



Introducción a Sistemas Distribuidos

Virginia Padilla

Copyright © 2023 Virginia Padilla

PUBLICADO POR VIRGINIA PADILLA

BOOK-WEBSITE.COM

Licensed under the Creative Commons Attribution-NonCommercial 4.0 License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <https://creativecommons.org/licenses/by-nc-sa/4.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Primera Impresión, Sep 2023



Prefacio

El propósito de este documento es proporcionar un texto guía al curso Sistema Distribuido que se imparte en la Universidad Nacional Experimental de Guayana, UNEG.

Este libro electrónico se ha alimentado con las notas de clase que uso para dictar este curso. Estos apuntes, los he construido revisando y tomando notas de textos actualizados asociados al tema. Cualquier error u omisión es solo atribuible a mi persona. En caso de que el lector tenga sugerencias o requiera reportar errores, puede escribirme al correo virginiapadillas@gmail.com. Les agradezco profundamente sus comentarios o sugerencias.

Este libro consta de 10 capítulos que he agrupado en cuatro partes. La primera parte, Conceptos y Arquitecturas se hace una presentación de los Sistemas Distribuidos y sus características; luego se aborda las arquitecturas de este tipo de sistemas. La segunda parte nombrada Comunicación entre Procesos, trata acerca de los procesos en los Sistemas Distribuidos y los protocolos de comunicación entre procesos. En Comunicación Indirecta, tercera parte de este texto, presenta distintas arquitecturas relacionadas con el tema como grupos, pub-sub, y sistemas P2P. Finalmente, en la cuarta parte del libro, Coordinación y Replicación se presenta los temas de Sincronización de Sistemas Distribuidos, Algoritmos de Consenso y Replicación.

Cada tema está acompañado de una sección titulada Casos de uso donde se presentan ejemplos de Sistemas Distribuidos o se amplían temas que se han considerado relevantes. Adicionalmente, se ha construido un glosario de términos destacados sobre el área de estudio.

La escritura de este documento se hizo usando el sistema tipográfico $\text{\LaTeX}$ usando la clase `textsc` [The legrand orange book](#).

Las figuras de las portadas y encabezados de capítulos fueron elaborados por Andrea Isabel Revilla. Pueden ver sus trabajos en Instagram: [@ave.rebel](#) y en la página de Art-Station [Avebel](#). Por otra parte, las figuras que acompañan los capítulos fueron elaboradas por Ana Virginia Revilla. Agradezco a ambas su disposición y el tiempo invertido en ello. Gracias, gracias.



Contenido

Prefacio	2
-----------------------	----------

I

Conceptos y Arquitecturas

1	Introducción a los Sistemas Distribuidos	19
1.1	Sistemas Distribuidos vs Centralizados	20
1.2	Principios de Diseño	22
1.3	Teorema CAP	24
1.4	Manejo de Fallas	30
1.5	Falacias de los sistemas distribuidos	32
1.6	Caso de Estudio: Tecnología Web	34
1.6.1	HTTP	34
1.6.2	HTML	38
2	Arquitectura de los Sistemas Distribuidos	47
2.1	Modelos Físicos	47
2.2	Modelos Arquitectónicos	49
2.2.1	Entidades	50
2.2.2	Paradigma de comunicación	54
2.2.3	Roles y responsabilidades	56
2.2.4	Correspondencia con la infraestructura física distribuida	57
2.3	Soluciones basadas en middleware	65
2.4	Caso de Estudio: Sistema de Nombres de Dominio	66
2.4.1	Términos asociados al DNS	67

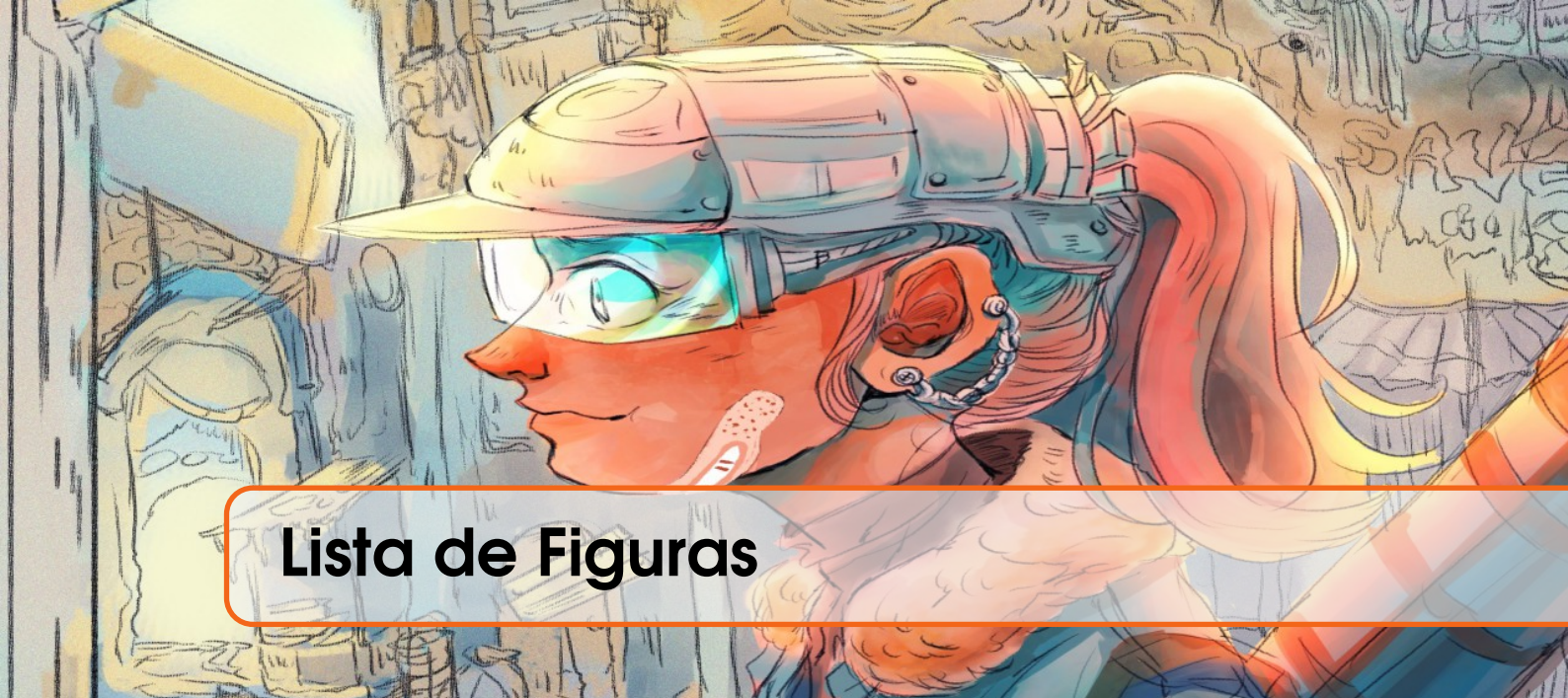
3	Procesos e Hilos	73
3.1	Clientes Multihilos	73
3.2	Servidor Multihilos	76
3.3	Virtualización de sistemas	81
3.3.1	VM y computación en la nube	83
3.4	Caso de Estudio: Virtualización de Redes	84
3.4.1	Skype: un ejemplo de una red superpuesta	85
4	Protocolos de la capa de Transporte	87
4.1	Socket	87
4.2	UDP	88
4.2.1	Uso de UDP	90
4.3	TCP	90
4.3.1	Mensajes de TCP	92
4.4	Multidifusión	93
4.4.1	La multidifusión IP	94
4.5	Caso de Estudio: ¿Como funciona un aplicación de Chat?	94
5	Comunicación entre Procesos Remotos	97
5.1	Protocolo Solicitud-Respuesta	97
5.1.1	Estructura del Mensaje	97
5.1.2	Fallas	100
5.1.3	Características del Protocolo solicitud-respuesta	100
5.1.4	Estilos del protocolo	100
5.2	Llamados a Procedimientos Remotos (RPC)	101
5.2.1	Programación con Interfases	101
5.2.2	Semántica de llamadas de RPC	101
5.2.3	Transparencia	102
5.2.4	Implementación de RPC	102
5.3	Invocación a Métodos Remotos (RMI)	104
5.3.1	Arquitectura RMI	105
5.3.2	Colector de Basura Distribuida	107
5.4	Caso de Estudio: API	107
5.4.1	¿Que es un API?	107
5.4.2	¿Que es REST?	107
5.4.3	GraphQL	108

6	Comunicación Indirecta	111
6.1	Grupos	111
6.1.1	Modelo de Programación	112
6.1.2	Membresía del Grupo	112
6.1.3	Características de la arquitectura de Grupos	113
6.1.4	Confiabilidad y ordenamiento en multidifusión	114
6.2	Publicación-Suscripción	115
6.2.1	Aplicaciones de los sistemas de publicación-suscripción	115
6.2.2	Modelo de Programación	116
6.2.3	Consideraciones de Implementación	117
6.3	Colas	120
6.3.1	Modelo de programación	121
6.3.2	Características	121
6.4	Casos de Estudio: Patrón y mensajería Pub/sub	122
6.4.1	Mensajería Publicación Suscripción	122
6.4.2	Rabbit MQ	124
7	Arquitectura orientada a Servicios	125
7.1	Informática orientada a Servicios	125
7.1.1	Elementos de Informática orientada a Servicios	125
7.1.2	Principios de diseño de orientación de servicio	128
7.2	Servicios Web	129
7.2.1	SOAP	130
7.2.2	Descripción de Servicios e Interfaces Web	132
7.2.3	Lenguaje de Descripción del Servicio	132
7.2.4	Descripción de los elementos de WSDL	133
7.2.5	Universal Description, Discovery and Integration service	139
7.2.6	Seguridad XML	140
7.2.7	Coordinación	141
7.3	REST	141
7.3.1	Restricciones de servicios REST	142
7.3.2	Metas de un servicios REST	144
7.3.3	Contratos de servicios no-Rest vs servicios REST:	145
7.4	Caso de Estudio: Monolitos vs Microservicios	148
8	Sistemas Punto a Punto	153
8.1	Introducción	153
8.2	Evolución de los sistemas P2P	155
8.2.1	1-era Generación: Napster	155
8.2.2	2-da Generación. Gnutella	156
8.2.3	3-era Generación. Algoritmos DHT	157

8.3	Caso de Estudio. Pastry	160
-----	-------------------------------	-----

IV **Coordinación y Replicación**

9	Tiempo y Consenso Distribuido	167
9.1	Tiempo	167
9.1.1	Relojes Físicos	168
9.1.2	Relojes Lógicos	172
9.2	Consenso Distribuido	176
9.2.1	Algoritmos basados en Token	176
9.2.2	Algoritmos basados en Marcas de Tiempo	178
9.2.3	Algoritmos basados en Elecciones	182
9.3	Caso de Estudio. Elección del Líder	185
10	Replicación	191
10.1	Replicación	191
10.1.1	Administrador de Réplicas	191
10.2	Tolerancia a Fallas	193
10.2.1	Linealizabilidad y Consistencia Secuencial	194
10.2.2	Modelo de Replicación Pasiva	195
10.2.3	Modelo de Replicación Activa	196
10.3	Caso de Estudio. Fragmentación	198
	Bibliografía	203
	Artículos	203
	Libros	206
	Glosario	211
	Índice	219



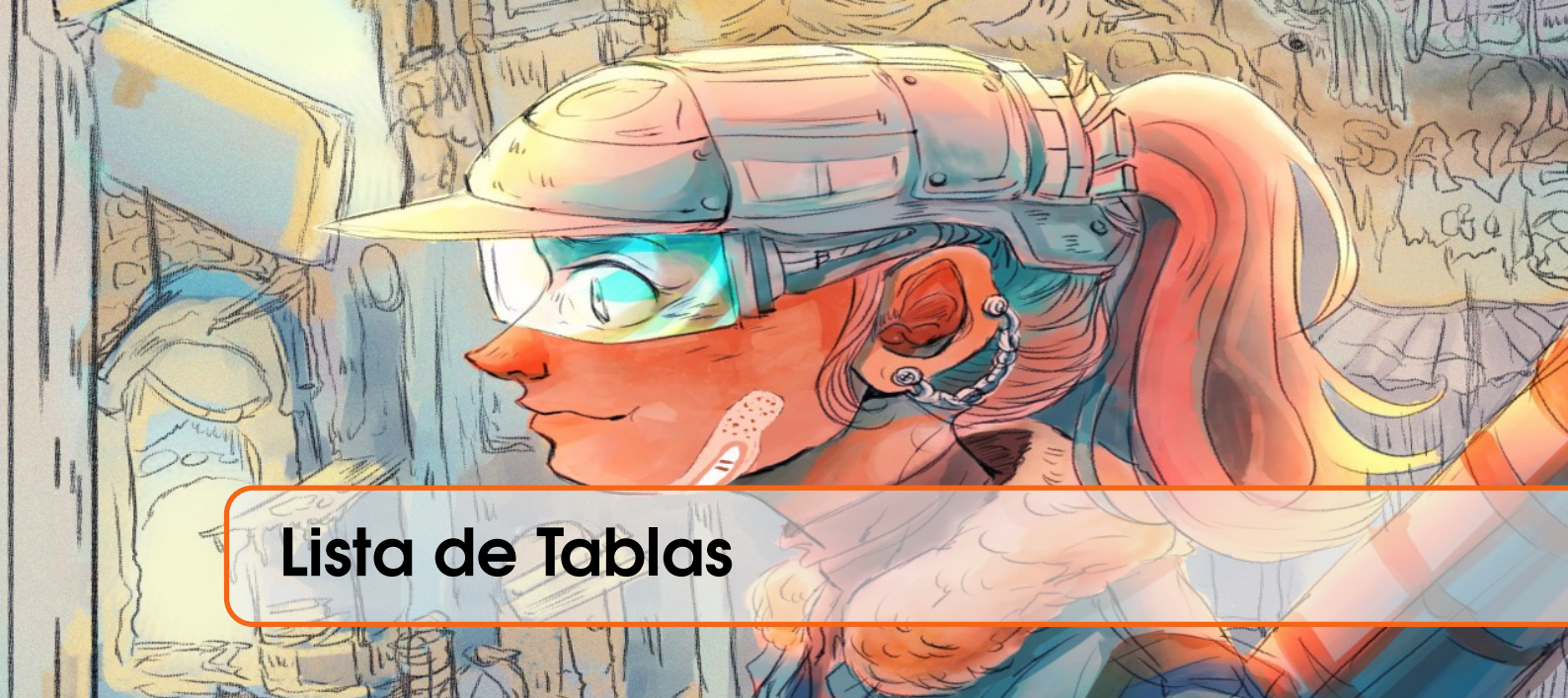
Lista de Figuras

1.1	Ejemplo de la división del espacio de nombres del DNS original a zonas. Tomado de (91)	24
1.2	Consistencia. Adaptado de (87)	25
1.3	Disponibilidad. Tomado de (87)	26
1.4	Tolerancia a la Partición. Adaptado de (90)	27
1.5	Relación entre conceptos del Teorema CAP. Adaptado de (57)	28
1.6	Significado del Teorema CAP	29
1.7	Procesos y canales.	31
1.8	Fallas de los Sistemas Distribuidos. Adaptado de (42)	33
1.9	Cliente y Servidor Web	35
1.10	Clientes, Servidor Web y Recursos	36
1.11	Partes de un URL	37
1.12	Transacción HTTP	37
1.13	Hola Mundo	39
1.14	Hola Mundo con hoja de estilo css	40
1.15	Solicitud página estática y página dinámica. Adaptado de (72)	43
1.16	Componentes de AJAX. Adaptado de (55)	45
1.17	Una interacción AJAX. Adaptado de (46)	46
2.1	Arquitectura Cliente Servidor en red LAN	48
2.2	Arquitectura Cliente Servidor	49
2.3	Arquitectura Punto a Punto	50
2.4	Organización de una base de datos para una red de monitoreo, mientras se almacena y procesa información (a) sólo en el lugar del operador, o (b) sólo en los sensores. Tomado de (88)	51
2.5	Mapeo de elementos de datos hacia nodos organizados en Chord. Tomado de (88)	52
2.6	Hilos en Arquitectura Cliente Servidor	52
2.7	Componentes de la arquitectura CORBA.	53
2.8	Arquitectura de software basado en componentes. Tomado de (54)	53

2.9	Arquitectura Publicación Suscripción.	55
2.10	Arquitectura Colas de Mensajes.	56
2.11	Arquitectura con múltiples servidores. Tomado de (54)	57
2.12	Arquitectura web proxy.	58
2.13	Arquitectura web applet.	59
2.14	Arquitectura de capas de dos niveles.	61
2.15	Arquitectura de capas de tres niveles.	62
2.16	Arquitectura de cliente ligero.	63
2.17	Patrón arquitectónico en servicios web.	64
2.18	Balanceador de Cargas con replicas. Tomado de (90)	65
2.19	Balanceador de Cargas. Tomado de (90)	65
2.20	Servidores Replicados. Tomado de (90)	66
2.21	Sistemas de Nombres de Dominio.	67
2.22	DNS: Búsqueda Interactiva desde el cliente.	68
2.23	DNS: Búsqueda Interactiva desde el servidor.	69
2.24	DNS: Búsqueda Recursiva desde el servidor.	70
2.25	DNS: Ejemplo de una búsqueda.	70
3.1	(a) Aplicación en red con su protocolo. (b) Aplicación con acceso a aplicaciones remotas. Adaptado de (91)	74
3.2	Transparencia de replicación de un servidor mediante una solución del lado del cliente. Adaptado de (91)	75
3.3	Servidor Multihilo.	77
3.4	Servidores multihilos por solicitud	78
3.5	Servidores multihilos por conexión	79
3.6	Servidores multihilos por objeto	80
3.7	Interfases en sistemas informaticos. Adaptado de (36)	82
3.8	Formas de Virtualización: (a) VM de proceso, (b) VMM nativo, (c) VMM alojado. Adaptado de (91)	83
3.9	Red Superpuesta.	84
3.10	Arquitectura Skype.	86
4.1	Socket.	88
4.2	Estructura del Datagrama	89
4.3	Protocolos TCP/IP	91
4.4	Intercambio de mensajes en TCP	92
4.5	Estructura de mensajes en TCP	93
4.6	Chat. Adaptado de (42)	95
5.1	Protocolo Solicitud Respuesta.	98
5.2	Estructura del mensaje.	99
5.3	Llamada a Procedimiento Remoto.	103
5.4	Arquitectura de RPC	104
5.5	Arquitectura RMI.	106
6.1	Model de Programación. Tomado de (54)	113

6.2	Grupos cerrados y abiertos. Tomado de (54)	114
6.3	Paradigma Publicación Suscripción.	116
6.4	Publicación-Suscripción centralizado.	118
6.5	Publicación-Suscripción distribuido. Tomado de (54)	118
6.6	Algoritmo de Filtrado. Tomado de (54)	120
6.7	Algoritmo de Encuentros. Tomado de (54)	121
6.8	Modelo de Programación.	122
6.9	Patrón Publicación-Suscripción.	123
7.1	Servicios y sus capacidades	126
7.2	Contrato de Servicios.	127
7.3	Composición de Servicios. Adaptado de (60)	127
7.4	Inventario de Servicios	128
7.5	Vista Física de Servicios. Tomado de (60)	128
7.6	Infraestructura de Servicios Web.	130
7.7	Servicio Web con SOAP y HTTP	132
7.8	IDL	133
7.9	Directorio de Servicios	140
7.10	Servidores sin estado	143
7.11	Cache. Tomado de (60)	144
7.12	Contrato SOAP. Tomado de (60)	146
7.13	Contrato REST. Tomado de (60)	148
7.14	Monolito. Tomado de (83)	149
7.15	Microservicios. Tomado de (83)	150
8.1	Arquitectura de Napster	156
8.2	Arquitectura de Gnutella	158
8.3	Tabla de enrutamiento Chord de 6 bits	159
8.4	Anillo Pastry. Tomado de (54)	161
8.5	Tabla de enrutamiento Pastry. Tomado de (54)	163
9.1	Sesgo y deriva del reloj	169
9.2	Algoritmo de Cristian	170
9.3	Tiempo en el Algoritmo de Cristian	170
9.4	Algoritmo de Bekerley	171
9.5	Protocolo NTP. Adaptado de h3c	172
9.6	Eventos concurrentes	173
9.7	Ejemplo del Algoritmo de Lamport.	174
9.8	Sincronización en el Algoritmo de Lamport. Tomado de (91)	174
9.9	Ejemplo del Algoritmo Vectorial.	176
9.10	Algoritmo de Servidor Central.	177
9.11	Algoritmo de Anillo de Procesos.	178
9.12	Algoritmo de Ricart-Agrawala.	181
9.13	Algoritmo de Anillo basado en Elección.	184
9.14	Algoritmo de Bully	187

9.15	Algoritmo de Generales Bizantinos. Caso 1.	188
9.16	Algoritmo de Generales Bizantinos. Caso 2.	188
9.17	Algoritmo de elección del líder Raft.	189
10.1	Arquitectura del administrador de replicas de datos	193
10.2	Protocolo de una solicitud. Tomado de (28)	194
10.3	Ejemplo de Linealizabilidad	195
10.4	Modelo de Replicación Pasiva. Adaptado de (54)	196
10.5	Modelo de Arquitectura de Replicación Activa. Adaptado de (54) ..	197
10.6	Fragmentación de base de datos	199
10.7	Fragmentación de datos	200
10.8	Caché fragmentado y replicado	201



Lista de Tablas

1.1 Sistema Centralizados vs Distribuido.	
Adaptado de (92)	20
1.2 Tipos de Transparencia.	
Adaptado de (91)	22
1.3 Tipos de Errores.	
Adaptado de (54)	32
1.4 Métodos Http	38
5.1 Semántica de Llamadas RPC. Adaptado de (54)	102
6.1 Acoplamiento en Espacio y Tiempo. Adaptado de (54)	112
7.1 Operación Solicitud-Respuesta en WSDL. Tomado de (40)	136



Lista de Algoritmos

1.1	Ejemplo programa en HTML	38
1.2	Hoja de estilo app.css	39
1.3	Hola Mundo con llamada a css	40
1.4	Hola Mundo con HTML y CSS	41
7.1	Estructura de mensaje SOAP	130
7.2	Solicitud en mensaje SOAP. Tomado de [40]	131
7.3	Respuesta en mensaje SOAP. Tomado de [40]	131
7.4	Protocolo de Transporte en SOAP. Tomado de [54]	131
7.5	Elemento <i>Definitions</i> . Tomado de [40]	133
7.6	Atributo xmlns. Tomado de [60]	133
7.7	Estructura de mensaje WSDL	134
7.8	Descripción de elemento <i><types></i> . Tomado de [40]	135
7.9	Descripción del elemento <i><message></i> . Tomado de [40]	136
7.10	Binding. Tomado de [40]	137
7.11	Definición de Servicio en WSDL.Tomado de [40]	137
7.12	Enlaces y Puertos en la definición de servicios. Tomado de [40]	138
7.13	Ejemplo de un mensaje WSDL. Tomado de [40]	138
7.14	Solicitud en contrato de servicio REST	147
7.15	Respuesta en contrato de servicio REST	147

Conceptos y Arquitecturas

1 Introducción a los Sistemas Distribuidos 19

1.1	Sistemas Distribuidos vs Centralizados	20
1.2	Principios de Diseño	22
1.3	Teorema CAP	24
1.4	Manejo de Fallas	30
1.5	Falacias de los sistemas distribuidos	32
1.6	Caso de Estudio: Tecnología Web	34

2 Arquitectura de los Sistemas Distribuidos 47

2.1	Modelos Físicos	47
2.2	Modelos Arquitectónicos	49
2.3	Soluciones basadas en middleware	65
2.4	Caso de Estudio: Sistema de Nombres de Dominio	66



1. Introducción a los Sistemas Distribuidos

Un sistema distribuido no solo es una red de computadoras; es una infraestructura conformada por un grupo de computadoras interconectadas por medio de enlaces de comunicación de diversos medios y topologías posibles, y que utilizan un conjunto de protocolos de comunicación para así mostrar al usuario la percepción de un sistema unificado.

