

CC51C Comunicación de Datos Capa Aplicación

1 Modelo Cliente-Servidor

El primer problema con el que se debe lidiar al querer comunicar dos procesos es la sincronización (*rendez-vous*). Suponer por ejemplo que existen dos programas que desean comunicarse en forma síncrona a través de un pipe. La mejor forma de lograr la sincronización es que uno de los dos programas espere hasta que llegue algún indicio de que existe otro programa que se quiere comunicar con él. En el modelo cliente-servidor, el servidor siempre está esperando una conexión (inicia la conexión en forma pasiva) y el cliente inicia la conexión en forma activa.

En general, un servidor debe ser capaz de atender simultáneamente a varios clientes, por esto son en general más complejos.

2 Modelo de Sockets

Nacieron en Berkeley, extienden el concepto de file descriptors para usarlos en conexiones remotas. Para efectos de programación, una conexión es idéntica a un pipe bidireccional, sólo cambia la forma de “abrir” el filedescriptor. Las funciones para manipular estos filedescriptors son las siguientes:

socket	crea un descriptor para usarlo en Tx sobre redes. Toma como parámetro la familia de protocolos, y el tipo de servicio (stream o datagram en el caso de TCP/IP).
connect	establece una conexión activa, recibe como parámetro la dirección y puerto de destino.
write	generalmente copia los datos a un buffer y los envía a medida que puede. Si los buffers del S.O. están llenos, se bloquea.
read	lee de la conexión, se bloquea si no hay datos, o entrega <u>a lo más</u> length datos (length es un parámetro a la función). En UDP, si hay más datos que length en el datagrama, el resto se pierde, ya que no tiene sentido hacer otro read si no existe el concepto de conexión.
bind	especifica dirección (número IP + puerto local) al cual se asocia el socket.
listen	pone el socket en modo pasivo, y setea el número máximo de conexiones que se encolarán (cuando llegan conexiones simultáneas).
close	termina la conexión y libera el socket. Si es un socket compartido, ref_count - ;
shutdown	Termina la conexión TCP/IP en una o ambas direcciones.
getpeername	retorna dirección remota del socket.
getsockopt	ver opciones del socket.
setsockopt	cambiar opciones del socket.

3 Envío-recepción de un archivo en Java

3.1 Servidor: ArchRecibidor

```
import java.io.*;
import java.net.*;

//traspasando archivos (Servidor-Recibidor)

public class ArchRecibidor {
    public static void main(String[] args) throws IOException {

        Socket cs = null;
        ServerSocket ss = new ServerSocket(4444);
        FileOutputStream out = null;
        InputStream in = null;

        System.out.println("esperando que alguien entre");
        cs = ss.accept();
        in = cs.getInputStream();

        // abrir archivo de escritura
        out = new FileOutputStream("yyy");
        System.out.println("escribiendo");
        int b; long i=0;
        byte ab[] = new byte[100];
        while ((b= in.read(ab,0,100) ) != -1) {
            out.write(ab,0,b);
            System.out.println(++i);
        }

        out.close();
        in.close();
        cs.close(); ss.close();
        System.out.println("listo");
    }
}
```

3.2 Cliente: ArchEnviador

```
import java.io.*;
import java.net.*;

//traspasando archivos (Cliente-Enviador)

public class ArchEnviador {
```

```
public static void main(String[] args) throws IOException {

    Socket echoSocket = null;
    OutputStream out = null;
    InputStreamReader in = null;

    try {
        //abrir el socket donde se transmitira
        echoSocket = new Socket(args[0], 4444);

        //ponerlo para escribir
        out = echoSocket.getOutputStream();

        //abrir el archivo que se va a leer
        in = new InputStreamReader(
            new FileInputStream(args[1]));
    } catch (IOException e) {
        System.err.println("Couldn't get I/O ");
        System.exit(1);
    }

    int b;
    while ((b = in.read()) != -1) {
        out.write(b);
    }

    out.close();
    in.close();
    echoSocket.close();
}
}
```

4 Aplicaciones distribuidas

Primero, tenemos que aclarar qué es una aplicación distribuida, básicamente entendiendo la diferencia entre sistema distribuido y red de computadores. En el caso de una red de computadores, el usuario hace uso de la red explícitamente, generando comandos que usan los recursos de la red (login, ftp, http, etc). En cambio, en un sistema distribuido, el usuario probablemente ni se entera de que el sistema operativo está haciendo uso de la red, o sea que el uso de la red es totalmente transparente desde el punto de vista del usuario (ejemplos: una base de datos distribuida, NFS).

