.UBA fiuba FACULTAD DE INGENIERÍA

Taller de Programación I

Arquitectura Cliente-Servidor y Protocolo HTTP

Ing. Pablo A. Deymonnaz - Uriel Kelman

pdeymon@fi.uba.ar - urielkelman@gmail.com

Facultad de Ingeniería

Universidad de Buenos Aires

Índice

1. Arquitectura Cliente-Servidor
2. Comunicación
3. Protocolo HTTP

¿Qué es un sistema distribuido?

- Un sistema distribuido es aquel en donde los componentes de software o hardware que lo conforman se encuentran conectados a mediante una red (LAN, internet, etc.), a través de la cual pueden comunicarse enviándose mensajes para coordinar sus acciones y cumplir algún objetivo específico.
- Cuando hablamos de arquitectura, nos referimos a la forma en la que los componentes son organizados y cómo se relacionan entre sí (cómo se van a comunicar, cuántos tipos distintos existirán, dónde estarán ubicados, etc.).

Paradigmas de comunicación

La comunicación entre componentes es uno de los aspectos más importantes de los sistemas distribuidos. Puede darse de distintas formas:

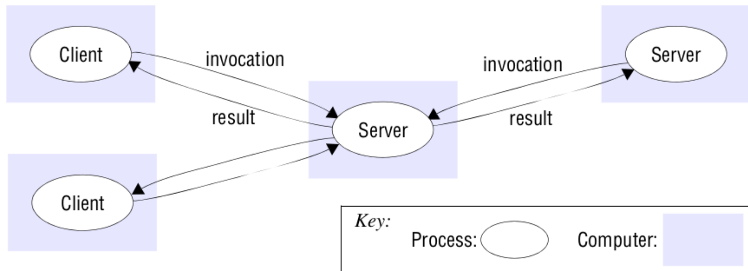
- Entre procesos: A través de APIs de bajo nivel (*channels, sockets, etc.*)
- Invocaciones remotas: Comunicación entre dos componentes del sistema a través de la invocación de una operación remota.
- En forma indirecta: A través de una tercera entidad. Los componentes no necesitan conocerse entre sí, y tampoco deben existir al mismo tiempo.

Cliente-Servidor(1)

Es la arquitectura más común en los sistemas distribuidos. Se distinguen dos roles para sus compoenentes:

- Servidor: Es un proceso que corre en una computadora que implementa un servicio específico. Este servicio es accedido y utilizado por los clientes.
- Cliente: Es un proceso que solicita un servicio al servidor. Para esto, debe establecer una conexión con el servidor, enviarle un mensaje indicando qué servicio quiere ejecutar y esperar por la respuesta.

Cliente-Servidor(2)



UBAfiuba FACULTAD DE INGENIERÍA

Cliente-Servidor(3)

Cuando diseñamos un servidor, se deben tener en cuenta algunos aspectos de diseños:

- Si será iterativo o concurrente (¿es el mismo proceso servidor el que atiende la solicitud, o la solicitud se resuelve en un proceso o *thread* diferente?)
- ¿Cuáles serán los endpoints del servidor? Los endpoints son los puertos abiertos donde el proceso servidor se encuentra escuchando por solicitudes.
- *Stateful* vs *Stateless*: ¿El servidor almacena algún tipo de información de los clientes?

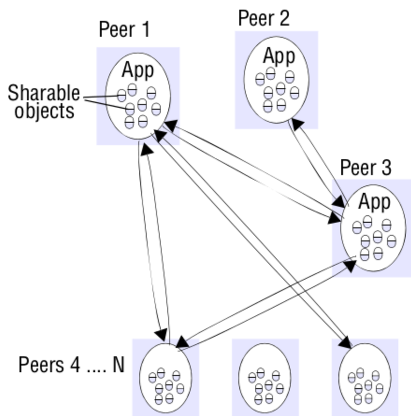
Cliente-Servidor(4)

Algunas claves acerca de la arquitectura son:

- El servidor tiene un rol pasivo (simplemente espera), mientras que el cliente tiene un rol activo (es quien inicia la comunicación enviando pedidos al servidor).
- Los clientes deben conocer la ubicación del servidor para poder localizarlo y enviarle pedidos.
- Los clientes no entablan ninguna comunicación entre ellos (no se conocen).
- Permite centralizar la toma de decisiones en una aplicación distribuida.
- Se suele asumir que los servidores tienen más capacidad de *hardware* que los clientes.

Peer-to-peer

Otro tipo de arquitectura es la conocidad como peer-to-peer. Es la utilizada en protocolos como BitTorrent o Napster.