Además de redes de computadoras, las capas adicionales, como el middleware, proporcionan la integración de aplicaciones construidas con diversos lenguajes, sistemas operativos y arquitecturas.

Los sistemas distribuidos pueden definir como: una colección de computadoras independientes que dan al usuario la impresión de constituir un único sistema coherente.

Esta definición [91] abarca dos elementos fundamentales: computadoras y usuarios o aplicaciones que usan este sistema. El primer elemento consta de componentes autónomos; el segundo, los usuarios (personas o programas) creen que realmente interactúan con un sistema único. Esto significa que de una manera u otra los componentes autónomos necesitan colaborar entre sí. En la forma de establecer esta colaboración radica el fondo del desarrollo de los sistemas distribuidos.

De igual manera en [92], define a los sistemas distribuidos como un sistema compuesto de varias computadoras las cuales se comunican a través de una red de computadoras, y que alberga procesos que usan un conjunto de protocolos distribuidos comunes, y que proporcionan una ejecución coherente de las actividades distribuidas.

Este concepto destaca los siguientes aspectos: una red de computadoras, los procesos que se ejecutan en esa red, los protocolos que permiten la comunicación entre los procesos y la integración de las aplicaciones construidas sobre la red de computadoras.

Por su parte, en [54], se define como aquel en el que los componentes de hardware o software están ubicados en una red de computadoras y, comunican y coordinan sus acciones solo pasando mensajes. Las computadoras que están conectadas por una red pueden estar separadas espacialmente por cualquier distancia, en continentes separados, en el mismo edificio o en la misma habitación.

Esta definición apunta a lo siguiente:

Concurrencia. en una red de computadoras, la ejecución concurrente de programas, es la norma.

Ausencia de reloj. No hay una noción global única del tiempo correcto. Esta es una

consecuencia directa del hecho de que la única comunicación es enviando mensajes a través de una red.

Fallas. Todos los sistemas informáticos pueden fallar, y es responsabilidad de diseñadores de sistemas planificar las consecuencias de posibles fallas.

1.1 Sistemas Distribuidos vs Centralizados

Los sistemas distribuidos contrasta con los sistemas de computación tradicionales, donde una sola computadora ejecuta el software que proporciona un servicio, o la computación cliente-servidor, donde varias maquinas accesan de manera remota a un servicio centralizado. En los sistemas distribuidos hay miles o millones de maquinas trabajando juntas para proporcionar un gran servicio.

Sistema Centralizado	Sistema Distribuido
Accesible	Ámbito geográfico
Homogéneo	Heterogéneos
Administrable	Modular
Consistente	Degradación elegante
Seguridad	Seguridad a costo bajo

Table 1.1: Sistema Centralizados vs Distribuido. Adaptado de [92]

En el cuadro 1.1 se muestra algunas de las diferencias entre los sistemas centralizado y distribuido [92].

Accesible vs Ámbito Geográfico. Los sistemas centralizados son homogéneos en cuanto a tecnologías y procedimientos; hay un acceso natural a los recurso porque estan constituidos por sistemas locales. Por su parte, los sistemas distribuidos tienen un alcance geográfico potencialmente más amplio, debido a que se puede operar y acceder de manera remota.

Homogéneos vs Heterogéneos. En cuanto a tecnologías y procedimientos, los sistemas centralizados son homogéneos ya que sus arquitecturas, protocolos, lenguajes son propios de un tipo de arquitectura; hay un acceso natural a los recurso porque están constituidos por sistemas locales. En cambio, los sistemas distribuidos son heterogéneos ya que poseen diferentes arquitecturas, protocolos y sistemas operativos.

Administrable vs Modular Los sistemas centralizados son administrables porque estan compuestos por estructuras centralizadas, y debido a ese tipo de estructura logran ser consistentes y seguros. Por su parte, los sistemas distribuidos, debido a su modularidad, resultan ser más expansibles y escalable en cuanto a número de sitios y extensión geográfica.

Consistente vs Degradación elegante. Es más fácil mantener un sistema coherente en sistemas centralizados debido a que su administración es sobre tecnologías y procedimientos homogéneos, y recursos constituidos por sistemas locales.

Degradación Elegante

degradación elegante es el concepto que describe a sistemas informáticos y de red que pueden operar de manera progresivamente degradada, en la medida que fallan sus componentes, pero sin la ocurrencia de un colapso en el sistema como consecuencia de esas fallas. Presentado en *Emerging resilience techniques for embedded devices*, Demara et al, 2017.

Los sistemas distribuidos expresan un comportamiento llamado degradación elegante. Esta característica potencia que los sistemas distribuidos puedan lograr confiabilidad y disponibilidad. Y, conjuntamente con la modularidad de los componentes, la separación geográfica, la redundancia y las técnicas de reconfiguración, logran que los sistemas distribuidos tenga una alta tolerancia a fallos.

Seguridad vs Seguridad a bajo costo. La seguridad en los sistemas centralizados se logra mediante el control de acceso físico y aislamiento. En cuanto a los sistemas distribuidos, se puede alcanzar un alto nivel de seguridad con un costo más bajo, siempre que sea logrado más a costa de reducir el efecto de las intrusiones, que el de las amenazas; esto es debido a lo difícil y costoso que resulta reducir amenazas en sistemas abiertos y públicos.

Ejemplos de Sistemas Distribuidos Hay una amplia gama de ejemplos en sistemas distribuidos. Muestra de ellos [54]:

Comercio electrónico y Finanzas El crecimiento del comercio electrónico, ejemplificado por empresas como Amazon y eBay, y tecnologías de pago como PayPal; banca y el comercio en línea y sistemas de difusión de información para los mercados financieros.

Industria del Entretenimiento El surgimiento de los juegos en línea, como una forma novedosa y altamente interactiva de entretenimiento; la disponibilidad de música y películas en el hogar a través de centros de medios en red e Internet por medio de contenido descargable o en tiempo real; temas generado por el usuario, por ejemplo a través de servicios como YouTube; nuevas formas de arte y entretenimiento habilitado por tecnologías emergentes.

Salud El crecimiento de la informática en la salud como disciplina con su énfasis en registros electrónicos de pacientes en línea y cuestiones relacionadas con la privacidad; el papel creciente de la telemedicina en el apoyo al diagnóstico remoto o servicios avanzados como cirugía remota.

Educación La aparición del *e-learning* mediante, por ejemplo, herramientas basadas en la web, como entornos virtuales de aprendizaje; apoyo asociado con la educación a distancia; apoyo para el aprendizaje colaborativo o comunitario.

Transporte El uso de tecnologías como GPS en sistemas de búsqueda de rutas y gestión de tráfico. Servicios como MapQuest, Google Maps y Google Earth.

Ciencias El surgimiento de Grid como tecnología fundamental para las eCiencias, incluyendo el uso de redes de computadoras complejas para apoyar el almacenamiento, análisis y procesamiento de datos científicos.

Gestión Ambiental El uso de tecnología de sensores (en red) para monitorear y administrar el medio ambiente natural, por ejemplo, para proporcionar una alerta temprana de desastres naturales como terremotos, inundaciones o tsunamis y para coordinar respuesta de emergencia; la recopilación y el análisis de los parámetros ambien-

Concepto	Descripción
Acceso	Permitir el acceso con las mismas operaciones de los recursos locales y remotos.
Ubicación	Que los recursos sean alcanzados sin conocer su ubicación física o de red.
Red	Combina ambas transparencias: en el acceso y en la ubicación.
Concurrencia	Permite que varios procesos operen simultáneamente usando recursos compartidos sin interferencia entre ellos.
Replicación	Permite que múltiples instancia de recursos sean usados para aumentar la confiabilidad y rendimiento sin que el usuario de aplicaciones conozca de ellas.
Fallas	Ocultamiento de fallas, permitiendo que usuarios y aplicaciones culminen sus tareas sin percatarse de ellas.
Movilidad	Permite el movimiento de recursos y clientes dentro del sistema sin afectar la operación de usuarios o programas.
Rendimiento	Permite que el sistema sea reconfigurado para mejorar el rendimiento cuando su carga varía.
Escalamiento	Permite que el sistema y la aplicación se incremente sin cambios en la estructura del sistema o en los algoritmos de aplicación.

Table 1.2: Tipos de Transparencia. Adaptado de [91]

tales para comprender mejor los complejos naturales fenómenos como el cambio climático.

1.2 Principios de Diseño

La siguiente sección describe los objetivos o lo que debe tomarse en cuenta cuando se diseña un sistema sistema distribuido: [54, 56, 57, 90, 91, 92]

Concurrencia. Tanto los servicios como las aplicaciones proporcionan recursos que los clientes pueden compartir en un sistema distribuido. Por tanto, existe la posibilidad de que varios clientes intenten acceder a un recurso compartido al mismo tiempo. La concurrencia se hace más compleja cuando existen actividades paralelas que interactúan o comparten los mismos recursos.

Compartir Recursos. Los recurso, como los periféricos, datos en bases de datos, bibliotecas, así como los datos (variables/archivos), no pueden replicar por completo en todos los sitios porque no resulta práctico ni rentable. Estos recursos se distribuyen normalmente por todo el sistema, para que su acceso no se convierta en un potencial cuello de botella. Por ejemplo, bases de datos distribuidas como MySQL Cluster o Ethereum, particionar los conjuntos de datos en varios servidores, además de replicarlos en algunos sitios para lograr un acceso rápido y proporcionar un servicio confiable.

Transparencia. El objetivo de la transparencia es hacer que ciertos aspectos de la distribución sean invisibles para el programador de aplicaciones. Por ejemplo, no necesitan preocuparse por la ubicación o los detalles de cómo otros componentes acceden a sus operaciones, o si serán replicados o migrados. Incluso se pueden presentar fallas en la redes y procesos en forma de excepciones, pero estas deben poder manejarse. En el cuadro 1.2 se detallan los niveles de transparencia que se deben tomar en cuenta acompañado de su descripción.

Sistemas Abiertos. Los sistemas abiertos son aquellos que puede ampliarse y reimplementarse.

mentarse de varias formas. La apertura está determinado principalmente por el grado en que los nuevos servicios de intercambio de recursos puede agregarse y estar disponible para su uso por una variedad de programas de cliente. Los sistemas abiertos se caracterizan por el hecho de que sus interfaces son públicas; en otras palabras, las aplicaciones proveen un mecanismo de comunicación uniforme e interfaces publicadas para el acceso a recursos compartidos.

Simplicidad. Es importante que un diseño sea lo más simple posible y al mismo tiempo pueda satisfacer las necesidades del servicio ya que los sistemas crecen y se vuelven más complejos con el tiempo. Esto facilita su mantenimiento.

Acoplamiento. El grado de acoplamiento entre un conjunto de módulos, ya sea hardware o software, se mide en términos de la interdependencia y vinculación y/o homogeneidad entre los módulos.

Acoplamiento

El concepto de acoplamiento fue presentado por Edward Yourdon en *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon, 1979.

Cuando el grado de acoplamiento es alto, se dice que los módulos están acoplados de manera ajustada o fuerte; por el contrario cuando el grado de acoplamiento es bajo, los módulos están acoplados de manera floja o débil. Un sistema debilmente acoplado facilita la sustitución o adición de componentes mientras está en funcionamiento. Como resultado, un subsistema puede ser reemplazado por uno que proporcione la misma interfaz abstracta incluso si su implementación es completamente diferente. Por el contrario, en los sistemas fuertemente acoplado es necesario que se compartan recursos comunes, como la memoria central, almacenamiento secundario (disco) y entrada/salida a través de un bus común.

Acoplamiento débil vs fuerte

Puede explorar las diferencias entre el acoplamiento fuerte versus el acoplamiento débil en este enlace [geeksforgeeks](https://www.geeksforgeeks.org/strong-weak-coupling/).

Escalable. Un sistema se describe como escalable si permanece efectivo cuando hay un aumento significativo en el número de recursos y el número de usuarios. El diseño de sistemas distribuidos escalable presenta los siguientes retos:

- Controlar el costo de los recursos físicos. Es posible ampliar un sistemas, a un costo razonable, a medida que crece la demanda de un recurso.
- Controlar la pérdida de rendimiento. La gestión de un conjunto de datos cuyo tamaño debe ser proporcional al número de usuarios o recursos en el sistema, para evitar el congestionamiento en el acceso a los recursos y afectación del rendimiento.
- Evitar cuellos de botella. En general, los algoritmos deben estar descentralizados para evitar cuellos de botella. Un ejemplo de un problema grave de cuello de botella es el predecesor del sistema de dominio de nombres DNS , en el que la tabla de nombres se mantenía en un archivo maestro único que se podía descargar a cualquier computador. Esto estuvo bien cuando solo había

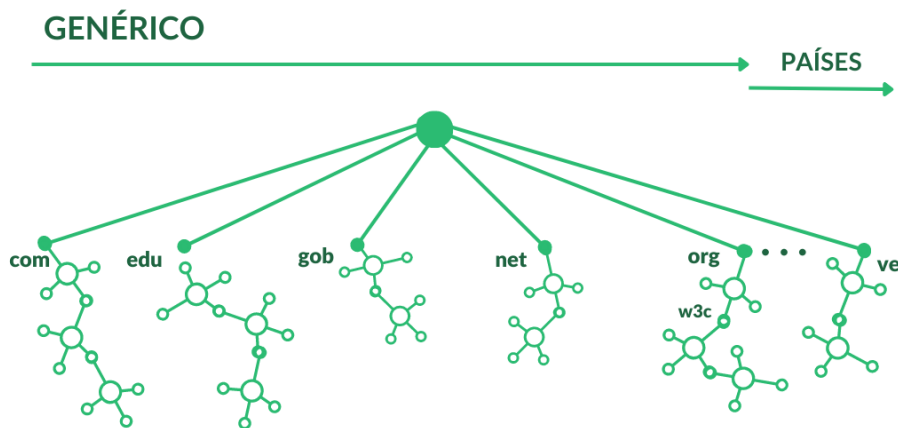


Figure 1.1: Ejemplo de la división del espacio de nombres del DNS original a zonas. Tomado de [91]

unos pocos cientos de computadoras en internet. Posteriormente, se eliminó este cuello de botella al dividir la tabla de nombres entre servidores ubicados en Internet y administrados localmente. Ver figura 1.1

Heterogeneidad. Los sistemas distribuidos son heterogéneos, ya que se pueden construir a partir de una variedad de redes, sistemas operativos, hardware informático y lenguajes de programación. Contemplar, por ejemplo, código móvil o máquina virtual. El middleware se usa para ocultar estas diferencias y permitir la comunicación y administración de datos entre aplicaciones distribuidas.

Seguridad. No es suficiente proporcionar acceso a servicios distribuidos. También es importante proporcionar garantías con respecto a cualidades asociadas con dicho acceso al servicio. Ejemplos de estas características incluyen parámetros relacionados con el rendimiento, seguridad y confiabilidad: En un sistema distribuido, el cliente envía solicitudes de acceso a través de la red a un conjunto de servidores, o almacena sus datos en servicios en la nube. Ambos ejemplos requieren que se construyan aplicaciones seguras donde se proteja la información que se envía por la red o la que se almacena en la nube. Este último requerimiento es estudiado por la criptografía de datos en bases de datos en la nube.

1.3 Teorema CAP.

El teorema CAP, inicialmente llamado como la conjetura CAP, recibió el estatus de teorema cuando se proporcionó una prueba matemática del concepto, ver [5]. CAP establece que en un sistema distribuido, se puede cumplir como máximo con dos de estas propiedades: consistencia, disponibilidad y tolerancia de partición.

En cuanto al concepto de consistencia en aplicaciones distribuidas, por ejemplo, si

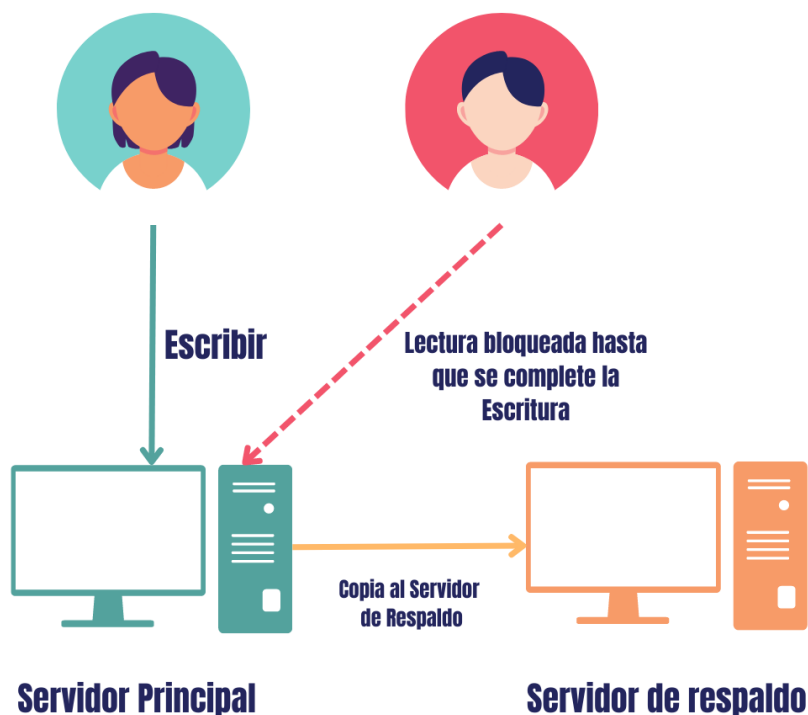


Figure 1.2: Consistencia. Adaptado de [87]

una empresa usa un servidor de base de datos de respaldo, cuando un usuario actualiza la cuenta de un cliente, esos mismos cambios se realizarían en el servidor de respaldo.

Esto requeriría que la base de datos escribiera los datos dos veces: una vez en el disco usado por el servidor primario y luego una vez más en el disco usado por el servidor de respaldo en una operación conocida como confirmación de dos fases, ver Figura 1.2. Mientras se ejecuta la confirmación de dos fases, se bloquean otras consultas a los datos. Los datos actualizados no estarán disponibles hasta que finalice la confirmación de dos fases. Esto favorece la coherencia sobre la disponibilidad de los datos.

La disponibilidad de datos se ilustra en el ejemplo del carrito de compras de comercio electrónico [87], donde es posible tener una copia de seguridad de los datos del carrito que no esté sincronizada con la copia principal. Los datos seguirían estando disponibles si el servidor primario falla, pero los datos del servidor de respaldo serían inconsistentes con los datos del servidor primario si el servidor primario falla antes de actualizar el servidor de respaldo. Ver la Figura 1.3.

El ejemplo más simple de tolerancia de partición es cuando el sistema continúa funcionando incluso si las máquinas involucradas en la prestación del servicio pierden la capacidad de comunicarse entre sí debido a que un enlace de red se cae (ver Figura 1.4).

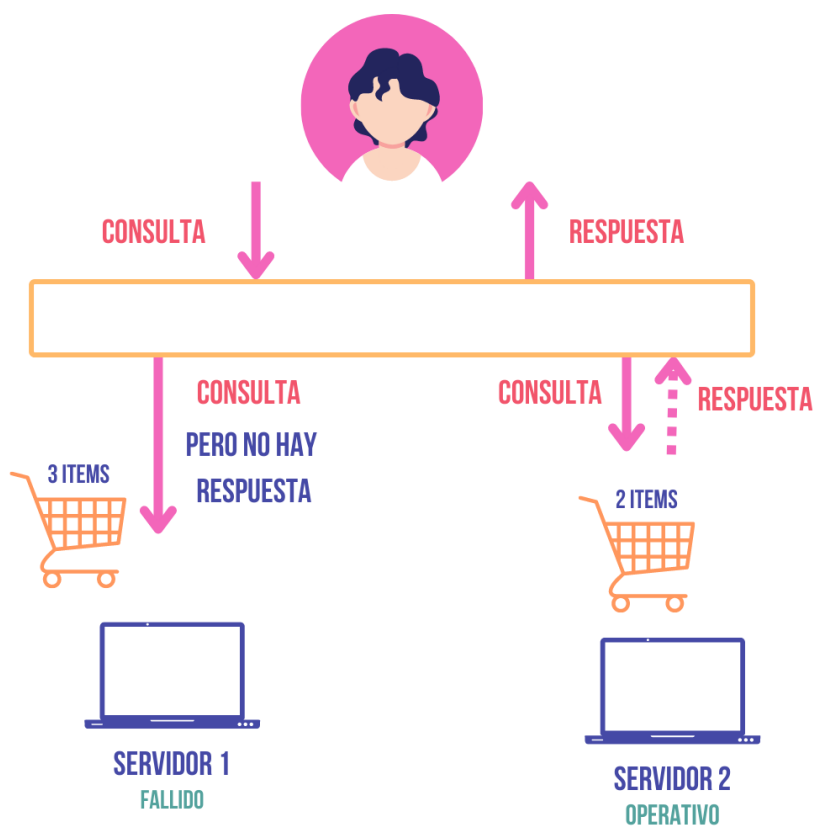


Figure 1.3: Disponibilidad. Tomado de [87]

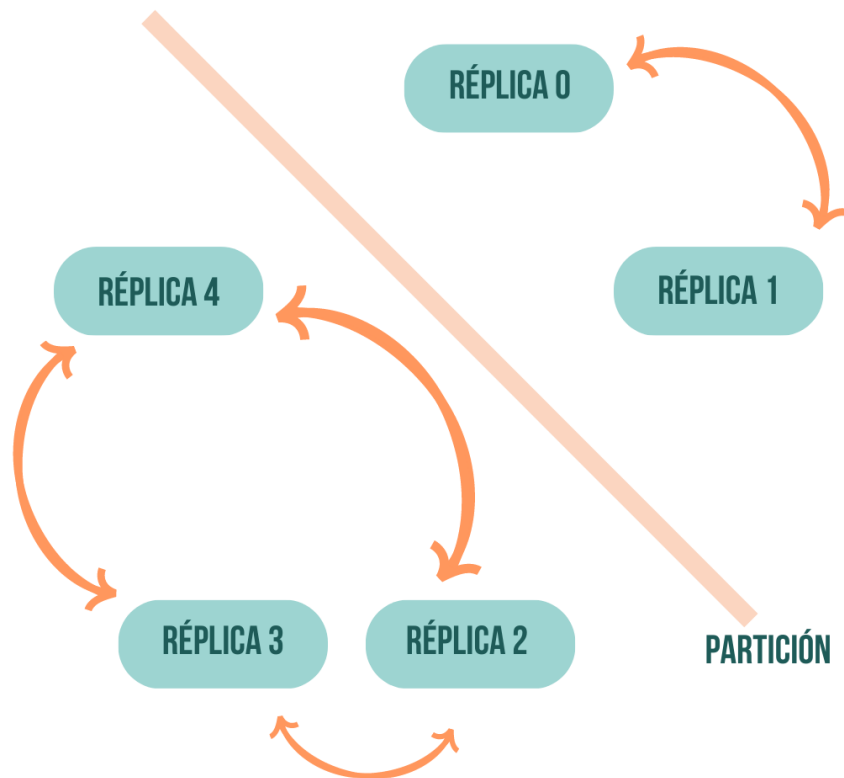


Figure 1.4: Tolerancia a la Partición. Adaptado de [90]

La Figura 1.5 esquematiza estos conceptos y la relación entre ellos. Allí se resalta los puntos de intersección entre las propiedades definidas en el teorema, por ejemplo, CA es la intersección para los sistemas que cumplan la propiedad de consistencia y disponibilidad, CP son los sistemas con las propiedades de consistencia y tolerancia a la partición, y AP para las propiedades de disponibilidad y tolerancia a la partición.