En general, vemos que hay 2 formas de programar aplicaciones distribuidas:

1. diseño orientado a la comunicación: se comienza con el protocolo de comunicación, diseñando el formato de los mensajes y la sintaxis. Diseño de cliente

y servidor especificando cómo reaccionan frente a mensajes entrantes y cómo generan mensajes salientes.

2. diseño orientado a la aplicación: se comienza con la aplicación, como si fuera una aplicación local. Una vez que se tenga ese sistema operativo, se concentra el esfuerzo en dividir el programa en varias partes, y ejecutar esas partes en computadores/procesadores separados.

Problemas del diseño orientado a la comunicación:

1. pueden pasar inadvertidos detalles de la aplicación al darle toda la importancia al protocolo, y encontrarse con que el protocolo finalmente no provee toda la funcionalidad necesaria.
2. el diseño de protocolos es muy complejo, y hay situaciones poco claras, que pueden llevar a deadlocks, o a fallas no esperadas (Heisenbugs). También puede sufrir la eficiencia de la programación.
3. como hay concentración en la comunicación, ésta se vuelve esencial para el funcionamiento del programa, produciendo código difícil de entender y modificar posteriormente.

4.1 Principio o modelo de RPC

El programador usa el diseño orientado a la aplicación, y una vez que se tiene un programa funcionando localmente, se le pueden hacer cambios menores (según el caso no se requiere ni siquiera recompilar el programa) para hacer ese programa distribuido. Así el programador puede seguir buenos principios de diseño y generar código modular y fácil de mantener.

El cliente cuenta con una declaración de la interfaz que provee el servidor, y la implementación puede constar, indistinguiblemente para el cliente, de:

- una implementación (procesamiento de los datos) local
- – una implementación local que simplemente envía los datos a un servidor (stub)
 - procesamiento de los datos (y por ende implementación de la función) en el servidor
 - envío de vuelta al cliente

En resumen: primero se solucionan los problemas propios de la aplicación y luego se aplica el programa a un ambiente distribuido.

El modelo de RPC se puede entender como una extensión del modelo procedural típico, ampliando los límites dentro de los cuales se pueden hacer llamados, para invocar procedimientos en otras máquinas. Pero también se tienen algunas restricciones adicionales, como lo es el no poder enviar punteros ni descriptores de archivos entre computadores (se pueden enviar, pero no tienen sentido). También aparecerá el problema de enviar estructuras de datos complejas entre dos máquinas que comparten datos

(cómo se envía un árbol o una lista circular son problemas comunes y no triviales). Para simplificar el concepto, podemos suponer que en un RPC se pasa un sólo argumento al procedimiento, y se recibe un sólo argumento de vuelta una vez que retorna el procedimiento. Esos argumentos serán un struct en lenguaje C, que pueden contener varios argumentos empaquetados. Esta simplificación tiene por objeto acercar la teoría con la realidad, ya que se enviará un stream TCP o UDP de ida al invocar el procedimiento, y un stream de vuelta con el resultado de la operación.

Al ejecutar remotamente un procedimiento, el thread de ejecución se transfiere de un computador a otro, o sea que no hay paralelismo, el proceso que invoca el procedimiento remoto se queda bloqueado hasta que éste retorne. Esto hace que un RPC sea varias órdenes de magnitud más costoso que una invocación local, según el estado de la red.

Existe otra diferencia más entre RPC e invocaciones locales. Como habíamos visto en el paradigma de cliente-servidor, el protocolo TCP/IP (y la mayoría de los protocolos de red) no provee ningún mecanismo de ejecución remota de procedimientos (problema de rendez-vous), así que debe existir previamente un método para recibir el requerimiento, algo así como un servidor que esté atento a las peticiones entrantes.

Otro detalle importante es la representación de los datos. Para que los mismos datos puedan ser procesados en distintas arquitecturas, es necesario tener algún tipo de XDR (eXternal Data Representation) que estandariza la forma en la cual los datos viajan por la red.

4.2 SUN RPC

Es la implementación más conocida del modelo, usada en el sistema NFS, YP/NIS, entre otros. Se usa un entero de 32 bits para identificar el programa remoto que proveerá el procedimiento a ejecutar. Además se usa otro entero para identificar el procedimiento dentro del programa. Conceptualmente, un procedimiento remoto queda identificado dentro de un computador con el par (programa, procedimiento). Para permitir cambios en programas en forma transparente y gradual, también se incluye un número de versión, permitiendo así la coexistencia de varias versiones de un mismo programa (ej: NIS v/s NIS+). Entonces, el identificador queda: (programa, version, procedimiento).