Índice

1. Arquitectura Cliente-Servidor

2. Comunicación

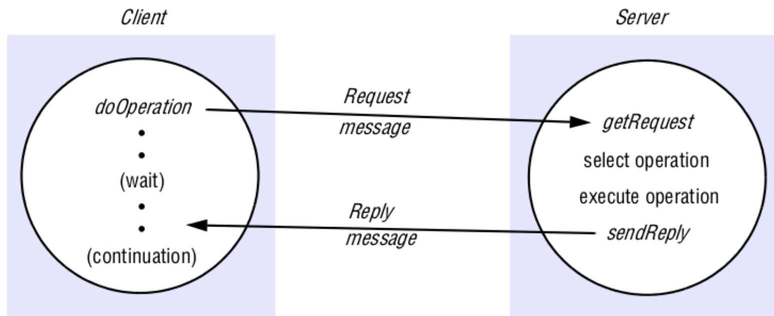
3. Protocolo HTTP

Cliente-Servidor: Patrón de comunicación (1)

El patrón de comunicación para esta arquitectura se conoce como *request-reply*. Sus características principales son:

- Habitualmente, es sincrónico. Esto quiere decir que el proceso cliente se bloquea esperando la respuesta del servidor.
- Se puede modelar a partir de la existencia de tres primitivas:
  - *doOperation*: Lo ejecutan los clientes, realizando una invocación al servicio remoto que se encuentra en el servidor. Dentro de su *scope* deben encontrarse los datos para acceder al servidor (ip, puerto) y los argumentos que recibe el servicio.
  - *getRequest*: Se ejecuta en el servidor, y permite recibir solicitudes de un cliente.
  - *sendReply*: La ejecuta el servidor una vez que ya tiene el resultado de la invocación, enviándolo al cliente.

Cliente-Servidor: Patrón de comunicación (2)



Cliente-Servidor: La importancia del protocolo TCP(1)

Al implementar la arquitectura Cliente-Servidor para una aplicación, debemos tener en cuenta que podrían existir una serie de fallas que no existen para aplicaciones no distribuidas:

- En la red que comunica al cliente con el servidor que deriven en la pérdida de paquetes correspondientes a solicitudes o respuestas, generando una falla en la comunicación.
- Mensajes que no son entregados entre nodos en el orden que deberían.

Cliente-Servidor: La importancia del protocolo TCP(2)

Utilizar el protocolo TCP para la comunicación entre nuestros componentes nos provee de ciertas garantías deseables para implementar la arquitectura.

- Tanto los mensajes de *request* como los mensajes de *reply* se entregan en forma confiable:
  - Si se pierde un paquete mientras se transmite en la red, el protocolo se encarga de realizar la retransmisión.
  - Si los paquetes llegan hacia el destino en desorden, el protocolo se encarga de que la aplicación reciba los mensajes en el mismo orden en el que se mandaron.

Marshalling y *Unmarshaling*(1)

Cuando queremos intercambiar datos entre un cliente y un servidor, debemos tener en cuenta el siguiente problema que puede surgir del intercambio:

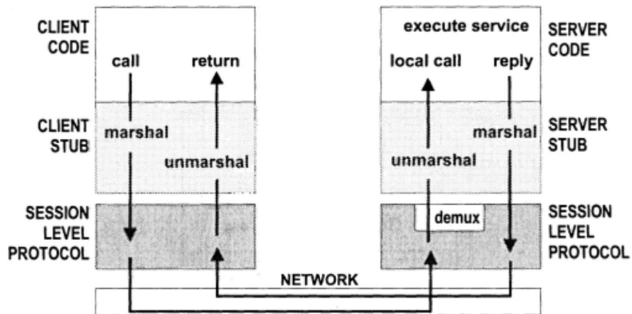
- Los programas almacenan la información en variables y estructuras de datos de diversos tipos.
- Cuando queremos transmitir información, esta debe ser convertida a una secuencia de *bytes* y reconstruida en el destino.
- Las computadoras almacenan valores primitivos de distintas formas (*big-endian* vs *little-endian*, *floating point*, *ASCII* vs *unicode*, etc.)
- Conclusión: Debe existir un estándar de representación de datos que permita intercambiarlos a pesar de las diferentes forma de representación que existan en el cliente y el servidor.

Marshalling y *Unmarshalling* (2)

A partir de esta necesidad de estandarizar la representación de los datos en el intercambio, surgen dos definiciones:

- *Marshalling*: Es el proceso en el cual los datos que se quieren enviar en un *request* o un *reply* se transforman a una forma estándar que permita transmitirlos en un mensaje.
- *Unmarshalling*: Es el proceso en el cual, ante el arribo de un *request* o un *reply*, se desempaquetan los datos repoduciendo valores similares antes del momento que fueran transformados.