Noten que la intersección entre los tres conceptos no está definida en la Figura 1.5. El principio CAP establece que no es posible construir un sistema distribuido que garantice los tres conceptos: consistencia, disponibilidad y tolerancia a la partición. Se pueden lograr uno o dos de ellos, pero no los tres simultáneamente. Al utilizar un sistema distribuidos se debe tener en cuenta qué principios puede garantizar, de acuerdo a su diseño.

En el contexto de una aplicación de red social global, o un sistema de comercio electrónico mundial, la solución deseada es mantener la disponibilidad incluso si se sacrifica cierta consistencia entre los usuarios. Por otra parte, en el sistema financiero mundial, requiere mantener la consistencia de los datos sobre la disponibilidad de los mismos, por tanto las actualizaciones podrían requerir más tiempo para su ejecución.

En la Figura, 1.6 se muestra las propiedades del Teorema CAP y los tipos de aplicaciones que cumplen con estas propiedades.

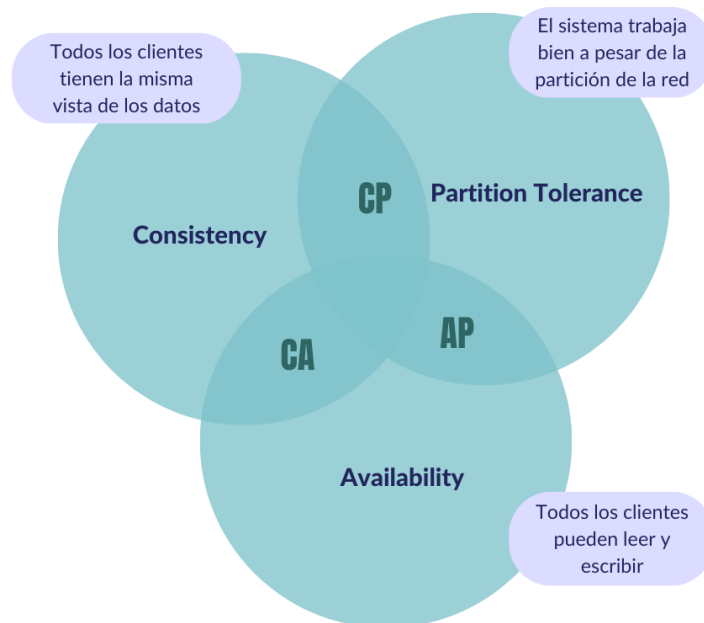


Figure 1.5: Relación entre conceptos del Teorema CAP. Adaptado de [57]

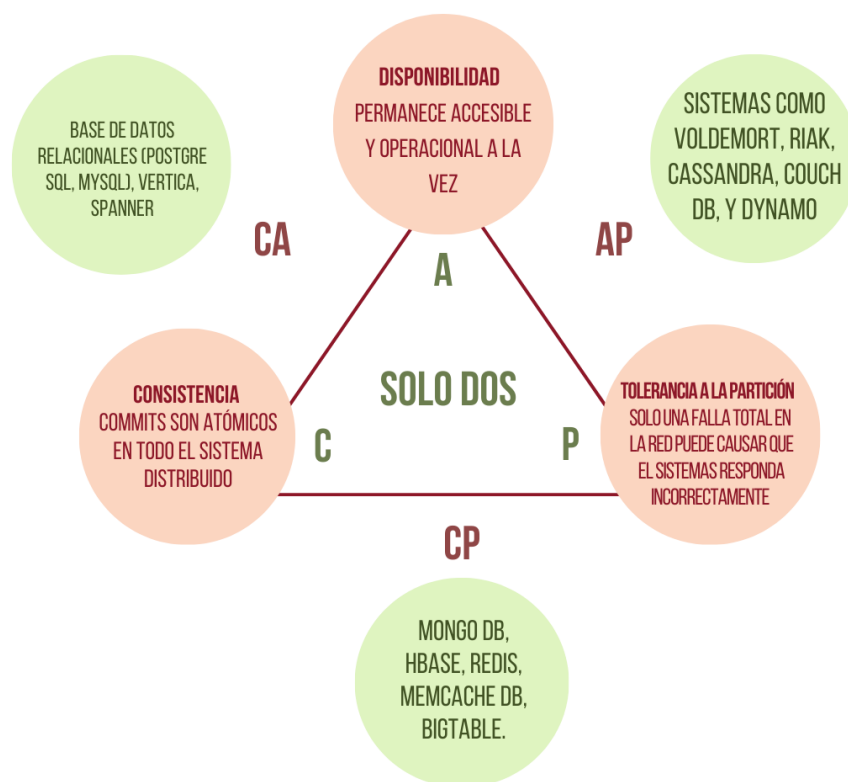


Figure 1.6: Significado del Teorema CAP.

1.4 Manejo de Fallas

Los sistemas informáticos a veces fallan. Cuando estas ocurren, ya sea en hardware o software, los programas pueden producir resultados incorrectos o detenerse antes de que hayan completado el cálculo. Las fallas en un sistema distribuido son parciales, es decir, algunos componentes fallan mientras otros continúan funcionando. Por lo tanto, el manejo de fallas es particularmente difícil.

Existen técnicas para lidiar con fallas: detección de la ocurrencia de fallas, enmascaramiento de las fallas, tolerancia a las fallas, recuperación luego de la ocurrencia de fallas y la redundancia de rutas, caminos o servidores para garantizar funcionamiento de las aplicaciones, a pesar de la fallas.

En [15] destacan que las fallas en los sistemas distribuidos puede atribuirse a la complejidad de la ingeniería de los mismos y se deben a los siguientes motivos:

Retos en Sistemas Distribuidos

Puede consultar este documento en [Retos en SD](#)

- Los ingenieros no pueden combinar las condiciones de error. En su lugar, deben considerar muchas combinaciones de fallas. La mayoría de los errores pueden ocurrir en cualquier momento, independientemente de cualquiera otra condición de error, por lo que podrían combinarse entre ellos.
- El resultado de cualquier operación de red puede ser DESCONOCIDO. En cuyo caso es posible que la solicitud haya fallado, se haya procesado correctamente o se haya recibido pero no procesado.
- Los problemas distribuidos se producen en todos los niveles. No solo en los equipos físicos de nivel bajo, sin tembién, en los niveles lógicos del sistema distribuido.
- Recursividad. Los problemas distribuidos empeoran en los niveles superiores del sistema, debido a la recursividad.
- Aparición del error. Los errores distribuidos suelen aparecer mucho después de su implementación en un sistema.
- Propagación del error. Los errores distribuidos se pueden propagar en todo el sistema.
- Origen de la falla. Muchos de estos problemas provienen de las leyes físicas de las redes, que no se pueden cambiar.

Las fallas se caracaterizan como [54]:

Fallos por omisión Los fallos clasificados como fallos por omisión se refieren a casos en los que un proceso o el canal de comunicación no realiza las acciones que se supone que debe realizar

Fallos por omisión del proceso El principal fallo por omisión de un proceso es que este se bloquee. Un proceso se ha bloqueado cuando se ha detenido y no ejecutará ningún paso de su programa. En los sistemas síncronos el método de detección de estas fallas se basa en el uso de tiempos de espera - es decir, un método en el que un proceso permite un período de tiempo fijo para que ocurra algo.

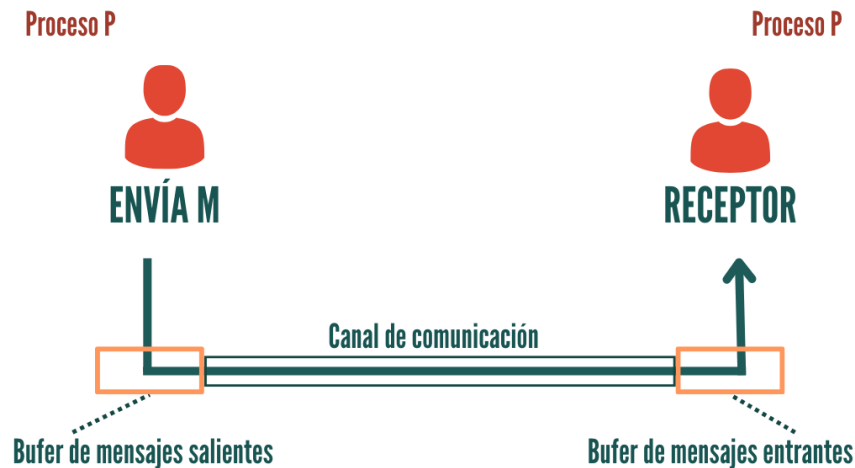


Figure 1.7: Procesos y canales.

Fallas de omisión en sistemas síncronos

Por ejemplo, si los procesos p y q están programados para que q pueda responder a un mensaje de p , y si el proceso p no ha recibido respuesta del proceso q en un tiempo máximo medido en el reloj local de p , entonces el proceso p puede concluir que el proceso q ha fallado.

En un sistema asincrónico, un tiempo de espera puede indicar solo que un proceso no responde: puede que se haya bloqueado o sea lento, o los mensajes pueden que no han llegado.

Fallos de omisión de comunicación Considere las primitivas de comunicación que envían y reciben mensajes. Un proceso p realiza un envío insertando el mensaje m en su buffer de mensaje saliente. El canal de comunicación transporta m al búfer de mensajes entrantes de q . El proceso q realiza una recepción tomando m de su búfer de mensajes entrantes y lo entrega (ver Figura 1.7). Los búferes de mensajes entrantes y salientes son proporcionado por el sistema operativo.

El canal de comunicación produce una falla por omisión si no transporta un mensaje del búfer de mensajes salientes de p al búfer de mensajes entrantes de q . Esto se debe a la falta de espacio en el búfer en el receptor o en una puerta de enlace intermedia, o por un error de transmisión de red, detectado por un control de suma (*check sum*)

llevada con los datos del mensaje.

Fallos arbitrarios El término fallo arbitrario o bizantino se utiliza para describir lo peor falla de semántica posible, en la que puede ocurrir cualquier tipo de error. Por ejemplo, un proceso puede establecer valores incorrectos en sus elementos de datos, o puede devolver un valor incorrecto en respuesta a una invocación. Un fallo arbitrario de un proceso es aquel en el que omite arbitrariamente los pasos de procesamiento o toma pasos de procesamiento no deseados.

Clase de Falla	Afecta	Descripción
Parada	Proceso	Proceso se detiene y permanece detenido. Otros procesos pueden detectar este estado.
Bloqueo	Proceso	Proceso se detiene y permanece detenido. Otros procesos no pueden detectar este estado.
Omisión	Canal	Mensaje insertado en un búfer de mensajes salientes no llega al búfer de mensajes entrante del otro extremo
Omisión Envío	Proceso	Proceso completa una operación de envío pero el mensaje no se coloca en su búfer de mensajes salientes.
Omisión Recepción	Proceso	Mensaje se coloca en búfer de mensajes entrantes de un proceso, pero ese proceso no lo recibe.
Arbitrario (Bizantino)	Proceso	Proceso/canal muestra un comportamiento arbitrario: puede enviar/transmitir mensajes arbitrarios en momentos arbitrarios o hacer omisiones; puede detenerse o tomar un paso incorrecto.

Table 1.3: Tipos de Errores. Adaptado de [54]

1.5 Falacias de los sistemas distribuidos

Las falacias, también nombrado como trampas, son un conjunto de errores o falsas creencias que se cometen al diseñar y desarrollar un sistema distribuido.

Historia de las falacias

Peter Deutsch, mientras trabajaba en la empresa Sun Microsystems, en la década de los 90 propuso la lista de 7 falacias de la computación distribuida; posteriormente, en 1994, James Gosling, fundador de Java, añadió una más para finalmente ser conocida como las ocho falacias de los sistemas distribuidos.

Las falacias son [19, 42]:

La red es fiable. Los sistemas no son inmunes a fallas: los servidores pueden estar fuera de servicio, la energía eléctrica puede fallar. Las aplicaciones deben estar construidas para sortear estas fallas.

La latencia es cero. La latencia en redes pequeñas pueden considerarse casi cero; pero en las redes Wan, con servidores remotos, y usuarios alrededor del mundo, la latencia puede ser significativa. En estos casos, las aplicaciones deben tener cuidado con las respuestas tardías. Contemplar mecanismos para desechar las solicitudes, hacer reintentos de peticiones idempotentes, como mecanismo de prevención de fallas.

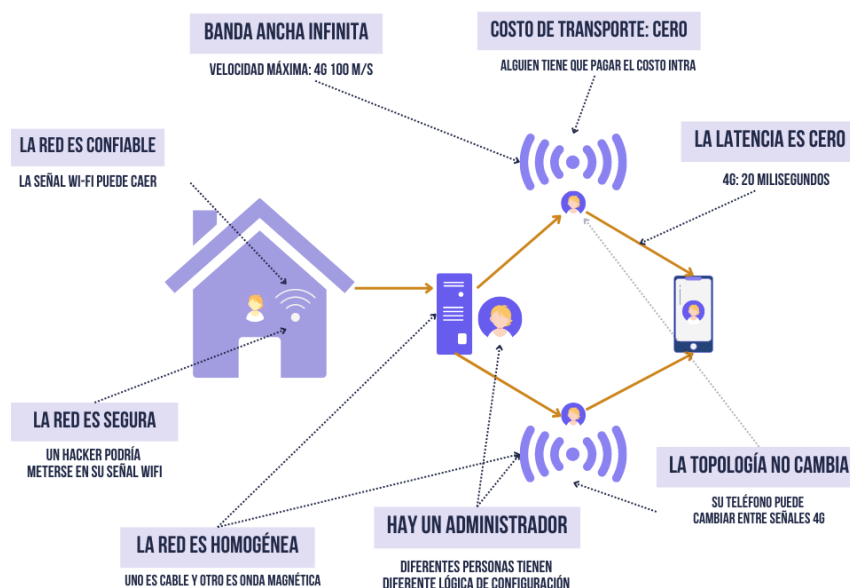


Figure 1.8: Fallas de los Sistemas Distribuidos. Adaptado de [42]

El ancho de banda es infinito. Así como el ancho de banda aumenta debido a las mejoras de la tecnología, el volumen de datos se incrementa. Por ello, podemos tener problemas con el ancho de banda lo cual influiría en la degradación del rendimiento de la aplicación.

La red es segura. Es un error no prestar atención a la seguridad de la aplicación. Las aplicaciones están expuestas a software maliciosos que pueden alterar las funcionalidades del software.

La topología no cambia. La red está en cambio constante: nuevas direcciones ip, servidores, dispositivos, servicios, clientes, entre otros. Los cambios en la topología influye sobre el ancho de banda y el rendimiento de las aplicaciones.

Hay un solo administrador. Un solo administrador es posible en redes pequeñas, pero para grandes redes, distribuidas geográficamente y con distintos propietarios, esto no se cumple.

El costo de transporte es cero. En la comunicación entre procesos distribuidos intervienen equipos, sistemas de balance de carga y ancho de banda; en cuanto a la comunicación entre las aplicaciones están involucrado el protocolo que se usa y como se serializa y deserializa.

La red es homogénea. Una red homogénea es pequeña con equipos bajo la misma tecnología, configuración y características. Pero en grandes redes esto no es así: allí se soportan una gran variedad de protocolos y dispositivos, aplicaciones con distintas necesidades y sistemas heterogéneos.

En la Figura 1.8 se muestra un esquema con la síntesis de las ocho falacias de los sistemas distribuidos ya referidas.

En resumen, diseñar una aplicación es una tarea que requiere considerar aspectos que están fuera del alcance del diseñador de la aplicación; por ello el proceso de diseño requiere

que se haga un ejercicio de escenarios probables de funcionamiento para determinar si la aplicación puede seguir operando a pesar de las posibles escollos que encuentre.

1.6 Caso de Estudio: Tecnología Web

La Web es un sistema abierto que se caracteriza por [54]: a) Su funcionamiento está basado en estándares de comunicación y contenido o en documentos que se publican e implementan libremente. Ejemplo, hay varios tipos de navegador, implementado en distintas plataformas; así como implementaciones de servidores web. b) La Web está abierta con respecto a los tipos de recurso que se pueden publicar y compartir. Los navegadores están diseñados para adaptarse a nuevas funcionalidades de presentación de contenido en forma de aplicaciones auxiliares y complementos o *plug-ins*.

La Web se basa en componentes tecnológicos que son estándares de la W3C, [40] y que se describen en esta parte. Se presentan una introducción a las siguientes tecnologías que se usan en la web: HTTP, HTML, CCS, JavaScript.

1.6.1 HTTP

HTTP (HyperText Transfer Protocol), HTTP es el protocolo usado cuando se visita cualquier sitio web desde un navegador (*browser*). Es un protocolo confiable que facilita la transferencia de información en la web. Esta característica de fiabilidad es debido a que HTTP utiliza protocolos de transmisión de datos confiables, garantiza que sus datos no se dañarán ni se codificarán durante el tránsito, incluso cuando provengan del otro lado del mundo [40], [70], [45].

HTTP

Conozca más de HTTP visitando este sitio [HTTP](#)

1.6.1.1 Clientes y Servidores Web

El contenido de la web está almacenado en un servidor web. Los servidores web usan el protocolo HTTP, por lo que a menudo se les llama servidores HTTP. Estos servidores almacenan los datos de Internet y proporcionan los datos cuando los solicitan los clientes HTTP. Los clientes envían solicitudes HTTP a los servidores y los servidores devuelven los datos solicitados en respuestas HTTP, como se muestra en la Figura 1.9.

1.6.1.2 Recursos

Los servidores web alojan recurso. Un recurso web es la fuente del contenido web. El tipo más simple de recurso web es un archivo estático en el sistema de archivos del servidor web: pueden ser archivos de texto, archivos HTML, archivos de Microsoft Word, archivos de Adobe Acrobat, archivos de imagen JPEG, archivos de película AVI o cualquier otro formato, ver Figura 1.10.

Los recursos también pueden ser programas de software que generan contenido bajo demanda. Estos recursos de contenido dinámico pueden generar contenido en función de su identidad, de la información que haya solicitado o de la hora del día, por ejemplo videos, transacciones de comercio electrónico, transacciones bancarias, entre otros

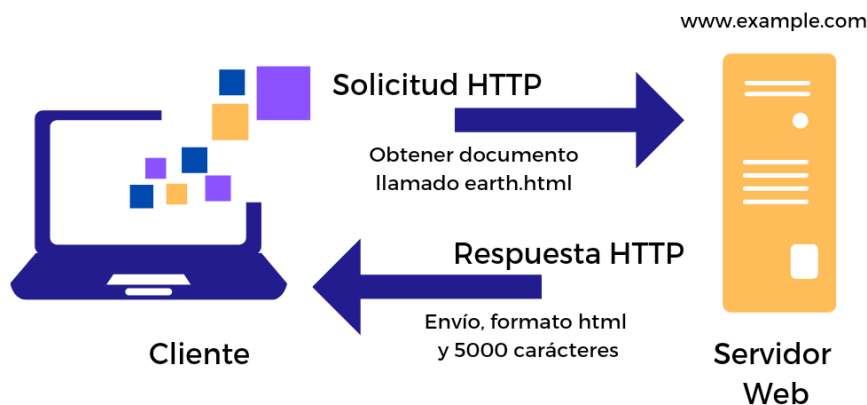


Figure 1.9: Cliente y Servidor Web

1.6.1.3 MIME

Debido a que Internet alberga miles de tipos de datos diferentes, HTTP etiqueta cada objeto que se transporta a través de la Web con una etiqueta de formato de datos denominada tipo MIME. MIME o *Multipurpose Internet Mail Extensions* (en español, Extensiones de correo de Internet multipropósito) se diseñó originalmente para resolver los mensajes entre los sistemas de correo electrónico. HTTP lo adoptó para describir y etiquetar su propio contenido multimedia.

Los servidores web adjuntan un tipo MIME a todos los datos de objetos HTTP. Cuando un navegador web recupera un objeto de un servidor, mira el tipo MIME asociado para ver si sabe cómo manejar el objeto. La mayoría de los navegadores pueden manejar cientos de tipos de objetos populares: mostrar archivos de imagen, analizar y formatear archivos HTML, reproducir archivos de audio a través de los parlantes de la computadora o iniciar software de complemento externo para manejar formatos especiales, entre otros

1.6.1.4 URL

Los navegadores examinan las URL para acceder a las correspondientes recursos. A veces, el usuario escribe una URL en el navegador, o el navegador busca la URL correspondiente cuando el usuario hace clic en un enlace o selecciona uno de sus marcadores o *bookmarks*.

URL

Conozca más de URL visitando este sitio [URL](#)

Ejemplo de las partes de un URL se esquematiza en la Figura 1.11:

- Esquema: es el protocolo usado para realizar la solicitud. Puede ser sin seguridad (http), encriptado (https) y para transferencia de archivos (ftp).

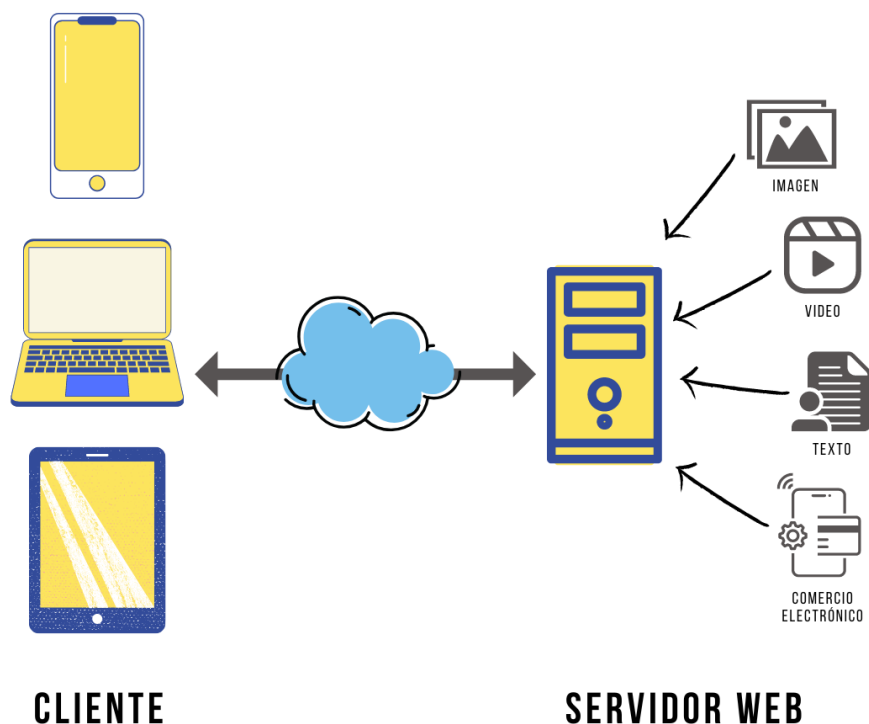


Figure 1.10: Clientes, Servidor Web y Recursos

- Dominio: donde se envía la solicitud. En las solicitudes de tipo HTTP, el nombre del servidor de destino (host) se fija con este valor.
- Puerto: por defecto 80 para HTTP y 443 para HTTPS.
- Camino: es la dirección que se debe seguir en el servidor.
- Consulta: parámetros de consulta usados para especificar la página que se solicita.
- Fragmento: no es enviado en la solicitud al servidor. Se usa para buscar una etiqueta en el documento HTML o por un programa JavaScript en la página.