Cuando una máquina ha recibido un requerimiento para un procedimiento dentro de cierto programa, y recibe otro requerimiento concurrente para el mismo programa (aunque sea para otro procedimiento), el requerimiento entrante será ejecutado sólo cuando el primero haya finalizado. Así es más fácil poder garantizar la consistencia de sistemas donde es crítico el orden en que se realizan las operaciones.

La implementación de RPC de Sun permite elegir el protocolo a utilizar: TCP o UDP. Esto se aleja algo del modelo, que implica el uso de un protocolo confiable. Dado que se puede usar UDP, pueden perderse o duplicarse paquetes en el camino, lo que tiene básicamente 2 implicaciones: - puede ser que el procedimiento invocado no se haya ejecutado - el procedimiento pudo ser ejecutado más de una vez. Al ejecutar un procedimiento remoto, se sabe que será ejecutado cero o más veces. Y si retorna exitosamente, se sabe que fue ejecutado al menos una vez. Es por esto que cada invocación a un procedimiento remoto debe ser idempotente, por ejemplo, un procedimiento que

agrega datos al final de un archivo (append) no es idempotente. En cambio, un procedimiento que escribe datos en algún sector específico de un archivo sí es idempotente.

4.2.1 Retransmisión en RPC

La biblioteca que provee Sun tienen timeouts seteables pero no adaptivos (no se adaptan al estado de la red), y después de una cierta cantidad de intentos deciden que no es posible ejecutar el procedimiento, y retornan un error. Cabe hacer notar eso sí que el fallo de ejecutar el procedimiento no significa que el procedimiento no se ejecutó.

Mapeo de programas remotos a puertos de protocolo.

UDP y TCP usan números de 16 bits para identificar sus puertos. Para que ambas partes de una ejecución de procedimientos remotos se puedan comunicar, deben saber qué puerto deben acceder. Como RPC usa 32 bits para identificar los programas, pueden existir más programas RPC que puertos UDP o TCP. Para evitar usar demasiados puertos UDP o TCP, se reservan puertos solamente por el tiempo que se van a usar (cuando parte el servidor, pide un puerto al S.O., y con ese se queda hasta que termine de ejecutar, lo mismo el cliente). Para que se puedan poner de acuerdo, existe un proceso llamado Port Mapper, que mantiene una tabla con los mapeos dinámicos de todos los programas que proveen RPC.

4.2.2 Algoritmo del RPC Port Mapper

- El portmapper crea un socket pasivo asociado al puerto 111 (TCP y UDP)
- repetidamente acepta requerimientos para registrar un nuevo par (numero_programa_rpc, puerto protocolo) para programas de la misma máquina, o requerimientos para buscar el puerto de protocolo para un programa

4.3 RMI: Remote Method Invocation (Java)

RMI es la versión Java (y por lo tanto orientada al objeto) de implementar RPC. Para ello, se proveen las siguientes funcionalidades:

1. localización de objetos remotos
2. comunicación con objetos remotos
3. traspaso de código a objetos remotos

Existe un equivalente al portmapper de Sun RPC, que se llama `rmiregistry`. Este corre en el servidor (el host donde se publica el objeto).

4.3.1 Ejemplo: Interfaz

```
import java.rmi.*;

public interface Numero extends Remote {
    public int Numero() throws RemoteException;
}
```

La interfaz debe contener toda la información necesaria para que el cliente implemente su parte. En el caso del ejemplo, lo único que el cliente sabe es que existe un método `Numero` que devuelve un entero y no recibe argumentos. No interesa el dónde se ejecuta ni si la clase que implementa el método `Numero` además implementa otros métodos.

4.3.2 Ejemplo: Servidor

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class NumeroImpl extends UnicastRemoteObject implements Numero {
    int cont = 0;

    public int getNumero() throws RemoteException {
        int ret = cont++;
        return ret;
    }

    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            NumeroImpl n = new NumeroImpl();
            Naming.rebind("//"+args[0]+"/elNumero",n);
            System.out.println("Numero creado");
        } catch (Exception e) {}
    }
}
```

En el ejemplo se implementa la interfaz, y se publica el objeto con un nombre simbólico, el cual puede ser luego usado por el cliente para obtener una referencia al objeto remoto. La publicación se hace con el método `Naming.rebind`, mediante el cual se le entrega la información al `rmiregistry` para que pueda entregar referencias remotas al objeto creado.

4.3.3 Ejemplo: Cliente