Algunos ejemplos de estándares para representar datos son: JSON, XML, etc.


CLIENT CODE
call
return
CLIENT STUB
marshal
unmarshal
SESSION LEVEL PROTOCOL
execute service
local call
reply
SERVER CODE
marshal
unmarshal
SERVER STUB
demux
SESSION LEVEL PROTOCOL
NETWORK

Índice

1. Arquitectura Cliente-Servidor
2. Comunicación
3. Protocolo HTTP

HTTP: *Hyper-Text Transfer Protocol*

HTTP es uno de los protocolos de comunicación más difundidos y utilizados. Algunas de sus características son:

- Es un protocolo de capa de aplicación. Esto significa que se monta sobre la capa de transporte, específicamente sobre el protocolo TCP.
- Es el protocolo que utilizan por default los navegadores web para solicitar recursos a web servers.
- Se utilizan URLs (*Uniform Resource Locator*) para localizar a los recursos en los servidores.
- Es un protocolo de tipo *request-reply*,

HTTP: Funcionamiento

En líneas generales, el funcionamiento del protocolo puede ser descrito a partir de los siguientes pasos:

- El cliente solicita establecer una conexión que debe ser aceptada por el servidor, utilizando URL y un puerto.
- El cliente envía la solicitud al servidor.
- El servidor resuelve la solicitud y envía la respuesta al cliente.
- La conexión es cerrada.

Esta es una aproximación. Crear y destruir conexiones para cada solicitud es caro, por lo tanto se suelen utilizar conexiones persistentes.

HTTP: Representación de los datos

- Las solicitudes y las respuestas son representadas en textos ASCII. Algunos recursos como imágenes pueden ser representados como una secuencia de bytes, incluso pueden ser comprimidos.
- Los recursos son representados con estructuras MIME (*Multiporpuse Internet Mail Extensions*). Cada dato tiene un prefijo que indican el tipo MIME, para que el receptor del mismo sepa como procesarlo. Algunos ejemplos de tipos MIME son:
  - *text/plain*
  - *text/html*
  - *image/gif*
  - *image/jpeg*

HTTP: Solicitudes

Una solicitud HTTP tiene la siguiente estructura:

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

- *Method*: Especifica el método (la acción) a realizar.
- *URL*: Es la dirección para localizar al recurso dentro del servidor.
- *HTTP Version*
- *Headers*: Opcional. Permiten enviar información (en general, parámetros) junto a la solicitud.
- *Message Body*: Opcional. Permiten enviar un cuerpo de texto (por ejemplo un documento JSON) en la solicitud.

HTTP: Métodos (1)

Los métodos más utilizados del protocolo son los siguientes:

- GET: Solicita el recurso indicado en la URL. Dentro de la URL se pueden especificar parámetros adicionales.
- POST: Se utiliza para dar de alta un recurso en el servidor. El recurso debe ser inexistente. Los datos asociados al recurso son enviados en el *body* de la solicitud.
- PUT: Solicita al servidor la creación o modificación de un recurso.
- DELETE: Se utiliza para eliminar el recurso con la URL especificada.

HTTP: Métodos (2)

Los siguientes métodos son secundarios y se utilizan con menos frecuencia:

- PATCH: Se utiliza para aplicar modificaciones parciales a un recurso.
- TRACE: Retorna la solicitud. Se puede utilizar para hacer algún diagnóstico.
- OPTIONS: Sirve para obtener información acerca de las operaciones (métodos) que se pueden aplicar a un recurso.

HTTP: Respuestas

Una respuesta HTTP tiene la siguiente estructura:

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

- Version
- *Status code*: Código que indica el estado de la solicitud.
- *Reason*: La razón que explica el resultado (para distintos códigos, existirán distintas razones).
- *Headers*: Similar a los headers de las solicitudes.
- *Message Body*: Datos asociados al recurso solicitado.

Status Codes

Los códigos que indican el resultado de una petición HTTP se agrupan en familias por centenas:

- 200-299: Agrupan las respuestas satisfactorias.
- 300-399: Indican que se produjo una redirección a otro recurso.
- 400-499: Indican que la solicitud del cliente tiene un error (URL inexistente, parámetros incorrectos, etc).
- 500-599: Indican que se produjo un error en el servidor mientras se procesaba la solicitud.

Ver descripción de los códigos en: HTTP Status Codes

Bibliografía

- George Colouris, Jean Dollimore, Tim Kindberg, Gordon Blair: *Distributed Systemas: Concepts and Desing*, Fifth Edition, 2012.
- Andrew Tanenbaum, Maarten Van Steen: *Distributed Systems*, Third Edition, 2018.