1.6.1.5 Transacciones HTTP

Una transacción HTTP consta de un comando de solicitud (enviado del cliente al servidor) y un resultado de respuesta (enviado del servidor al cliente). Esta comunicación ocurre con bloques de datos formateados llamados mensajes HTTP, como se ilustra en la Figura 1.12

1.6.1.6 Métodos HTTP

HTTP proporciona comandos de solicitud diferentes, llamados métodos HTTP. Cada mensaje de solicitud HTTP tiene un método. El método le dice al servidor qué acción realizar (obtener una página web, ejecutar un programa de puerta de enlace, eliminar un archivo, etc.). En la tabla 1.4 hay una muestra de cuatro de ellos.

https://examplemed.com:443/cats?color=green #43xy

Esquema dominio puerto camino consulta fragmento

Figure 1.11: Partes de un URL

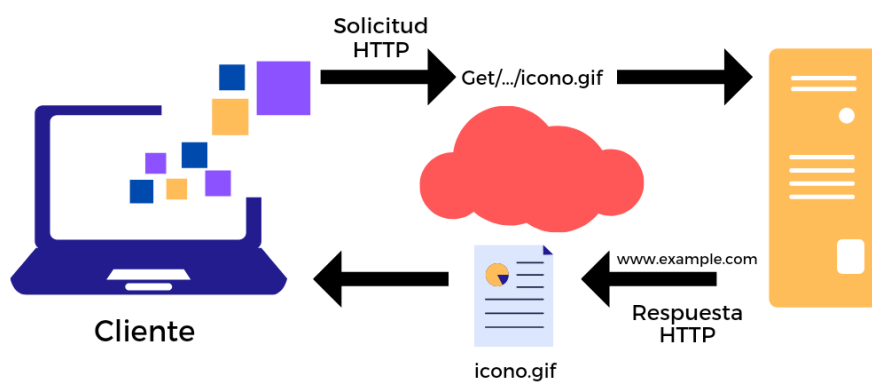


Figure 1.12: Transacción HTTP

Método	Descripción
GET	Enviar nombre del recurso desde el servidor al cliente.
PUT	Almacenar datos del cliente en el recurso de destino.
DELETE	Borra un recurso en específico.
POST	Envíar los datos del cliente a una aplicación en la puerta de enlace del servidor.

Table 1.4: Métodos Http

1.6.2 HTML

El lenguaje de marcado de hipertexto, HTML se utiliza para especificar el texto e imágenes que componen el contenido de una página web, y cómo se colocan y formatean, para su presentación al usuario. *HTML* también se utiliza para especificar enlaces y recursos que están asociados a ellos. [40]

HTML

Conozca más de HTML visitando este sitio [HTML](#)

A continuación, se muestra un fragmento de texto *HTML*, ver :

Algoritmo 1.1: Ejemplo programa en HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi primera Ejemplo </title>
  </head>
  <body>
    <h1>Este es el Encabezado </h1>
    <p> Hola Mundo! </p>
  </body>
</html>
```

En el listado 1.6.2 del programa se puede distinguir las siguientes partes:

- La declaración `<!DOCTYPE html>` indica que este documento es un documento HTML5.
- El elemento `<html>` es el elemento raíz de una página HTML.
- El elemento `<head>` contiene metainformación sobre la página *HTML*.
- El elemento `<title>` especifica un título para la página *HTML* (que se muestra en la barra de título del navegador o en la pestaña de la página).
- El elemento `<body>` define el cuerpo del documento y es un contenedor de todos los contenidos visibles, como encabezados, párrafos, imágenes, hipervínculos, tablas, listas, etc.
- El elemento `<h1>` define un encabezado grande.
- El elemento `<p>` define un párrafo.



Figure 1.13: Hola Mundo

Un documento *HTML* proporciona tres conceptos: etiqueta, atributo y valor.

- Etiqueta: las etiquetas comparten el mismo formato: empiezan con el signo menor que "<" y terminan con el signo mayor que ">". Por ejemplo, el elemento `html` tiene dos etiquetas: etiqueta de inicio `<html>` del documento *HTML* y la etiqueta de cierre `</html>` que indica el final del documento *HTML*.
- Atributo: Son las propiedades que se le pueden asignar a los elementos. Su formato es `<elemento atributo='valor'> ... </elemento>`.
- Valor: es la cualidad asignada al atributo. Ejemplo: `<html lang='es'> ... </html>` indica el idioma en que está escrito el documento (`lang`) y el valor asignado es un código de idioma ("es" para el español).

Para ejecutar un documento *HTML*, solo abra el documento con el navegador de su preferencia; obtendría la siguiente Figura 1.13

1.6.2.1 CSS

CSS es la abreviación para referirse a las hojas de estilo en cascada (Cascading Style Sheets), estas se utilizan para dar formato a las páginas web. Por ejemplo, *CSS* se puede utilizar para definir el color, el ancho, la altura, los márgenes, la opacidad, el relleno, entre otros atributos. etc. Usemos el programa 1.6.2 para mostrar como podemos incluir formato en la página *Hola Mundo*.

Escriba el programa 1.6.2.1 y guardelo como *app.css* en una carpeta llamada **css**. El programa tiene tres parte: la primera asigna color al fondo de la página, la segunda parte esta dedicada a colorear los elementos marcados con la etiqueta `<h1>`, y en la tercera parte se le asigna color a los elementos contenidos en la etiqueta `<p>`.

Algoritmo 1.2: Hoja de estilo *app.css*

```
html, body {
margin: 0px;
background-image: linear-gradient(to bottom right, #95d9f9, #ae4bc3);
}
h1 {
```

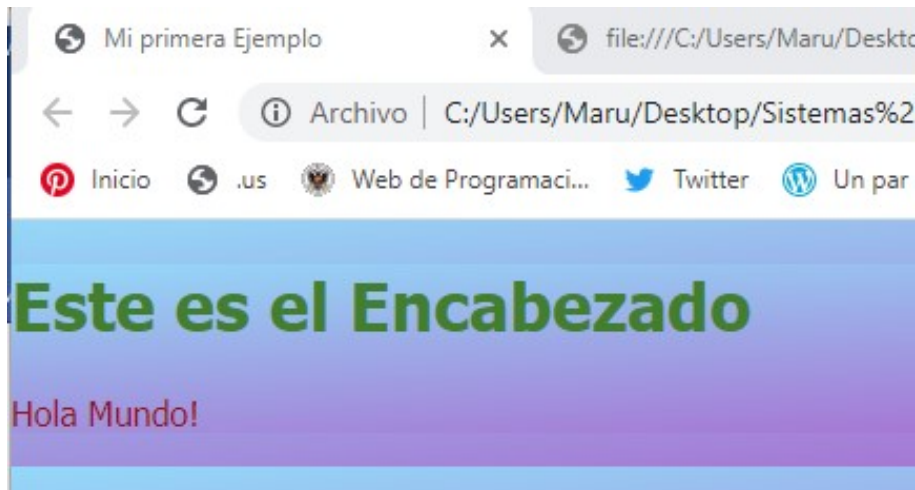


Figure 1.14: Hola Mundo con hoja de estilo css

```
color: #42802f;
font-family: tahoma, sans-serif;
}
p {
color: #991224;
font-family: tahoma, sans-serif;
}
```

Incluya en el programa Hola Mundo la siguiente línea `<link rel="stylesheet" href="css/app.css">`, debajo de la etiqueta *title*

El programa Hola Mundo modificado se muestra en 1.6.2.1.

Algoritmo 1.3: Hola Mundo con llamada a css

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi primer Ejemplo</title>
    <link rel="stylesheet" href="css/app.css">
  </head>
  <body>
    <h1>Este es el Encabezado</h1>
    <p> Hola Mundo!</p>
  </body>
</html>
```

El programa Hola Mundo tiene ahora la siguiente salida, como puede ver en la Figura 1.14.

Hay tres formas de implementar una hoja de estilo CSS en un programa *HTML*: interno, externo, y archivo en línea.

- Interno en una etiqueta: se añade una etiqueta `<etiqueta style="">` en el programa *HTML* con la definición del estilo.
- Externo: En la sección `<head>` de la página se puede incluir una etiqueta `<style>` que integre todas las reglas de estilo de la página.
- Archivo: Mediante la etiqueta `<link>` se puede incluir un fichero externo que incluya todas las reglas.

En el primer caso sólo se deben especificar los estilos que se aplicarán al elemento en cuestión. Mientras que en los otros dos es necesario especificar, además, a qué elementos de la página se aplicarán los estilos. El último caso es el mostrado en esta sección.

CSS

Puede ampliar el tema de CSS y crear otros estilos revisando esta página [CSS](#)

Puede ampliar este tópico y crear otros estilos revisando la siguiente bibliografía referida en la nota al margen del texto y en [59, 85].

1.6.2.2 XHTML

XHTML significa lenguaje de marcado de hipertexto extensible (eXtensible HyperText Markup Language). Es una versión diferente de *HTML*, basada en *XML*. *XHTML* se usa para hacer páginas web y tiene una forma muy específica de escribirse para que sea correcto y sin errores. *XHTML* también distingue entre mayúsculas y minúsculas. Las etiquetas están escritas en minúsculas y deben cerrarse. El orden de las etiquetas también debe estar en el orden correcto para que la información salga correcta. Hay tres secciones principales de *XHTML* que consisten en una *declaración de declaración*, una *declaración principal* y un *cuerpo*, [77, 79].

Un documento *XHTML* se compone de cuatro componentes:

- Definición de tipo de documento (DTD): La DTD describe el idioma o la gramática en la que se ha codificado el texto. Es opcional en *HTML*, pero en *XHTML* es requerido. Pueden ser de tipo *Strict* (no soporta etiquetas antiguas y el código debe estar escrito correctamente), *Transitional* (como *XHTML Strict DTD*, pero las etiquetas en desuso están permitidas) y *Frameset* (la única DTD *XHTML* que soporta *Frameset*, que divide el espacio del documento) donde haya sido insertado, y permite la carga de otros documentos en cada espacio.
- Contenido de texto: los encabezados y párrafos que aparecen en la página.
- Referencias: contenido avanzado como enlaces e imágenes.
- Mark-Up: Instrucciones sobre cómo se debe mostrar el contenido.

Cada uno de estos componentes puede guardarse en formato de texto y verse en cualquier navegador. Para adaptar un documento *HTML* a *XHTML* solo debe agregar al inicio del programa el DTD, ver programa 1.6.2.2

