendezvous general no existe.

¿Qué viene ahora????? ASÍ ES, EJEMPLOS

Ejemplo: Buffer limitado

Texto, Carta

Descripción generada automáticamente

El scheduling no hace falta en este caso, si tengo elementos acepto la comunicación retirar, saco el elemento y decremento cantidad, la otra solo se ejecuta cuando el buffer no está lleno.

Ejemplo: Time server

Texto, Carta

Descripción generada automáticamente

La condición delay solo se ejecuta cuando un proceso se debe despertar, y tiene un scheduling para reordenar la cola de espera de acuerdo al que se tenga que despertar, no hago nada, solo lo uso como mecanismo de sincronización por condición.

Ejemplo: Alocador SJN

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

La opción pedir se ejecuta si el procesador está libre, y ordena los pedidos en espera por el que tiene el menor tiempo con la expresión de scheduling.

Esta interesante la verdad.

# ADA – Lenguaje con Rendezvous

Fue desarrollado por el departamento de **defensa de USA** para que sea el estándar de programación de aplicaciones de defensa. {America, fuck yeah}

Desde el punto de vista de la concurrencia, un programa ADA tiene tasks que pueden ejecutar independientemente y contienen primitivas de sincronización.

Una tarea puede decidir si acepta la comunicación mediante la primitiva accept. Se puede declarar un type task y luego crear instancias de procesos identificado con dicho tipo (arreglo, puntero, instancia simple). Sirve para definir múltiples tareas idénticas como por ejemplo clientes en un servidor.

Texto

Descripción generada automáticamente con confianza media

Una especificación de TASK define una única tarea. Una instancia del correspondiente task body se crea en el bloque en el cual se declara el TASK.

El Rendezvous es el principal mecanismo de comunicación de ADA pero tiene otros que no vamos a usar.

Sentencia entry

**Entry:** Declaración de entry simples y familia de entry (el último no lo vamos a ver).

Declaración y funciones:

Texto

Descripción generada automáticamente

Sentencia de entrada accept

La tarea que declara un entry sirve para llamados al entry con accept.

**Accept** nombre (parámetros formales) **do** sentencias **end nombre;**

Demora la tarea hasta que haya una invocación, copia los parámetros reales en parámetros formales y ejecuta las sentencias. Cuando termina, los parámetros formales de salida son copiados a los parámetros reales. Luego ambos procesos continúan.

La **sentencia wait selectiva** soporta comunicación guardada.

Texto

Descripción generada automáticamente

Cada línea se llama alternativa, las cláusulas when son opcionales. Puede contener una alternativa else, or delay, or terminate.

Uso de atributos del entry: count, calleable

Un accept sin cuerpo es:

ACCEPT(parámetros);

Ni más ni menos, mucho muy importante para los exámenes.

El select de entry es para comunicarte o hacer otra cosa, en cambio el select del accept es para comunicación guardada.

Después de esto, vienen los EJEMPLOS SIIIIIIIIIIII

Hay un ejemplo de mailbox para 1 mensaje y uno de lectores/escritores pero no los voy a incluir porque son para entender la sintaxis.

Ejemplo: Mailbox para N mensajes (buffer limitado)

Arriba en el resumen está la solución con Rendezvous general para comparar.

Solución con ADA (principalmente es un cambio de sintaxis)

Texto, Carta

Descripción generada automáticamente

Caso con un arreglo:

Texto, Carta

Descripción generada automáticamente

Ejemplo: Time server

El problema principal es que no hay scheduling comparado con Rendezvous general.

La solución es más o menos así:

Texto

Descripción generada automáticamente

Cada cliente recibe su identificación para saber quien es.

El body del TimeServer:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Doy prioridad a tick usando el count, porque siempre debo priorizar los ticks del reloj antes que aceptar un pedido que se quiere dormir. El que no tiene condición es el que tiene mas prioridad. Ese accept tick solo acepta el pedido, las sentencias de eso van fuera del accept para no ir demorando el tiempo con la comunicación.

Ejemplo: Alocador SJN

Mismo problema que el anterior, no tenemos scheduling. Así que la solución es parecida a time server

Texto, Correo electrónico

Descripción generada automáticamente

Cuerpo del alocador

Texto, Carta

Descripción generada automáticamente

Si el recurso está libre me guardo quien es la persona para avisarle al salir de la comunicación y marcó que está ocupado, si no está libre lo guardo en la colaOrdenada por tiempo.

Cuando me mandan un liberar si la cola está vacía marco libre en true, sino saco e l siguiente de la cola guardando quien es para avisarle después.

Si hay alguien para usar el procesador le mando un usar.