```
import java.rmi.*;
import java.rmi.server.*;

class ClienteNumero {
    public static void main(String args[]) {
        try {
            Numero N = (Numero)
                Naming.lookup("//"+args[0]+"/elNumero");
            System.out.println("El numero vale ahora"+N.getNumero());
        } catch( Exception e) {}
    }
}
```

}

5 DNS (RFC 1035)

5.1 Antecedentes

En 1970, ARPAnet consistía en pocos cientos de computadores. Toda la información necesaria se almacenaba en un solo archivo (HOSTS.TXT). Los cambios se enviaban por mail y el HOSTS.TXT se podía bajar por ftp. La idea es básicamente mapear nombres de archivos a direcciones.

Con el aumento explosivo de hosts, surgieron varios problemas:

- tráfico y carga (hoy en día (2001), la zona .com. pesa alrededor de 2.6 GB)
- colisiones de nombres: no existía una autoridad que pudiera garantizar unicidad en los nombres de dominio ni resolver conflictos.
- consistencia: cuando todos los hosts actualizaban el nuevo HOSTS.TXT, éste ya había cambiado nuevamente.

Por esto se creó una base de datos distribuida llamada Domain Name System (DNS).

5.2 Estructura

- raíz
- parecido a sistema de archivos (árbol jerárquico) (altura máxima: 127)
- dominios, subdominios, subsubdominios, etc. (cada nombre tiene un máximo de 63 caracteres)
- ¿relación entre hosts de un mismo subdominio?: arbitraria (ej: geográfica, administrativa, orden alfabético, etc...)
- Zonas: Una zona contiene información sobre una parte (puede ser un dominio o bien varios) del espacio de nombres completo.
- qué datos contiene el DNS?: Resource Records
 - cada RR tiene un formato específico
- ¿cómo descentralizar?: Se delega una subárbol a otra administración.

5.3 Resolución de nombres

- UDP
- resolución recursiva v/s iterativa
- resolución reversa: in-addr.arpa.
 - ej: reverso para 146.83.4.11 se consulta como: 11.4.83.146.in-addr.arpa
- cache, TTL

5.4 Configuración

- master/slaves (primario/secundarios): tienen autoridad sobre una zona
- transferencias de zonas: TCP
- tipos de RRs
 - SOA (domain, admin, serial, refresh, retry, expire, minimum ttl)
 - A
 - NS
 - PTR
 - MX (menor significa mayor prioridad)
 - CNAME (se prohíbe cualquier otro RR)
 - TXT, HINFO, etc.
- otras zonas
 - stub
 - stealth
 - forwarder

5.5 Seguridad

- cache poisoning
- denial of service
- spoofing
- glue fetching

5.6 Características avanzadas

- Dynamic update (RFC 3007)
- notify (RFC 1996)
- Incremental Zone Transfer (IXFR) (RFC 1995): estilo diff-patch. Se envían solamente cambios a la zona anterior (líneas agregadas, líneas eliminadas. Una modificación se envía como una eliminación de la línea antigua seguida de agregar la línea nueva).
- DNS security extensions (DNSSEC) (RFC 2535, 3008)
 - servicios:
 - * distribución de llaves (mediante el nuevo RR KEY)
 - * autenticación de origen de datos (mediante el nuevo RR SIG)
 - * autenticación de respuestas (ver SIG(0))
 - autenticar no-existencia de datos: NXT RR, ordenar.
 - la zona padre debe firmar la llave de las zonas hijo
 - un CNAME necesita un RR, pero un CNAME no debe tener más RRs
 - autenticación de respuestas: el cliente puede estar seguro que recibe respuesta del servidor a quien preguntó y que la respuesta corresponde a la consulta.
- TSIG (Transaction Signature) (RFC 2845) La idea es autenticar clientes y/o servidores (Ej: clientes autorizados para hacer dynamic updates). Estos clientes/servidores se conocen a priori. Es más simple que implementar DNSSEC.
- SIG(0) (RFC 2931): cambios a la autenticación definida en RFC 2535. Permite autenticar transacciones usando los RR KEY y demases, sin necesidad de que se comparta un secreto entre cliente y servidor.
- IPv6 (RFC 1886):
 - AAAA: para las direcciones de 128 bits en vez de 32
 - IP6.INT: reversos, ej: 4321:0:1:2:3:4:567:89ab -> b.a.9.8.7.6.5.0.4.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.0.0.0.1.2.3.4.IP6.INT.