Algoritmo 1.4: Hola Mundo con HTML y CSS

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<html>
```

```
<head>
```

```
<title>Mi primer Ejemplo</title>
<link rel="stylesheet" href="css/app.css">

</head>
<body>
    <h1>Este es el Encabezado</h1>
    <p> Hola Mundo!</p>
</body>
</html>
```

La diferencia de *XHTML* versus *HTML* son las siguientes:

- `<!DOCTYPE>` es obligatorio
- El atributo `xmlns` en `<html>` es obligatorio
- `<html>`, `<head>`, `<title>` y `<body>` son obligatorios
- Los elementos siempre deben estar correctamente anidados
- Los elementos siempre deben estar cerrados.
- Los elementos siempre deben estar en minúsculas
- Los nombres de los atributos siempre deben estar en minúsculas.
- Los valores de los atributos siempre se deben citar.
- La minimización de atributos está prohibida.

Est proporciona las siguientes ventajas en el uso de *XHTML*: 1) Se pueden incorporar elementos de distintos espacios de nombres XML (como MathML y Scalable Vector Graphics). 2) Un navegador no necesita implementar heurísticas para interpretar el texto por lo que el *parser* puede ser mucho más sencillo. Y 3) Se pueden utilizar fácilmente herramientas creadas para procesamiento de documentos XML genéricos (editores, XSLT, entre otras).

1.6.2.3 Páginas Dinámicas

Las páginas dinámicas [72, 90] son páginas *HTML* generadas a partir de lenguajes de programación (*script*) que son ejecutados en el propio servidor web. A diferencia de otros scripts, como *JavaScript*, que se ejecutan en el propio navegador del usuario, los scripts en el lado del servidor (*'Server Side' scripts*) generan un código *HTML* desde el propio servidor web.

Por su parte, las páginas estáticas no son páginas sin movimientos, son páginas que permanece tal y como fue diseñada. Sin embargo, una página web estática también puede proporcionar una experiencia de usuario "en vivo", "dinámica" o "interactiva". El contenido (texto, imágenes, campos de formulario, etc.) en una página web puede cambiar, en respuesta a diferentes contextos o condiciones.

Estos comportamientos se explican de la siguiente manera:

- página estática. Uso de secuencias de comandos del lado del cliente para cambiar los comportamientos de la interfaz dentro de una página web específica, en respuesta a las acciones del ratón o *mouse*, o del teclado o en eventos de tiempo específicos. En este caso, el comportamiento dinámico ocurre dentro de la presentación.
- página dinámica. Uso de secuencias de comandos en el lado del servidor para cambiar la fuente de la página proporcionada entre páginas, ajustando la secuencia o recarga de las páginas web o el contenido web proporcionado al navegador. Las respuestas del servidor pueden estar determinadas por condiciones tales como datos en un formulario HTML publicado, parámetros en la URL, el tipo de navegador que

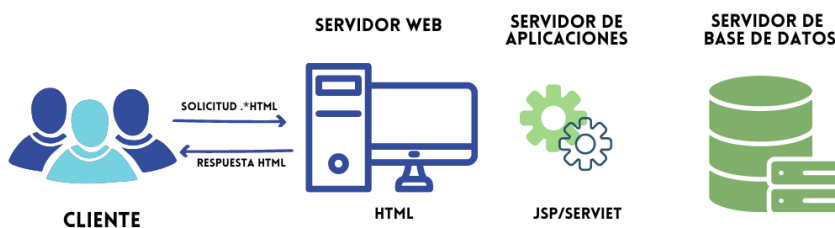
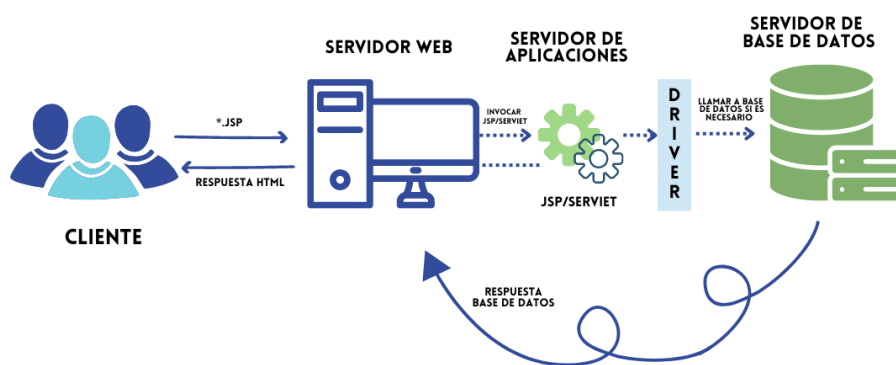
SOLICITUD EN PAGINA ESTATÁTICA**SOLICITUD EN PAGINA DINÁMICA**

Figure 1.15: Solicitud página estática y página dinámica. Adaptado de [72]

se utiliza, el paso del tiempo o una base de datos o el estado del servidor.

Sin embargo, si un usuario que envía una solicitud de página web no sabe si la solicitud que se envía se está devolviendo a una página web estática o una página web cuyo contenido es una mezcla de información estática y dinámica recuperada de una base de datos [72]. En la Figura 1.15 se ilustra cómo es el tratamiento de la información cuando se realiza un requerimiento al servidor.

JavaScript Javascript es el lenguaje de programación de la Web [66]. JavaScript es un lenguaje interpretado que es ejecutado por el navegador que utilizamos para ver las páginas. Eso hace posible desarrollar páginas dinámicas de muy diverso tipo, desde generadores de HTML, comprobadores de formularios, hasta programas que gestionen las capas de una página .

También se puede usar para actualizar partes del contenido de una página web sin obtener una versión completamente nueva de la página y volver a renderizarla. Estas actualizaciones dinámicas pueden ocurrir ya sea debido a una acción del usuario , o cuando el navegador adquiere nuevos datos del servidor que suministró la página web. En este ultimo caso, el momento de la llegada de los datos no está relacionado con ninguna acción del usuario en el propio navegador, por ello se denomina asíncrono. Una técnica conocida como *AJAX* se utiliza en tales casos.

JavaScript

Una breve historia de JavaScript en este vídeo [JavaScript](#)

AJAX *Ajax* (Asynchronous JavaScript and XML) [46] se refiere a un grupo de tecnologías que se utilizan para desarrollar aplicaciones web. Al combinar estas tecnologías, las páginas web son más receptivas puesto que los paquetes de datos que se intercambian con el servidor y las páginas web no se vuelven a cargar cada vez que un usuario realiza un cambio de entrada.

El término *Ajax* proviene de la agrupación de las tecnologías que lo sustentan [68]:

1. un canal de comunicación asíncrono entre el navegador y el servidor.
2. JavaScript y XML.

Ajax se compone de las siguientes tecnologías:

- Presentación basada en estándares usando XHTML y CSS.
- Visualización dinámica e interacción utilizando el **Modelo de Objetos del Documento** (DOM, Document Object Model) del navegador.
- Intercambio y manipulación de datos mediante XML y XSLT.
- Recuperación asíncrona de datos usando el objeto XMLHttpRequest.
- JavaScript que une todo.

En la Figura 1.16 se muestra como encajan estas tecnologías [55]. JavaScript mantiene unida la aplicación, definiendo el flujo de trabajo del usuario y la lógica de negocio. La interfaz de usuario se manipula y actualiza mediante el uso de JavaScript para gestionar el modelo de objetos de documento (DOM), redibujando y reorganizando continuamente los datos presentados a los usuarios y procesando sus interacciones basadas en el mouse y en el teclado. Las hojas de estilo en cascada (CSS) brindan una apariencia consistente a la aplicación para la manipulación del DOM. El objeto *XMLHttpRequest* se utiliza para hablar con el servidor de forma asíncrona, ejecutando las solicitudes de los usuarios y la obtención de datos actualizados mientras el usuario continúa operando.

En una aplicación web tradicional, las solicitudes HTTP que se inician mediante la interacción del usuario con la interfaz web, se envía a un servidor web. El servidor web procesa la solicitud y devuelve una página HTML al cliente. Durante el transporte HTTP, el usuario no puede interactuar con la aplicación web. Esta técnica tradicional para crear aplicaciones web funciona correctamente, pero cuando se realizan peticiones continuas al servidor, el usuario debe esperar a que se recargue la página con los cambios solicitados.

Ajax define un método de iniciar un cliente con la comunicación del servidor sin recargas de páginas. Proporciona una forma de permitir actualizaciones de página parciales mediante la creación de un elemento intermedio entre el cliente y el servidor. En la Figura 1.17 se visualiza un ejemplo de una interacción AJAX [46].

1. Un evento del lado del cliente desencadena un evento Ajax. Puede ser desde un simple evento onchange hasta alguna acción específica del usuario.
2. Se crea una instancia del objeto XMLHttpRequest. Con el método open(), se configura la llamada: la URL junto con el método HTTP deseado, generalmente GET o POST. La solicitud se activa a través de una llamada al método send().
3. Se realiza una solicitud al servidor. Esto podría ser una llamada a un servlet, un script CGI o cualquier técnica del lado del servidor.
4. El servidor puede hacer cualquier cosa que se le ocurra, incluso acceder a una base de datos u otro sistema

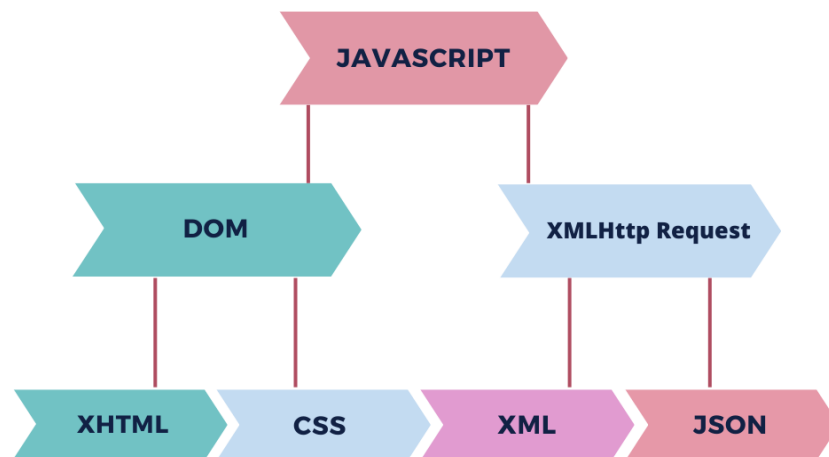


Figure 1.16: Componentes de AJAX. Adaptado de [55]

5. La solicitud se devuelve al navegador. El tipo de contenido se establece en tipo texto/xml (objeto XMLHttpRequest solo puede procesar resultados del tipo texto/html). La respuesta podría incluir ser más compleja e incluir JavaScript, manipulación DOM, u otras tecnologías relacionadas.
6. En este ejemplo, el objeto XMLHttpRequest llama a la función callback() cuando regresa el procesamiento. Esta función verifica la propiedad readyState en el objeto XMLHttpRequest y luego mira el código de estado devuelto por el servidor. Si todo es como se esperaba, la función de devolución de llamada podría hacer algo en el lado del cliente.

Este comportamiento es distinto de una interacción solicitud-respuesta. Desde la perspectiva del usuario de página web, significa una mejora de la interacción con una aplicación web, que proporciona al usuario más control de su entorno, y que es similar a la de una aplicación de escritorio.

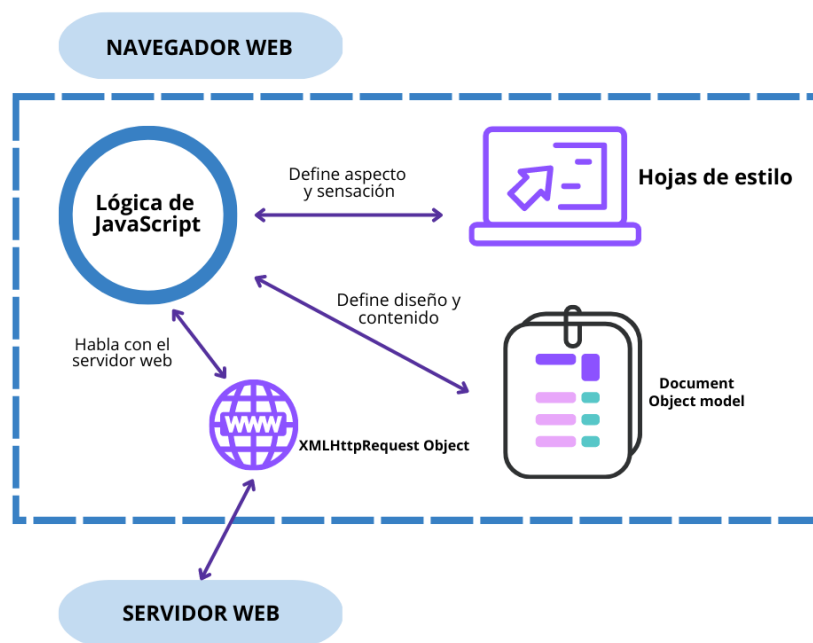


Figure 1.17: Una interacción AJAX. Adaptado de [46]



2. Arquitectura de los Sistemas Distribuidos

Los sistemas que están destinados a su uso en entornos del mundo real deben diseñarse para funcionar correctamente en la mayor variedad posible de circunstancias y frente a muchas posibles dificultades y amenazas.

Las propiedades y los problemas de diseño de los sistemas pueden capturarse y discutirse mediante el uso de modelos descriptivos. Cada tipo del modelo tiene la intención de proporcionar una descripción abstracta, simplificada pero consistente de un aspecto relevante del diseño de sistemas distribuidos. Los modelos descriptivos incluyen a los modelos físicos y modelos arquitectónicos, entre otros

2.1 Modelos Físicos

Los modelos físicos son la forma más explícita de describir un sistema; ellos capturar la composición de hardware de un sistema en términos de las computadoras, dispositivos, y sus redes de interconexión. Bajo esta óptica se puede identificar útilmente tres generaciones de distribuido sistemas [54].

Sistemas distribuidos tempranos Surgieron entre 1970 y principios de 1980 en respuesta a la aparición de la tecnología de redes de área local, generalmente Ethernet. Estos sistemas constaban de entre 10 y 100 nodos interconectado por una red de área local, con conectividad a internet limitada y compatible una pequeña gama de servicios, como impresoras locales compartidas y servidores de archivos, correo electrónico y transferencia de archivos a través de Internet.

Sistemas distribuidos tempranos

Son sistemas individuales, en gran medida homogéneos y la apertura no era la característica relevante. Las metas de diseño se enfocaban en la calidad del servicio [54]

Ejemplo de este tipo de sistema es la arquitectura cliente-servidor que tenía una red LAN y un solo cliente conectado a un servidor. El cliente solicita algún requerimiento al servidor, que era atendido y luego se enviaba la respuesta a cliente. En la figura 2.1 se ilustra esta arquitectura

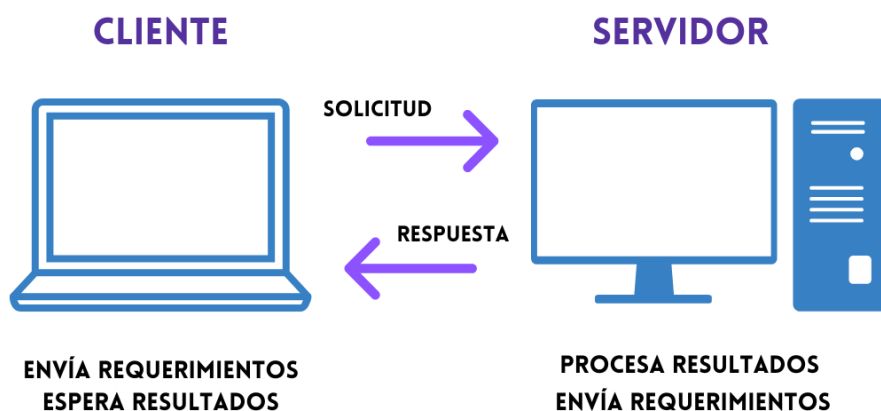


Figure 2.1: Arquitectura Cliente Servidor en red LAN

Sistemas distribuidos a escala de internet. Comenzaron a surgir en 1990 en respuesta al crecimiento de Internet. En estos sistemas la infraestructura física subyacente consiste en un modelo físico como un conjunto extensible de nodos interconectados por una red de redes (Internet). Incorporan grandes cantidades de nodos y proporcionar servicios de sistema distribuido para organizaciones globales y en toda la organización.

Sistemas distribuidos a escala de internet

Son sistemas heterogéneos en términos de redes, arquitectura de computadoras, sistemas operativos, idiomas empleados y equipos de desarrollo involucrados.

Los nodos pueden ser nodos estáticos, nodos discretos y nodos autónomos.

Ejemplos de modelos físicos de esta época están en:

- Arquitectura cliente-servidor. Puede presentar varios clientes conectados a un servidor, como el de la figura 2.2, varios clientes y servidores conectados a internet y conversando entre ellos.
- Arquitectura P2P. Constituido por un grupo de máquinas o nodos conectados entre sí, ver figura 2.3. Cada nodo cumple las mismas funciones y tiene las mismas responsabilidades que el resto de los nodos.

Algunas aplicaciones con esta arquitectura: [Ares](#), [bitcoin](#), [eDonkey](#).

Sistemas distribuidos contemporáneos Las tendencias claves, interoperabilidad y sistemas abiertos, han dado como resultado importantes desarrollos como:

- Informática móvil. Donde los nodos, tales como las computadoras portátiles o los teléfonos inteligentes, pueden moverse de una ubicación a otra en un

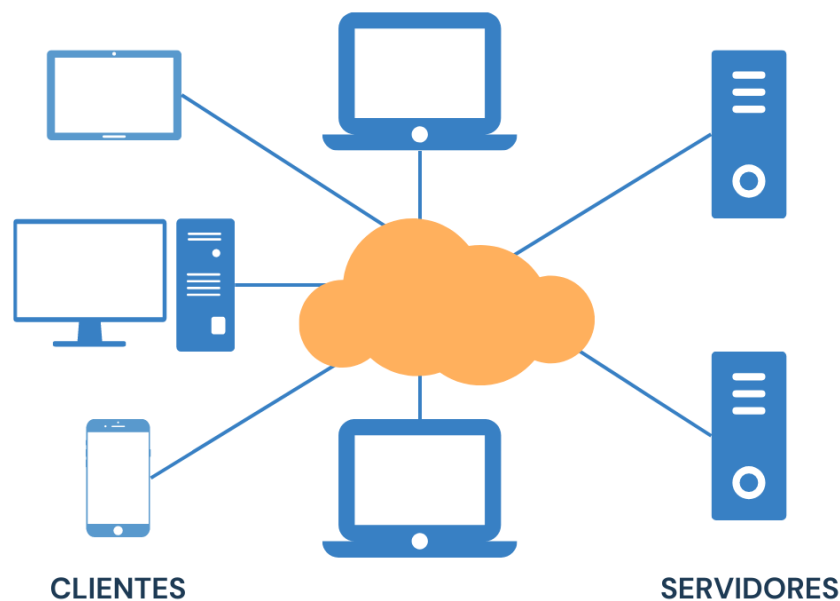


Figure 2.2: Arquitectura Cliente Servidor

sistema, lo que lleva a la necesidad de capacidades adicionales como el descubrimiento de servicios y apoyo a la interoperación espontánea.

- Computación ubicua. La computación ubicua ha llevado a pasar de nodos discretos a arquitecturas donde las computadoras están incrustadas en objetos cotidianos.
- Computación en la Nube. En computación en la nube, y, en particular, las arquitecturas de clúster ha llevado a un movimiento que va desde nodos autónomos que realizan un rol determinado a grupos de nodos que juntos brindan un servicio específico.

Algunas empresas que prestan este servicio: [Google](#) y [Amazon](#).

Sistemas de Sistemas Distribuidos. Son sistemas distribuidos complejos y a gran escala (arquitecturas físicas y redes de redes). Un sistema de sistemas se puede definir como un sistema complejo que consiste en una serie de subsistemas que son sistemas por derecho propio y que se unen para realizar una tarea o tareas particulares. Ejemplos son sistemas que usan bases de datos distribuidas en tiempo real, como en [33] donde se relaciona las publicaciones que hacen las personas en Twitter (tweets) relacionadas con un terremoto, lo que permite detectar rápidamente la ocurrencia de un terremoto y proponen un algoritmo para monitorear tweets y detectar un evento objetivo.

Y en [3] presentan un sistema basado en redes de sensores ambientales predictivos donde se presenta una aplicación para predecir de inundaciones de ríos.

2.2 Modelos Arquitectónicos

Los modelos arquitectónicos clasifican su contenido en componentes de la arquitectura: entidades, paradigmas de comunicación, roles y responsabilidades, y correspondencia con

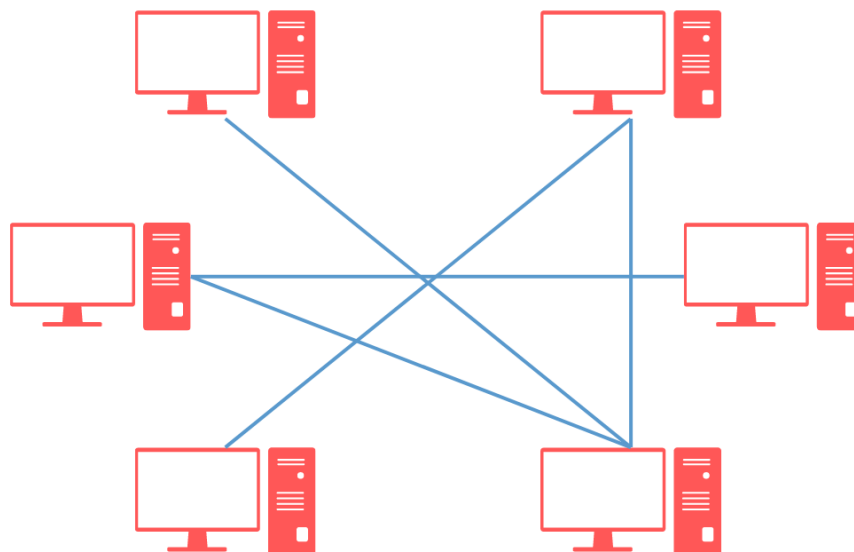


Figure 2.3: Arquitectura Punto a Punto

la infraestructura física [54].

A continuación se describe los modelos arquitectónicos de los sistemas distribuidos de acuerdo a esa caracterización.

2.2.1 Entidades

Las entidades presentes en un sistema distribuidos son: procesos, nodos, hilos, componentes, servicios web. Seguidamente, se describen.

Procesos Conducen a visión predominante de un sistema distribuido como procesos acoplados y paradigmas de comunicación entre procesos. Por ejemplo[88], en la figura 2.4 se ilustra la organización de una base de datos una red de monitoreo y los procesos que se ejecutan en ella, en el lugar del operador (a), o en los sensores (b).

Nodos En algunos entornos primitivos, como las redes de sensores, los sistemas operativos pueden que no admitan abstracciones de proceso y, por lo tanto, las entidades que se comunican en este tipo de sistemas son nodos. La Figura 2.5 muestra la organización de los nodos en la arquitectura de Chord [38], un protocolo y algoritmo para sistemas P2P con tablas hash distribuidas y que proporciona un servicio de búsqueda descentralizado que almacena pares clave/valor para esas redes.

Chord

Es un protocolo y algoritmo para la implementación de tablas hash distribuidas para sistemas P2P. Ver [Chord](#)

Hilos En la mayoría de los entornos de sistemas distribuidos, los procesos se complementan con hilos, que son los puntos finales de la comunicación. En la Figura 2.6 se presenta los hilos que se establecen en la comunicación del cliente y el servidor con

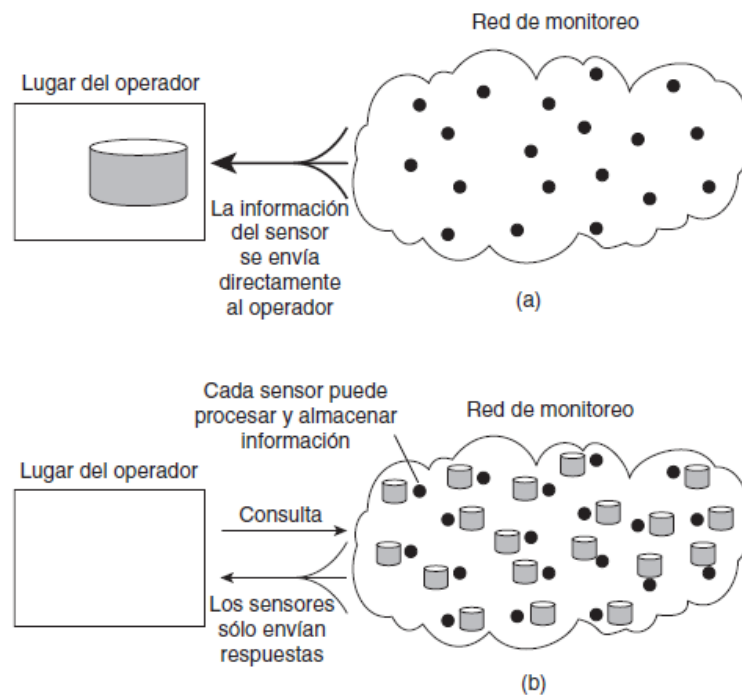


Figure 2.4: Organización de una base de datos para una red de monitoreo, mientras se almacena y procesa información (a) sólo en el lugar del operador, o (b) sólo en los sensores. Tomado de [88]

el protocolo solicitud-respuesta.

Objetos En el enfoque de sistemas distribuido basado en objetos, un cálculo consiste en una serie de objetos interactivos que representan unidades para el dominio del problema dado. Se accede a los objetos a través de interfaces, con un lenguaje de definición de interfaz asociado (*interface definition language, IDL*) que proporciona una especificación de los métodos definidos en un objeto. Por ejemplo, en la Figura 2.7, se muestra la constitución de la arquitectura de Corba, detallando los objetos y sus funciones [9].

Componentes. En los componentes de software, cada componente es una unidad de composición con interfaces especificadas contractualmente y dependencias de contexto explícitas únicamente. Esto indica que no hay dependencias implícitas.

Los componentes de software son como objetos distribuidos en el sentido de que están encapsulados en unidades de composición, pero un componente dado especifica tanto sus interfaces proporcionadas al mundo exterior y sus dependencias de otros componentes en el entorno distribuido [52]. La Figura 2.8 muestra la arquitectura de un sistema de archivos simple que proporciona una interfaz a otros usuarios y, a su vez, que requiere conexión a un componente de servicio de directorio y un componente de servicio de archivo plano.

Servicios Web. Los servicios web están estrechamente relacionados con objetos y componentes, adoptando un enfoque basado en la encapsulación de comportamiento y acceso a través de interfaces. Están integrados en la *World Wide Web*, utilizando estándares web para representar y descubrir servicios [60].

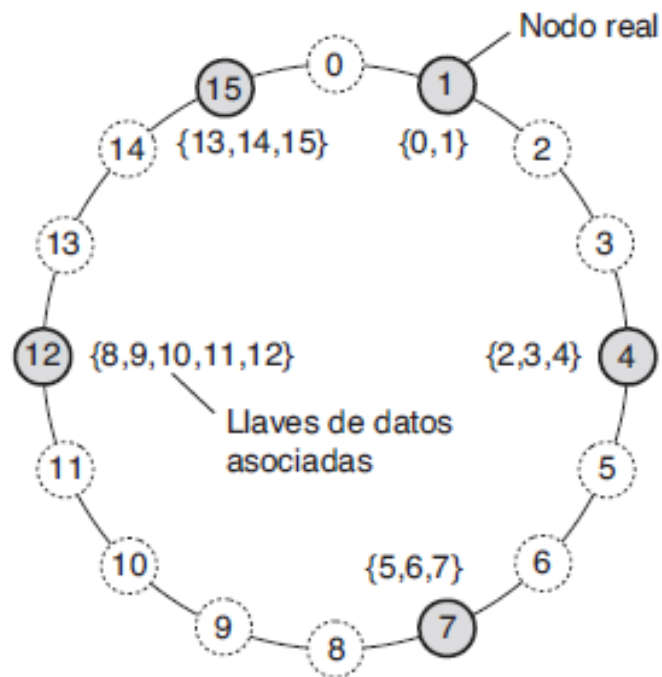


Figure 2.5: Mapeo de elementos de datos hacia nodos organizados en Chord. Tomado de [88]

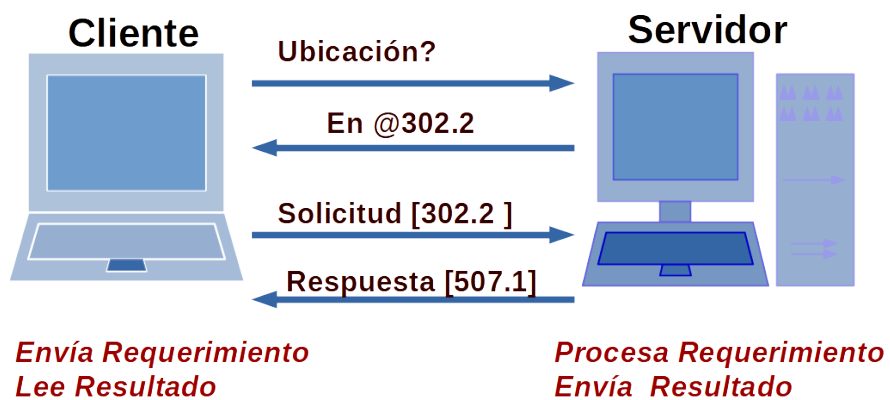


Figure 2.6: Hilos en Arquitectura Cliente Servidor

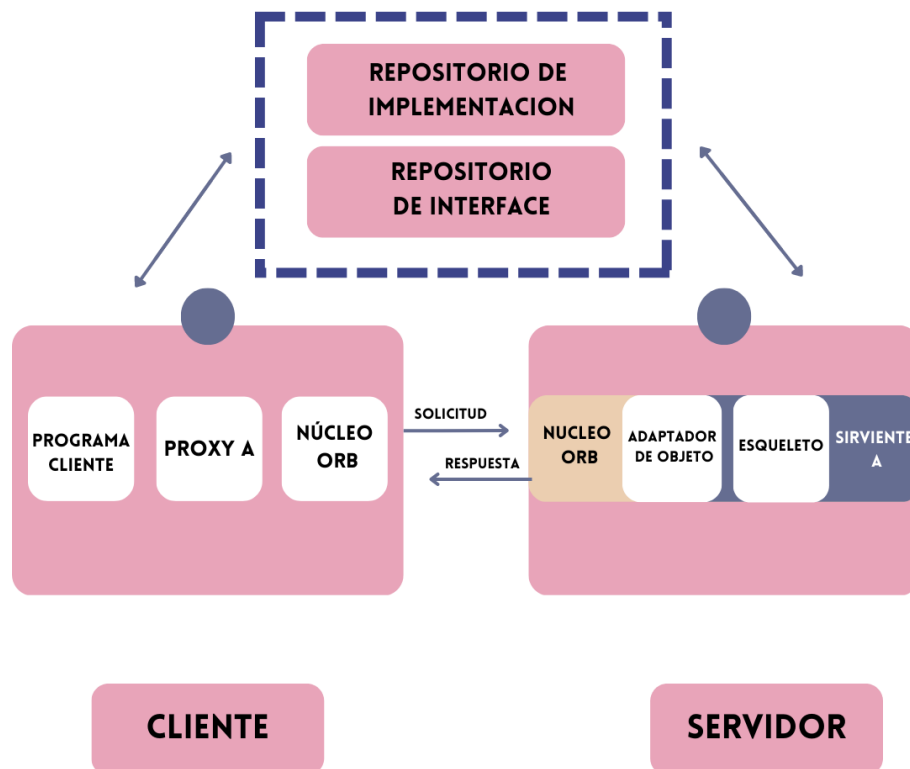


Figure 2.7: Componentes de la arquitectura CORBA.

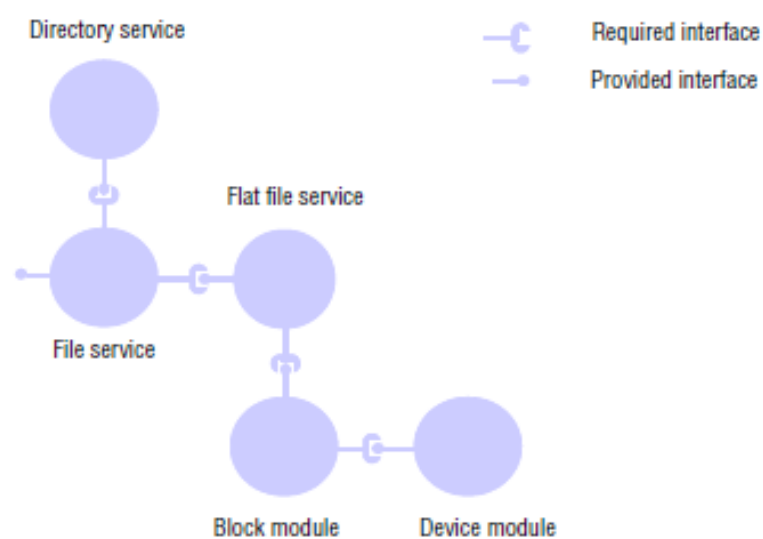


Figure 2.8: Arquitectura de software basado en componentes. Tomado de [54]

En [40] lo definen como un sistema software diseñado para soportar la interacción máquina-a-máquina, a través de una red, de forma interoperable. Cuenta con una interfaz descrita en un formato procesable por un equipo informático (específicamente en WSDL), a través de la que es posible interactuar con el mismo mediante el intercambio de mensajes SOAP, típicamente transmitidos usando serialización XML sobre HTTP conjuntamente con otros estándares web.

Servicios WEB

Servicios WEB, SOA, es un estándar de la OMG. Puede verlo en [SOA](#)

2.2.2 Paradigma de comunicación

Considere tres tipos de paradigma de comunicación: comunicación entre procesos, Invocación remota y comunicación indirecta [91] [13] [86] .

Comunicación entre procesos Se refiere al soporte de bajo nivel para comunicación entre procesos en sistemas distribuidos .

Comunicación entre procesos

- Pase de mensaje
- Sockect
- Multidifusión

Invocación remota Representa el paradigma de comunicación más común en sistemas distribuidos, cubren una gama de técnicas basadas en un intercambio bidireccional entre entidades comunicantes.

Invocación remota

Incluye:

- Protocolo Solicitud-respuesta
- Llamada a procedimientos remotos
- Llamada a métodos remotos

Protocolo solicitud-respuesta . El protocolo solicitud-respuesta es un patrón impuesto en un servicio subyacente de transmisión de mensajes para admitir la informática cliente-servidor. Este paradigma es bastante primitivo y se usa en sistemas embebidos donde el rendimiento es primordial. El enfoque también se utiliza con el protocolo HTTP.

Llamadas a procedimiento remoto . (*Remote Procediment Call, RPC*) es computación cliente-servidor con servidores que ofrece un conjunto de operaciones a través de un interfaz de servicio y clientes que llaman a estas operaciones directamente como si estuvieran disponibles en la zona.

Invocación de métodos remotos . La invocación de métodos remotos (*Remote Method Invocation, RMI*) se parece mucho a llamadas a procedimiento remoto pero en un mundo de objetos distribuidos

Comunicación Indirecta Se realiza a través de una tercera entidad, lo que permite un fuerte grado de desacoplamiento entre remitentes y receptores. En este paradigma los remitentes de los mensajes no necesitan saber a quién están enviando mensajes;

y no es necesario que los emisores y los receptores existan al mismo tiempo.

Comunicación indirecta

Comprende los siguientes

- Comunicación grupal.
- Sistema Publicación-Suscripción.
- Cola de Mensaje.
- Espacios de Tupla.
- Memoria compartida Distribuida.

Comunicación grupal. La comunicación grupal se basa en la abstracción de un grupo que está representado en el sistema por un identificador de grupo.

Sistemas de publicación-suscripción. Los sistemas pub-sub (publicación-suscripción) donde los productores distribuyen elementos de información de interés o eventos, a los consumidores (llamado también, sistemas basados en eventos distribuidos). Ver esquema de la arquitectura en la Figura 2.9.

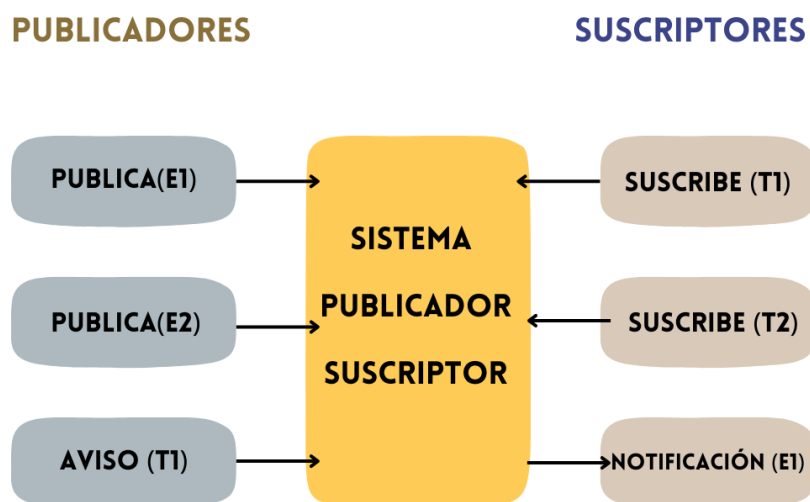


Figure 2.9: Arquitectura Publicación Suscripción.

Colas de mensajes. Servicio punto a punto mediante el cual el productor puede enviar mensajes a una cola específica y el consumidor puede recibir mensajes de la cola o ser notificado de la llegada de nuevos mensajes en la cola. Ver Figura 2.10.

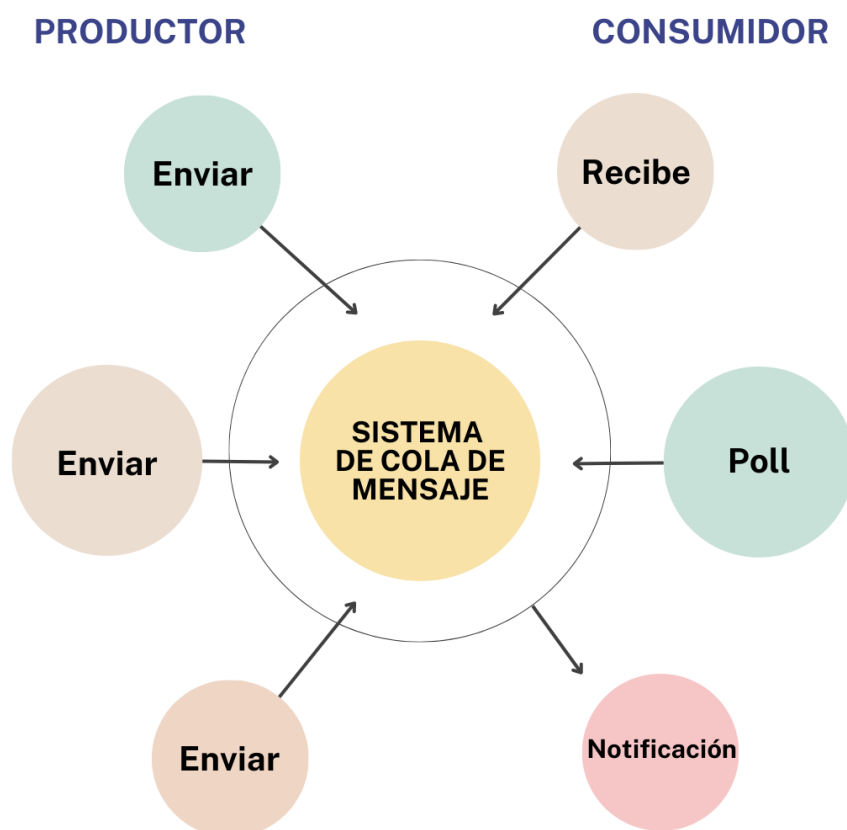


Figure 2.10: Arquitectura Colas de Mensajes.

Espacios de tupla. Los procesos pueden colocar elementos arbitrarios de datos estructurados o tuplas, en un espacio de tuplas persistente; otros procesos pueden leer o eliminar tales tuplas del espacio de tuplas especificando patrones de interés.

Memoria compartida distribuida. Proporciona una abstracción para compartir datos entre procesos que no comparten memoria física. A los programadores se les presenta una abstracción de lectura o escritura de estructuras de datos (compartidas) como si estuvieran en sus propios espacios de direcciones locales.

2.2.3 Roles y responsabilidades

Los estilos de arquitectura según los roles y responsabilidades que cumplen los nodos, incluyen:

- Rol que juegan los nodos ya sea como clientes y servidores en la arquitectura cliente-servidor.
- Mismas funciones que cumplen los nodos en una arquitectura p2p, por ejemplo (sección 2.1).
- Rol como publicadores o suscriptores de eventos en una arquitectura pub-sub (ver descripción en sección 2.1).
- Rol como productores o consumidores de mensajes en una arquitectura basada en

colas (ver descripción en sección 2.1).

2.2.4 Correspondencia con la infraestructura física distribuida

En esta clasificación se considera cómo las entidades, objetos o servicios se asignan o forman parte de la infraestructura física distribuida subyacente. Se exploran tres aspectos [54], [90]:

- Elementos que conforman la arquitectura
- Patrones que usa la arquitectura y
- Soluciones basadas en middleware

Elementos de la Arquitectura

Comprende los siguientes

- Correspondencia de servicios con servidores.
- Memoria Cache.
- Código Móvil.
- Agentes Móviles.

Elementos de la Arquitectura

Correspondencia de servicios con servidores. Los servicios pueden implementarse como varios servidores de procesos en computadoras separadas que interactúan según sea necesario para proporcionar un servicio a procesos del cliente. En la Figura 2.11 se muestra un ejemplo.

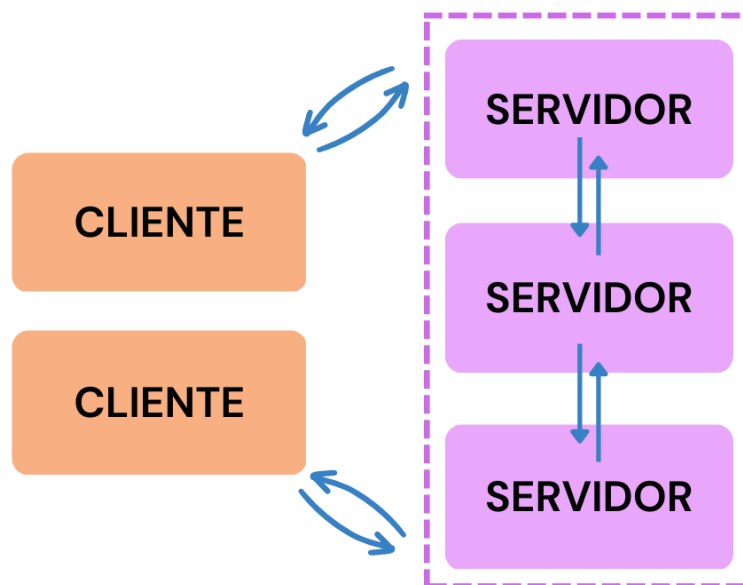


Figure 2.11: Arquitectura con múltiples servidores. Tomado de [54]

Cache. Los navegadores web mantienen un caché con las páginas web visitadas recientemente y otros recursos web en el sistema de archivos local del cliente, utilizando una solicitud HTTP para verificar con el servidor original que las páginas en caché

están actualizadas. Un servidor proxy web (ver Figura 2.12) proporciona un caché compartido de recursos web para las máquinas cliente en un sitio o en varios sitios.

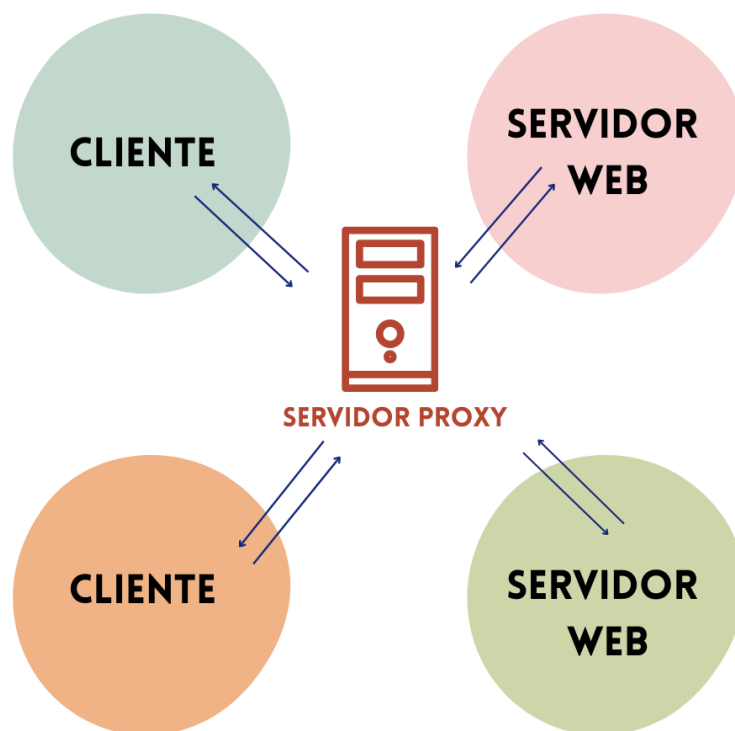


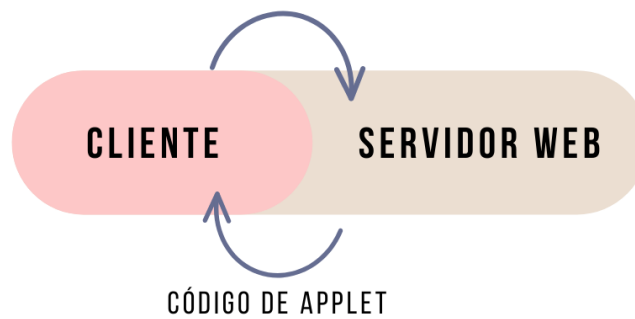
Figure 2.12: Arquitectura web proxy.

Código móvil El applet es un ejemplo de código móvil: el usuario que ejecuta un navegador selecciona un enlace a un subprograma cuyo código se almacena en un servidor web; el código se descarga en el navegador y lo ejecuta, como se muestra en la Figura 2.13.

Agentes móviles El agente móvil puede realizar invocaciones a los recursos locales en cada sitio que visita, por ejemplo, acceder a entradas individuales de la base de datos. Los agentes móviles pueden usarse para instalar y mantener software en las computadoras dentro de una organización o para comparar los precios de productos de varios proveedores visitando el sitio de cada proveedor y realizando una serie de operaciones de base de datos.

Patrones de Arquitectura Los patrones arquitectónicos [80] dan una descripción de los elementos y el tipo de relación que tienen junto con un conjunto de restricciones. Un patrón arquitectónico expresa un esquema de organización estructural esencial para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones. No son necesariamente soluciones completas en sí mismas, sino que ofrecen conocimientos parciales que, cuando se combinan con otros patrones, llevan al diseñador a una solución para un dominio de problema dado.

1. LA SOLICITUD DEL CLIENTE DA COMO RESULTADO LA DESCARGA DEL CÓDIGO DEL APPLET



2. CLIENTES INTERACTÚAN CON EL APPLET



Figure 2.13: Arquitectura web applet.

Patrones de Arquitectura

Se presentan los siguientes:

- Capas
- Arquitectura de capas
- Cliente flacos
- Patrón Proxy
- Brokerage
- Balanceador de carga con réplicas (*backend*)
- Servidor con réplicas
- Árbol de servidores

Capas En un enfoque por capas, un sistema complejo se divide en varias capas, con una capa dada haciendo uso de los servicios ofrecidos por la capa siguiente. La capa referida ofrece una abstracción de software, con capas superiores que desconocen detalles de su implementación, o de cualquier otra capa debajo de ellos. En términos de sistemas distribuidos, esto equivale a una organización vertical de servicios en capas de servicio. Ejemplo de esta estructura es el middleware.

Arquitectura de capas Las arquitecturas de capas es una técnica para organizar la funcionalidad de una capa determinada y colocarla en servidores apropiados y, como consideración secundaria, en los nodos físicos. Esta técnica se asocia más comúnmente con la organización de aplicaciones y servicios. Por ejemplo, en la 2.14 se observa una arquitectura de dos capas, cliente y servidor; mientras la Figura 2.15 es el esquema de una arquitectura de tres capas: cliente, servidor y servidor de base de datos.

Clients flacos La tendencia en la computación distribuida es alejar la complejidad del dispositivo del usuario final hacia los servicios en Internet. Esto es más evidente en la computación en la nube, pero también se puede ver en la arquitectura de niveles. Esta tendencia ha suscitado el interés en cliente ligero, que permite el acceso a sofisticados servicios en red, proporcionados por una solución en la nube, con pocas suposiciones o demandas en el dispositivo del cliente, entre otros. La Figura 2.16 ilustra un cliente ligero que accede a un servidor informático a través de Internet.

Patron Proxy El patrón proxy es un patrón recurrente en sistemas distribuidos, diseñados para apoyar la transparencia de la ubicación en llamadas de procedimiento remoto (RPC) o invocación del método (RMI). Es un intermediario entre un objeto y el resto que lo invoque.

Brokerage El uso de corredores o *brokerage* en servicios web puede verse como un patrón que admite la interoperabilidad en infraestructuras distribuidas potencialmente complejas. Este patrón consta del trío de proveedores de servicios, solicitante de servicios y corredor de servicios (un servicio que coincide con los servicios prestados a los solicitados), como se muestra en la Figura 2.17.

Otros Patrones En [90], el autor presenta una clasificación de las arquitecturas de los sistemas distribuidos basada en patrones. Estos patrones forman parte de la estructura de los sistemas distribuidos, considere que los mismos están compuesto por múltiples componentes que aportan la funcionalidad que presentan los sistemas distribuidos, como la disponibilidad y escalamiento de servicios. Se detallan tres patrones en particular:

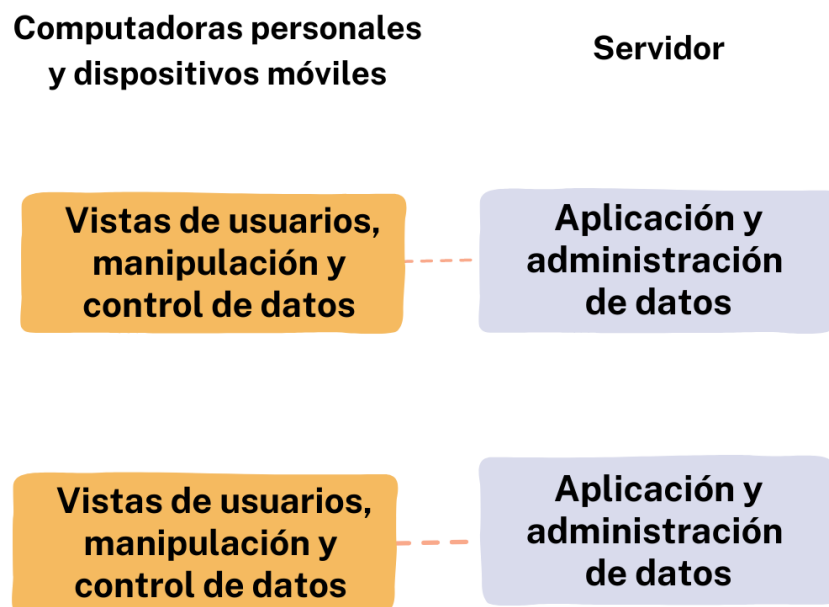


Figure 2.14: Arquitectura de capas de dos niveles.

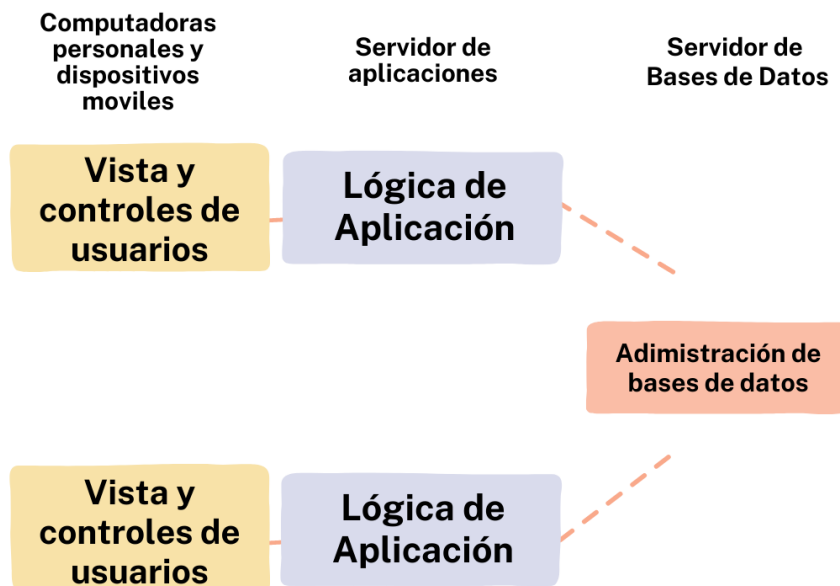


Figure 2.15: Arquitectura de capas de tres niveles.

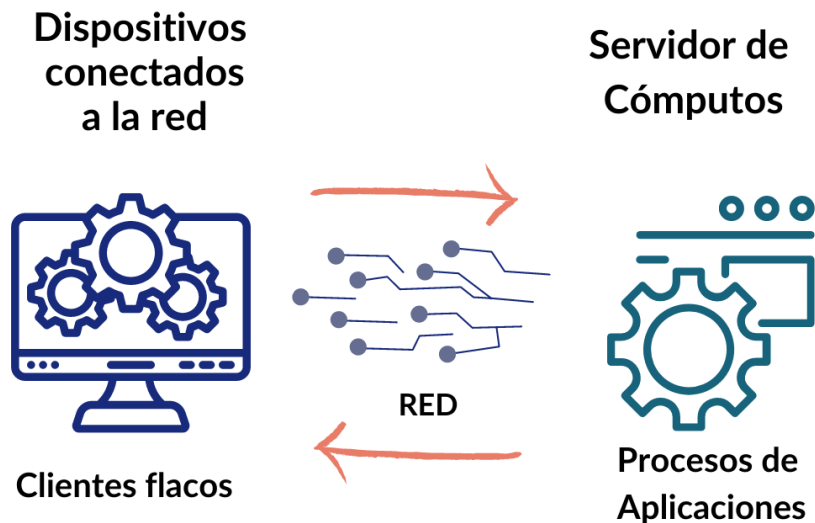


Figure 2.16: Arquitectura de cliente ligero.

Balanceador de carga con múltiples réplicas (*backend*)

Los patrones que se muestran a partir de este, son parte de la estructura de los sistemas distribuidos presentados en [90]. En el patrón de composición del balanceador de carga con múltiples réplicas, ver la Figura 2.18, las solicitudes se envían al servidor del equilibrador de carga. Para cada solicitud, selecciona un *backend* y se reenvía la solicitud allí. La respuesta vuelve al servidor del equilibrador de carga, que a su vez la transmite al solicitante original. Una solicitud enviada a cualquier réplica debería producir la misma respuesta.

Servidor con varias réplicas El siguiente patrón de composición es un servidor con múltiples réplicas o *backends*. El servidor recibe una solicitud, envía consultas a muchos servidores réplicas y redacta la respuesta final combinando esas respuestas. Este enfoque se utiliza normalmente cuando la consulta original se puede descomponer en una serie de consultas independientes que se pueden combinar para formar la respuesta final.

La Figura 2.20 de la izquierda ilustra cómo un motor de búsqueda simple procesa una consulta con la ayuda de múltiples respaldos. La interfaz recibe la solicitud. Transmite la consulta a varios servidores respaldos.

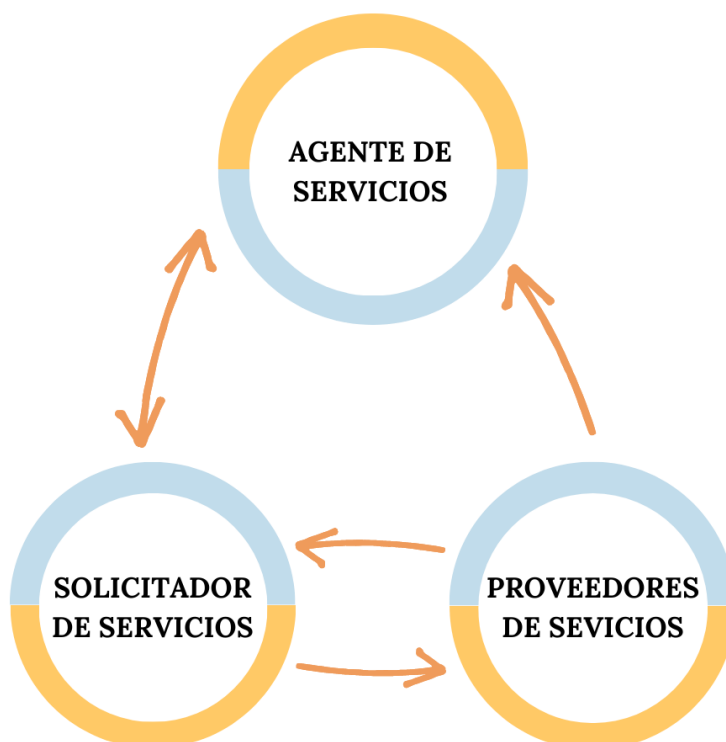


Figure 2.17: Patrón arquitectónico en servicios web.

Búsqueda en servidores replicados

Si la búsqueda es textual, el corrector ortográfico responde con información para que el motor de búsqueda sugiera ortografías alternativas. Los backends de búsqueda de imágenes y web responden con una lista de sitios web e imágenes relacionadas con la consulta. Una vez que se reciben las respuestas, la interfaz utiliza esta información para construir el HTML que forma la página de resultados de búsqueda para el usuario, que se envía como respuesta.

La Figura 2.20, esquina derecha, ilustra la misma arquitectura con backends replicados y con equilibrio de carga. Se aplica el mismo principio, pero el sistema puede escalar y sobrevivir mejor a las fallas.

Árbol de servidores El otro patrón de composición fundamental es el árbol de servidores. Como ilustra la Figura ??, en este esquema varios servidores trabajan en cooperación con uno como la raíz del árbol, los servidores padres debajo de él y los servidores hoja en la parte inferior del árbol.

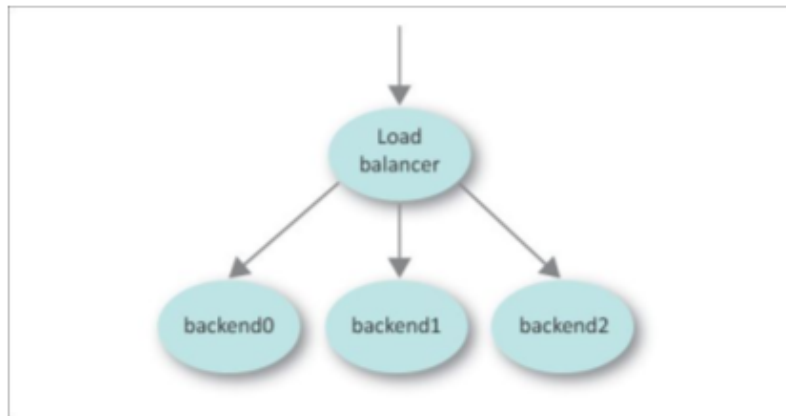


Figure 2.18: Balanceador de Cargas con réplicas. Tomado de [90]

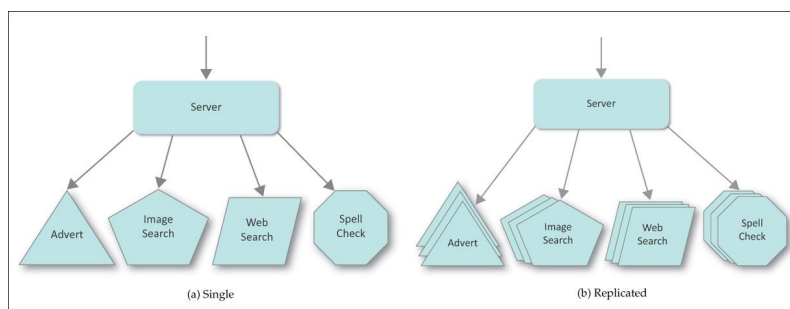


Figure 2.19: Balanceador de Cargas. Tomado de [90]

Consultas en árbol de servidores

Este patrón se utiliza para acceder a un gran conjunto de datos o corpus. Cada hoja almacena una fracción de los datos. La raíz recibe la consulta original y la reenvía a los padres, quienes la reenvían a los servidores hoja. Cada hoja envía sus hallazgos a los padres, que clasifican y filtran los resultados antes de enviarlos a la raíz. La raíz toma las respuestas, combina los resultados y responde.

Brokerage

Puede leer más de este tema en: [Red Hat](#)

2.3 Soluciones basadas en middleware

La tarea del middleware es proporcionar un nivel superior abstracción de programación para el desarrollo de sistemas distribuidos y, con las capas, abstraer la heterogeneidad en la infraestructura subyacente para promover interoperabilidad y portabilidad. Además de las soluciones de middleware presentados, también hay arquitecturas más complejas como las siguientes:

- Objetos Distribuidos: Corba, Java RMI
- Componentes distribuidos: Fractal

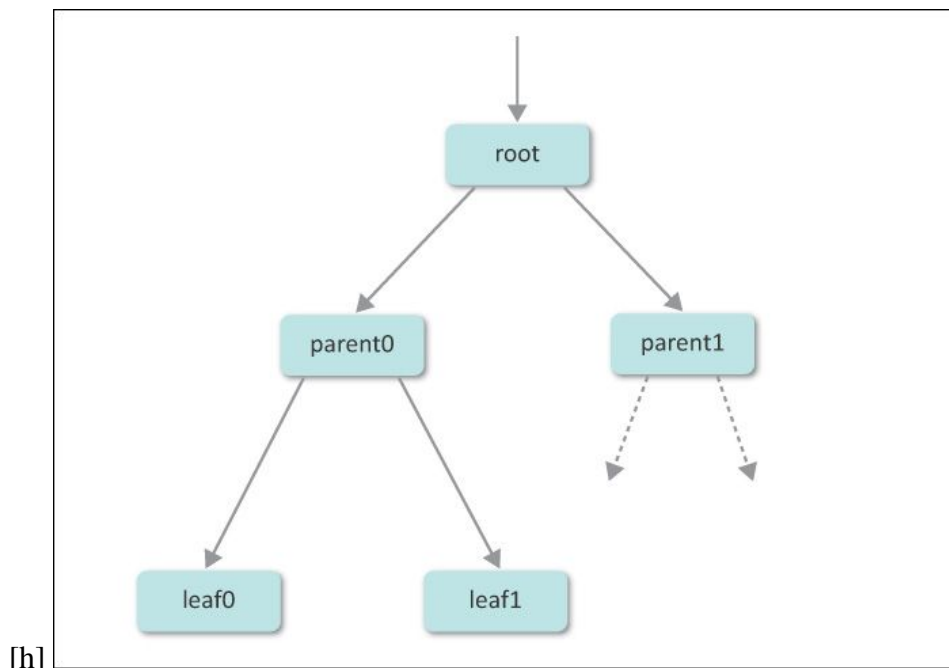


Figure 2.20: Servidores Replicados. Tomado de [90]

- Servidores de aplicaciones: Sun EJB, JBoss
- Servicios Web: Apache Axis
- Publicación-suscripción: Event Service, JMS
- Colas de mensajes: Websphere MQ
- Peer-to-peer: PASTRY, Tapestry, OceanStore, Gnutella

2.4 Caso de Estudio: Sistema de Nombres de Dominio

El sistema de Nombres de Dominios (DNS, Domain Name Systems) es un sistema de nombres que asigna nombres a otros datos, como direcciones IP, información de enrutamiento de correo y más. Es una base de datos distribuida cuya finalidad es permitir el control local de los segmentos de la base de datos general, los datos de cada segmento están disponibles en toda la red a través de un esquema cliente/servidor [75] [74]. La robustez y el rendimiento adecuado del DNS se logran mediante la replicación de servidores y el almacenamiento de datos en caché.

DNS también es un sistema cliente-servidor, con clientes DNS consultando servidores DNS para recuperar datos almacenados en esa base de datos distribuida. Los clientes DNS se denominan resolutores, mientras que los servidores DNS a veces se denominan servidores de nombres. Toda la base de datos (o sistema de archivos) se representa como un árbol invertido, con el nodo raíz en la parte superior. Cada nodo del árbol tiene una etiqueta de texto que identifica el nodo en relación con su padre, ver Figura 2.21. En la Figura 2.21 se visualiza la estructura del árbol que caracteriza al DNS. Cada nodo representa nodos y las hojas pueden ser nodos o recursos. Los nodos de ubicados a la izquierda se refieren a los dominios genéricos: com, edu, org, gov; los nodos hacia la derecha se muestra los servidores de nombres dedicados a países, por ejemplo: ac, al, co, pe, ve, entre otros.

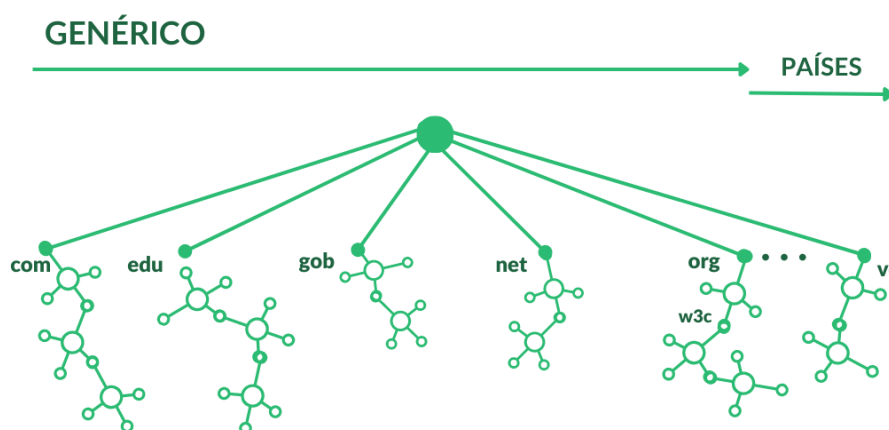


Figure 2.21: Sistemas de Nombres de Dominio.

2.4.1 Términos asociados al DNS

Dominio. Refiriéndonos a la Figura 2.21, cada nodo es la raíz de un nuevo subárbol del árbol general. Cada uno de estos subárboles representa una partición de la base de datos general: un **dominio** en el DNS. Cada dominio o directorio se puede dividir en particiones adicionales, llamadas **subdominios**, como los subdirectorios de un sistema de archivos. Los subdominios, como los subdirectorios, se dibujan como elementos secundarios de sus dominios principales. Por ejemplo, en la Figura 2.21, el dominio **org** y el subdominio **w3c**.

Espacio de nombres de dominio. La base de datos distribuida de DNS está indexada por nombres de dominio. El servicio intentará buscar un nombre válido, aunque se demuestre que ese nombre no corresponde a ningún objeto, es decir, que no poseeé algún vínculo. Cada nombre de dominio es esencialmente solo una ruta en el árbol, llamado **espacio de nombres de dominio**. Entonces, un dominio es un subárbol del espacio de nombres del dominio. El nombre de dominio de un dominio es el mismo que el nombre de dominio del nodo en la parte superior del dominio, por ejemplo, la parte superior del subarbol **w3.org/** es un nodo de nombre **w3.org/**. Y un **espacio de nombres** es la colección de todos los nombres válidos reconocidos por un servicio en particular. El nombre de dominio completo de cualquier nodo en el árbol es la secuencia de etiquetas en la ruta desde ese nodo hasta la raíz. Ejemplo **https://www.w3.org/**, sería la ruta para la hoja principal, dentro del subárbol **org** de la Figura 2.21.

Alias. Un **alias** es un nombre de dominio definido para denotar información sobre el servidor. Los alias permiten sustituir nombres más convenientes por otros menos complicados y permiten que distintas personas utilicen nombres alternativos para la misma entidad.

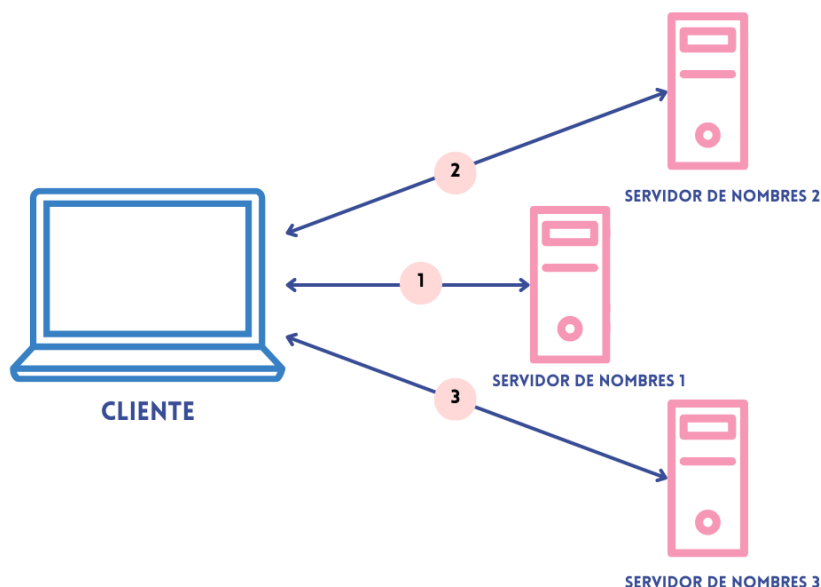


Figure 2.22: DNS: Búsqueda Interactiva desde el cliente.

Nombres de Dominios. Nombres de dominios es un espacio de nombres para el cual existe una única autoridad administrativa general responsable de asignar nombres dentro de él. Esta autoridad tiene el control general de qué nombres pueden vincularse dentro del dominio, pero también puede delegar esta tarea. Los dominios en DNS son colecciones de nombres de dominio; sintácticamente, el nombre de un dominio es el sufijo común de los nombres de dominio que contiene, pero por lo demás no se puede distinguir, por ejemplo, de un nombre de computadora.

Resolución El proceso de obtención de una dirección de Internet a partir de un nombre de sistema principal se conoce como **resolución de nombres**. La resolución de nombres es un proceso iterativo o recursivo mediante el cual un nombre se presenta repetidamente a contextos de nombres para buscar los atributos a los que se refiere.

El proceso de localizar datos de nombres en más de un servidor de nombres para resolver un nombre se llama **navegación**. El software de resolución de nombres de clientes realiza la navegación en nombre del cliente. Se comunica con los servidores de nombres según sea necesario para resolver un nombre. Los modelos de navegación que soporta el DNS son los siguientes [54]:

- Navegación Iterativa desde el cliente. Para resolver un nombre, un cliente presenta el nombre al servidor de nombres local, que intenta resolverlo. Si el servidor de nombres local tiene el nombre, devuelve el resultado inmediatamente. Si no es así, sugerirá otro servidor que podrá ayudar. La resolución procede en el nuevo servidor, con más navegación según sea necesario hasta que se localiza el nombre o se descubre que no está vinculado, ver esquema en la Figura 2.22
- Navegación Iterativa controlada por el servidor.

En la Figura 2.23, se muestra otra alternativa al modelo de navegación iterativa, en la que un servidor de nombres coordina la resolución del nombre y devuelve el resultado al usuario. Bajo la navegación iterativa controlada por el servidor, el

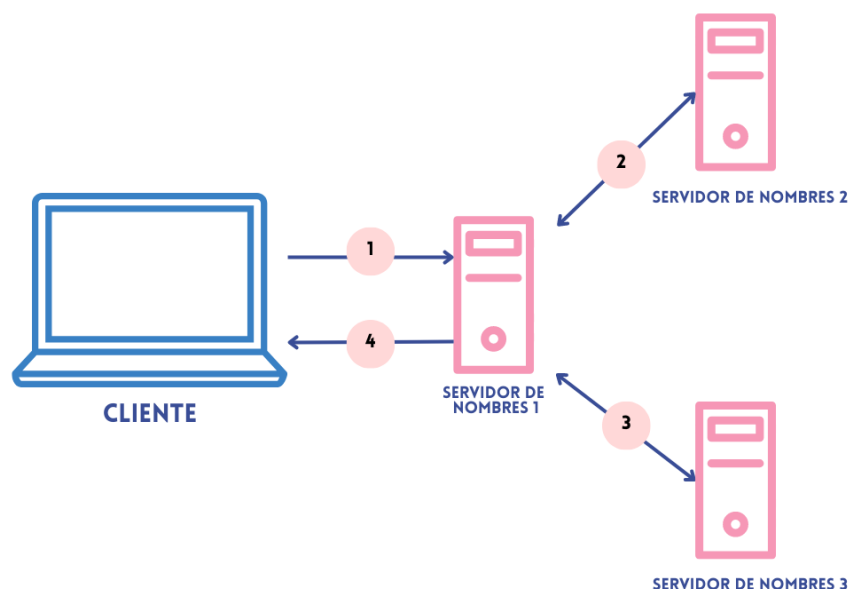


Figure 2.23: DNS: Búsqueda Interactiva desde el servidor.

cliente puede elegir cualquier servidor de nombres. Este servidor se comunica iterativamente o por multidifusión con sus pares, como si fuera un cliente.

- Navegación Recursiva controlada por el servidor. En la navegación recursiva controlada por el servidor, ver Figura 2.24, el cliente contacta a un solo servidor. Si este servidor no almacena el nombre, el servidor contacta a un compañero que almacena un prefijo del nombre, que a su vez intenta resolverlo. Este procedimiento continúa recursivamente hasta que se resuelve el nombre.

Ejemplo de una resolución de nombres En el DNS, el software de resolución de nombres de clientes y servidores mantienen un **caché** de los resultados de resoluciones de nombres anteriores. Cuando un cliente solicita una búsqueda de nombre, el software de resolución de nombres consulta su caché. Si contiene un resultado reciente de una búsqueda anterior del nombre, lo devuelve al cliente; de lo contrario, se pone a buscarlo desde un servidor. El almacenamiento en caché es clave para el rendimiento de un servicio de nombres y ayuda a mantener la disponibilidad tanto del servicio de nombres como de otros servicios a pesar de las caídas del servidor de nombres.

En la Figura 2.25 se muestra el proceso de resolución de la dirección de un host en un dominio. El servidor de nombres local consulta a un servidor de nombres raíz la dirección de *ejemplo.info.com.ve* y se remite a los servidores de nombres **ve**. El servidor de nombres local le pregunta a un servidor de nombres **ve** la misma pregunta, y se remite a los servidores de nombres **com.ve**. El servidor de nombres **com.ve** remite el servidor de nombres local a los servidores de nombres **info.com.ve**. Finalmente, el servidor de nombres local pide la dirección a un servidor de nombres **info.com.ve** y obtiene la respuesta.

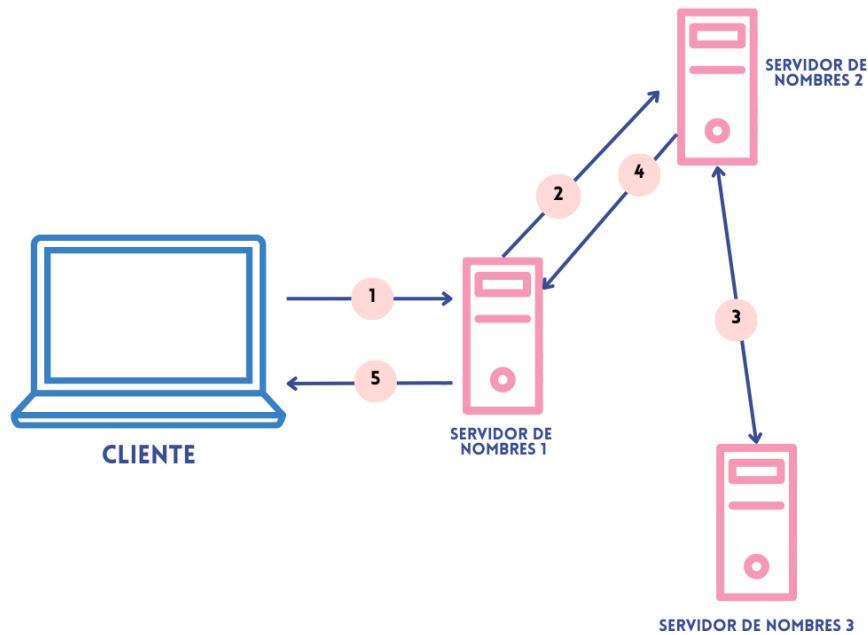


Figure 2.24: DNS: Búsqueda Recursiva desde el servidor.

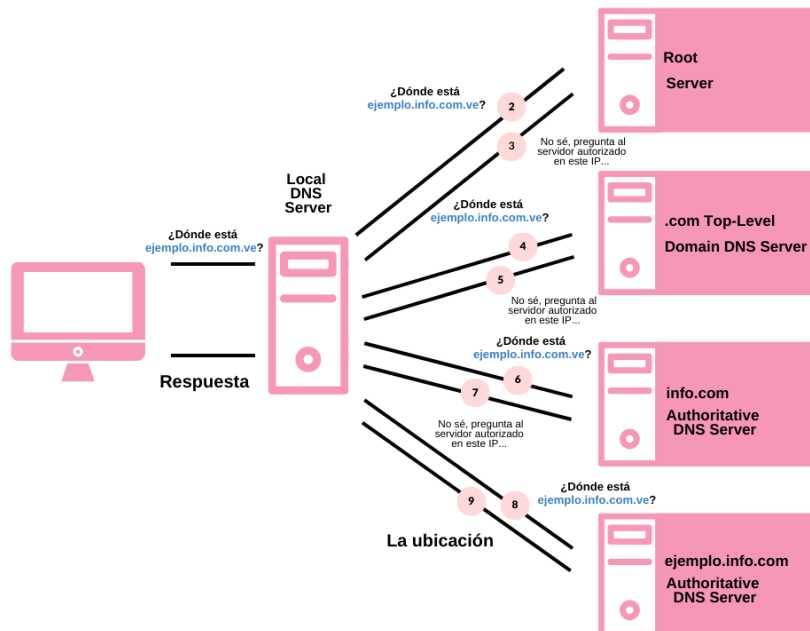


Figure 2.25: DNS: Ejemplo de una búsqueda.

Comunicación entre Procesos

3	Procesos e Hilos	73
3.1	Clientes Multihilos	73
3.2	Servidor Multihilos	76
3.3	Virtualización de sistemas	81
3.4	Caso de Estudio: Virtualización de Redes	84
4	Protocolos de la capa de Transporte	87
4.1	Socket	87
4.2	UDP	88
4.3	TCP	90
4.4	Multidifusión	93
4.5	Caso de Estudio: ¿Como funciona un aplicación de Chat?	94
5	Comunicación entre Procesos Remotos	97
5.1	Protocolo Solicitud-Respuesta	97
5.2	Llamados a Procedimientos Remotos (RPC)	101
5.3	Invocación a Métodos Remotos (RMI)	104
5.4	Caso de Estudio: API	107



3. Procesos e Hilos

Las computadoras son sistemas o dispositivos capaces de ejecutar procesos. Los procesos son programas ejecutándose dentro de su propio espacio de direcciones. Se puede decir que un proceso es un supervisor de hilos de ejecución. Un hilo es una secuencia de código en ejecución dentro del contexto de un proceso. [91]

Un proceso consta de un entorno de ejecución junto con uno o más subprocesos o hilos. Un hilo es la abstracción del sistema operativo de una actividad. Un entorno de ejecución consiste principalmente en [54]:

- un espacio de direcciones;
- sincronización de subprocesos y recursos de comunicación como semáforos e interfaces de comunicación (por ejemplo, sockets);
- recursos de nivel superior como archivos abiertos y ventanas. Los entornos de ejecución normalmente son costosos de crear y administrar, pero varios hilos pueden compartirlos, es decir, pueden compartir todos los recursos accesibles dentro de ellos.

Entre las ventajas que tiene el uso de hilos en procesos centralizados podemos indicar las siguientes. Los hilos comparten el mismo espacio de direcciones. El cambio de contexto de hilos puede ser hecho independiente del sistema operativo. Por su parte, el cambio de proceso es más costoso ya que implica poner el sistema operativo en el bucle, es decir, bloquear el kernel. El uso de hilos permite evitar el cambio de procesos, se estructura las aplicaciones, no como una colección de procesos sino a través de múltiples hilos.

Otra ventaja es que ayuda en la exploración del paralelismo, en un proceso de hilos múltiples, los hilos pueden ser programado para ejecutarse en paralelo en un procesador multiprocesos o multinúcleo.

3.1 Clientes Multihilos

En el contexto de arquitecturas cliente-servidor, la implementación de hilos en el lado del cliente proporciona los siguientes beneficios:

- Ocultar latencias de red: El navegador web escanea una página HTML entrante y encuentra que hay más archivos que necesita, entonces continúa con el proceso de recuperación de los mismos. Los archivos recuperados se despliegan conforme

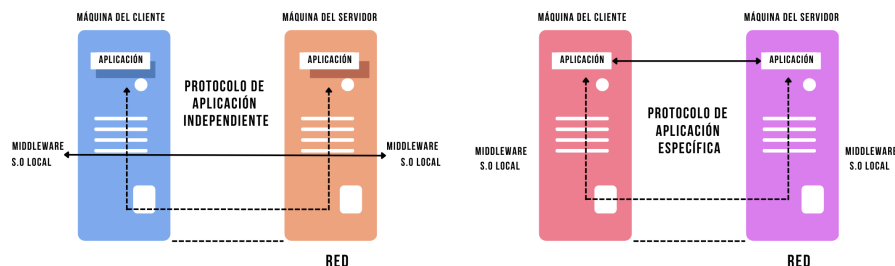


Figure 3.1: (a) Aplicación en red con su protocolo. (b) Aplicación con acceso a aplicaciones remotas. Adaptado de [91]

van llegando. De esta manera, el usuario no necesita esperar hasta que todos los componentes de la página sean recuperados por completo.

- Cada archivo es recuperado por un hilo: La programación de la configuración de una conexión y la lectura desde el servidor pueden llevarse a cabo mediante una solicitud de tipo HTTP(bloqueo). Las llamadas de este tipo no suspende el proceso por completo.
- Conexiones simultáneas: Cuando se utiliza cliente multihilos, las conexiones pueden configurarse como réplicas diferentes lo cual permite que los datos sean transferidos en paralelo asegurando efectivamente que se despliega la página web completa en un tiempo más corto que con un servidor no replicado.

Ejemplo: Múltiples llamadas a RPC

Un cliente realiza varias llamadas al mismo tiempo a procesos remotos, cada una por una diferente hilo. Luego, espera hasta que se hayan devuelto todos los resultados. Si las llamadas son a diferentes servidores, podemos tener una rapidez lineal en la respuesta a la solicitud.

Interfases de Usuario en la Red Las interfases de usuario en la red pueden implementarse a nivel de aplicación y a nivel de middleware.

Este ejemplo, tomado de [91], ilustra la diferencia de la implementación de interfases a nivel del cliente o en el middleware. Aquí se ilustra el comportamiento de una agenda que se ejecuta en una PDA de un usuario y que requiere sincronizarse con una agenda compartida remota. En este caso, un protocolo a nivel de aplicación manipulará esta sincronización, como podemos ver en la Figura 3.1 parte (a). En la parte (b) de la Figura 3.1 se muestra una solución donde se proporciona acceso directo a servicios remotos solamente por medio de la oferta en la interfaz de usuario. Esto significa que la máquina cliente sólo se utiliza como terminal sin necesidad de almacenamiento local. En este caso de interfaces de usuario en red, todo es procesado y almacenado en el servidor. Este método de cliente ligero llamado también clientes delgados, recibe mayor atención al incrementarse la conectividad a internet, y a medida que los dispositivos portátiles (*hand-held*) se han

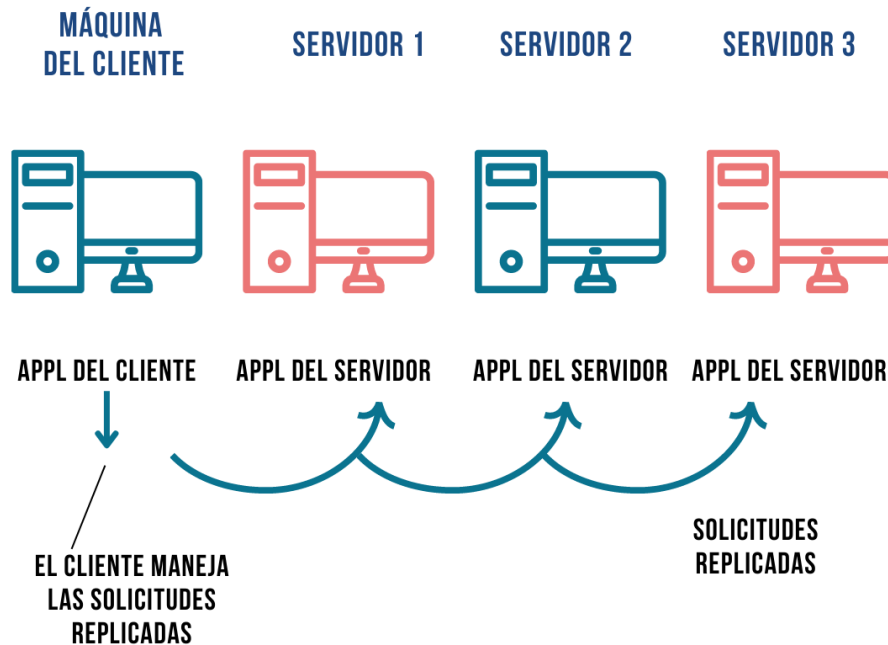


Figure 3.2: Transparencia de replicación de un servidor mediante una solución del lado del cliente. Adaptado de [91]

vuelto más sofisticados.

Software del lado del Cliente para transparencia en la distribución Un cliente no solo consta de una interfaz de usuario y de la aplicación, el software del cliente tiene los componentes necesarios para lograr la transparencia en el acceso, migración, distribución y fallas. A continuación se menciona cada una de este tipo de transparencia [91]:

- Transparencia de acceso: Implementando *stubs* (apéndices o conectores) del lado del cliente para las llamadas a procedimientos remotos (RPC). La transparencia en el acceso es gestionada a partir de una definición de interfaz donde se muestra lo que el cliente tiene que ofrecer.
- Transparencia de ubicación/migración: deje que el software del lado del cliente realice un seguimiento de ubicación actual. Para ello es importante el uso de un adecuado sistema de nombres, ver la sección referida al DNS en 2.
- Transparencia de replicación: múltiples invocaciones manejadas por el código auxiliar del cliente. El software del lado del cliente puede recopilar de manera transparente todas las respuestas y pasar solamente una respuesta a la aplicación del cliente. Esquema de este comportamiento se muestra en la Figura 3.2.
- Transparencia de falla: El enmascaramiento de las fallas de la comunicación con un servidor se hace a través del middleware del cliente: ejemplo, la configuración del

middleware del cliente para intentar repetidamente la conexión a un servidor, o tratar con otro servidor después de varios intentos fallidos; también cuando *middleware* del cliente devuelve datos que tenía en caché durante una sesión previa.

3.2 Servidor Multihilos

El uso de clientes multihilos presenta importantes beneficios para los sistemas distribuidos, pero el uso principal de la tecnología multihilos está del lado del servidor. Veamos sus características.

Ventajas y desventajas El uso de hilos en el lado del servidor proporciona las siguientes ventajas que redundan en la mejora del rendimiento para las aplicaciones distribuidas:

- Iniciar un hilo es más barato que comenzar un nuevo proceso.
- Tener un servidor multihilos evita la ampliación a un sistema con multiprocesadores.
- Un hilo no afecta a otros hilos. En un servidor multihilos, si se produce algún error en cualquiera de los hilos, ningún otro hilo se verá afectado, todos los demás hilos seguirán ejecutándose con normalidad. En un servidor de hilo único, todos los demás clientes tenían que esperar si ocurría algún problema en el hilo.
- Al igual que con los clientes: oculta la latencia de la red reaccionando a la siguiente solicitud mientras la anterior está siendo respondida.
- Rápido y eficiente: el servidor multihilo podría responder de manera eficiente y rápida a las crecientes consultas de los clientes rápidamente.
- El tiempo de espera para los usuarios disminuye: en un servidor hilo único, otros usuarios tenían que esperar hasta que se completara el proceso en ejecución, pero en servidores multihilos, todos los usuarios pueden obtener una respuesta a la vez, por lo que ningún usuario tiene que esperar a que finalicen otros procesos.

Pero las ventajas relevantes descansan en que proporciona aplicaciones mejor estructuradas ya que simplifica el flujo de la información:

- La mayoría de los servidores tienen altas demandas de solicitudes E/S. Con las llamadas con bloqueos, se simplifica la estructura general de la aplicación.
- Los programas multihilos tienden a ser más pequeños y fáciles de entender debido a que se simplifica el flujo de la información.

Las desventajas de esta arquitectura:

- Código complicado: puede resultar difícil escribir el código del servidor multihilo. Estos programas no se crean fácilmente.
- La depuración es difícil: analizar la razón principal y el origen del error no es fácil de seguir.

Organización del servidor multihilo En la Figura 3.3 se ilustra la organización de un servidor multihilos. El servidor espera una petición de entrada para una operación de archivo, posteriormente ejecuta la petición, y luego envía la respuesta de regreso. En la Figura 3.3, un hilo servidor, lee las peticiones de entrada para una operación con archivos. Las peticiones son enviadas por clientes hacia un puerto conocido por este servidor. Después de examinar la petición, el hilo en el servidor elige un hilo trabajador sin utilizar (desbloqueado) y le asigna la petición.

El hilo trabajador procede a realizar una lectura con acción de bloqueo en el sistema de archivos local, ello puede provocar que el hilo se suspenda hasta que los datos sean recuperados desde el disco local. Si el hilo se suspende, se puede seleccionar otros hilo

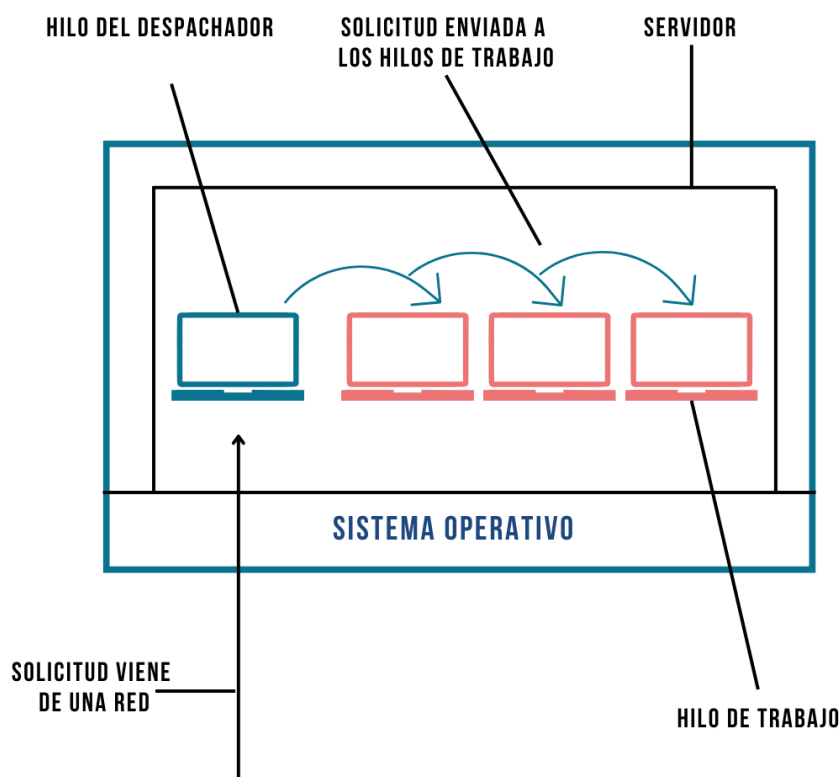


Figure 3.3: Servidor Multihilo.

para su ejecución. Por ejemplo, el hilo servidor puede seleccionar la adquisición de más hilos trabajadores.

Arquitectura para servidores multihilos En la arquitectura de hilos por solicitud (Figura 3.4), el hilo de E/S genera un nuevo hilo de trabajo por cada solicitud, y ese hilo trabajador se destruye a sí mismo cuando ha procesado la solicitud contra el objeto remoto designado. Esta arquitectura tiene la ventaja de que los hilos no compiten por una cola compartida y el rendimiento se maximiza potencialmente ya que el hilo de E/S puede crear tantos hilos trabajadores como solicitudes pendientes. La desventaja es la sobrecarga de las operaciones en la creación y destrucción de hilos.

La arquitectura hilo por conexión representado en la Figura 3.5, asocia un hilo con cada conexión. El servidor crea un nuevo hilo de trabajo cuando un cliente realiza una conexión y destruye el hilo cuando el cliente cierra la conexión. Mientras tanto, el cliente puede realizar muchas solicitudes a través de la conexión, dirigidas a uno o más objetos remotos.

La arquitectura hilo por objeto en la Figura 3.6, asocia un hilo con cada objeto remoto. Un subproceso de E/S recibe solicitudes y las pone en cola para los trabajadores, pero esta vez hay una cola por objeto.

En cada una de estas dos últimas arquitecturas, el servidor se beneficia de una gestión de hilos con bajo costo si se compara con la arquitectura de hilos por solicitud. Su desventaja

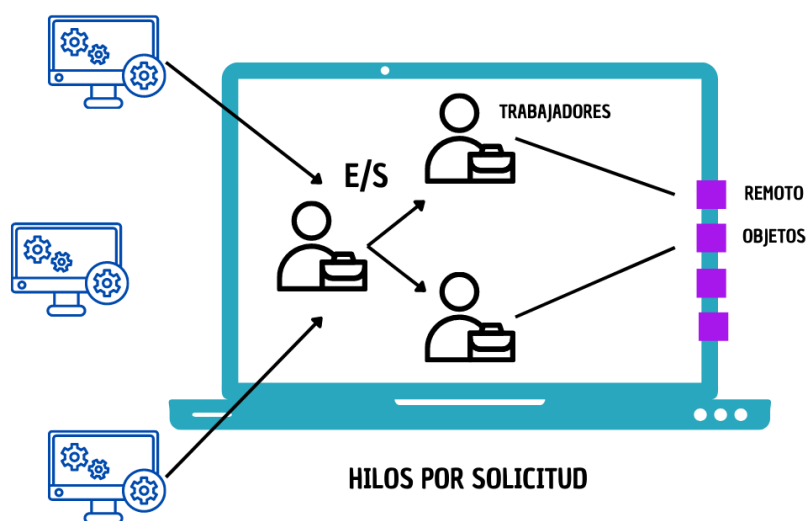


Figure 3.4: Servidores multihilos por solicitud

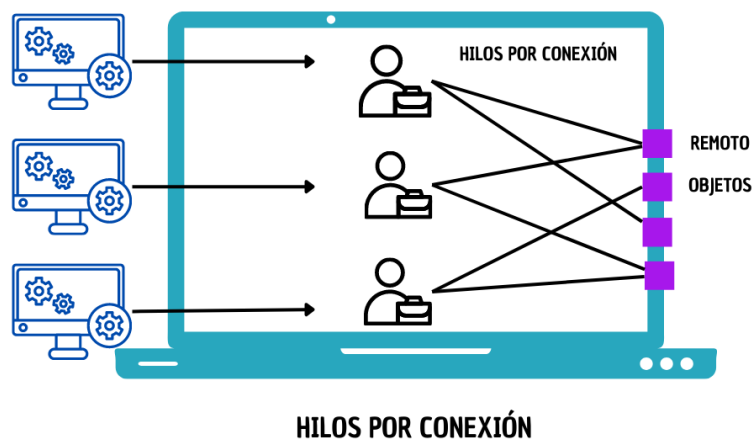


Figure 3.5: Servidores multihilos por conexión

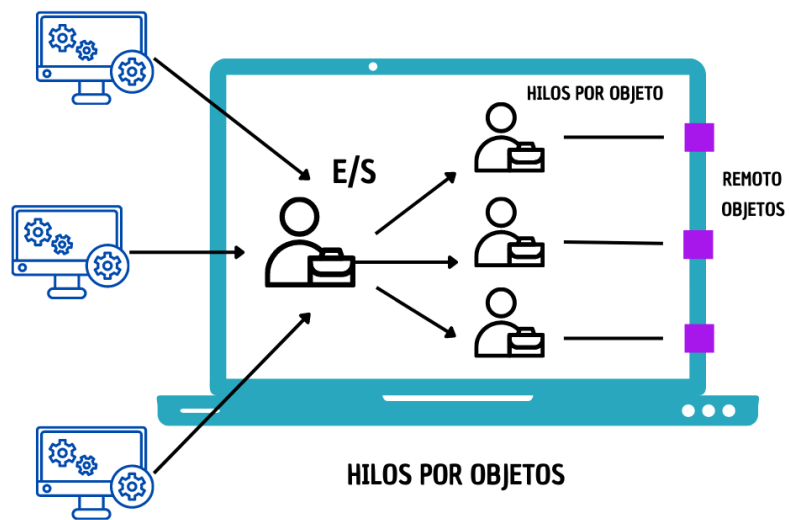


Figure 3.6: Servidores multihilos por objeto

es que los clientes pueden retrasarse mientras un hilo de trabajo tiene varias solicitudes pendientes pero otro hilo no tiene trabajo que realizar.

Ejemplo de servidor multihilo

En esta dirección [Servidor multihilo](#) puede ver y ejecutar un servidor multihilo

3.3 Virtualización de sistemas

La virtualización es una tecnología que se puede usar para crear representaciones virtuales de servidores, almacenamiento, redes y otras máquinas físicas. La virtualización puede ser aplicada en el contexto de la creación de redes superpuesta, que ofrecen soporte para tipos particulares de aplicaciones distribuidas. Otra aplicación es la virtualización de sistemas, que es en el contexto de los sistemas operativos; esta es la que se estudia en esta sección.

El objetivo de la virtualización de sistema [54] es proporcionar múltiples máquinas virtuales (imágenes de hardware virtual) sobre la arquitectura de la máquina física subyacente, con cada virtual máquina que ejecuta una instancia de sistema operativo separada. El concepto surge de la observación de que las arquitecturas informáticas modernas tienen el rendimiento necesario para admitir potencialmente un gran número de máquinas virtuales y recursos multiplexados entre ellos. Varias instancias del mismo sistema operativo pueden ejecutarse en máquinas virtuales o se puede admitir una gama de diferentes sistemas operativos. El sistema de virtualización asigna los procesadores físicos y otros recursos de una máquina física entre todas las máquinas virtuales que admite.

Tipos de Virtualización En La Figura 3.7 se muestra, de acuerdo a [36], cuatro tipos de interfaces en tres niveles diferentes de las capas de implementación en un sistema informático típico: la arquitectura del conjunto de instrucciones (ISA, *instruction set architecture*), la interfaz binaria de la aplicación (ABI, *application binary interface*) y interfaz de programación de aplicaciones (API, *application programming interface*).

- ISA. La ISA marca la división entre hardware y software, y consta de las interfaces: (a) la ISA del usuario e incluye aquellos aspectos visibles para un programa de aplicación (interfaz 4); (b) es un superconjunto del usuario ISA e incluye aquellos aspectos visibles solo para el software del sistema operativo responsable de administrar los recursos de hardware, (interfaz 3).
- ABI. La ABI da acceso a un programa a los recursos y servicios de hardware disponibles en un sistema a través de la ISA de usuario (interfaz 4); y de la interfaz de llamada al sistema (interfaz 2).
- API. La API le da a un programa acceso a los recursos y servicios de hardware disponibles en un sistema a través del usuario ISA (interfaz 4) complementado con llamadas de biblioteca de lenguaje de alto nivel (HLL) (interfaz 1). Cualquier llamada del sistema generalmente se realizan a través de bibliotecas.

La esencia de la virtualización estriba en la capacidad de imitación de las interfaces mencionadas. En la Figura 3.8 se ilustra las formas de virtualización.

- (a) Conjunto separado de instrucciones, un intérprete/emulador, que se ejecuta sobre un sistema operativo. Se puede construir un sistema en tiempo de ejecución que esencialmente proporcione un conjunto de instrucciones abstractas que se utilizará para ejecutar aplicaciones. Las instrucciones se pueden interpretarse (ejemplo, Java) o emularse (emular SO Windows sobre Linux).

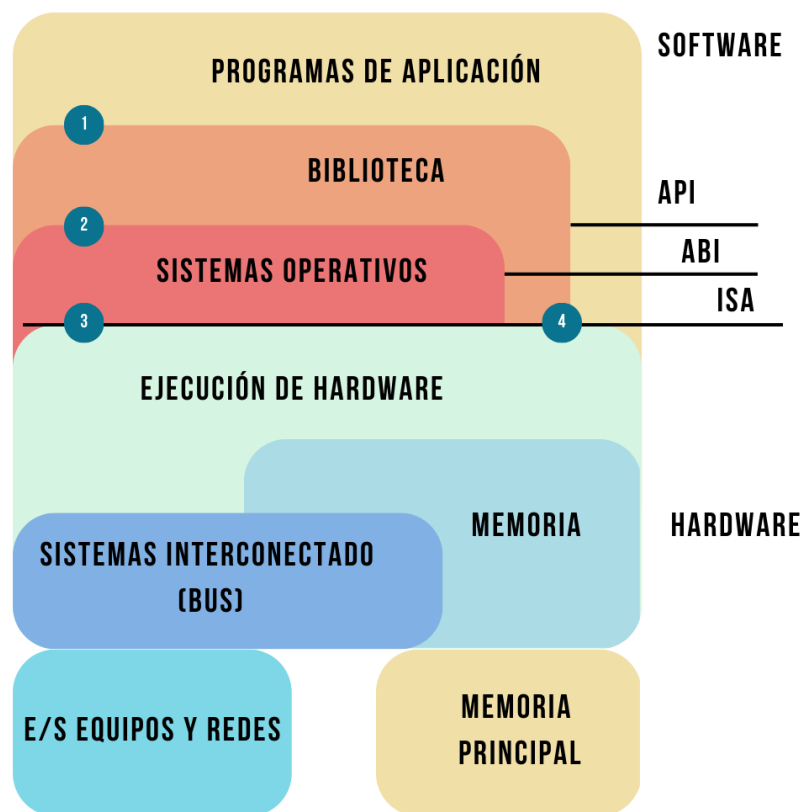


Figure 3.7: Interfases en sistemas informáticos. Adaptado de [36]

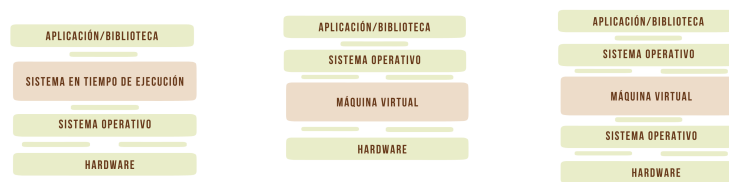


Figure 3.8: Formas de Virtualización: (a) VM de proceso, (b) VMM nativo, (c) VMM alojado. Adaptado de [91]

- (b) Instrucciones de bajo nivel, junto con un sistema operativo mínimo básico. Este enfoque se conoce como un monitor de máquina virtual nativo. Se llama nativo porque se implementa directamente en parte superior del hardware subyacente. Tenga en cuenta que la interfaz que ofrece un monitor de máquina virtual se puede ofrecer simultáneamente a diferentes programas. Tendría que proporcionar y regular el acceso a varios recursos, como almacenamiento externo y redes; esto implica que tendrá que implementar controladores de dispositivo para esos recursos. Como resultado, ahora es posible tener múltiples y diferentes sistemas operativos invitados ejecutándose de forma independiente y simultánea en la misma plataforma.
- (c) Instrucciones de bajo nivel, pero delegando la mayor parte del trabajo a un sistema operativo completo. Esta configuración llamada monitor de máquina virtual alojada se ejecutará sobre un sistema operativo alojador (*host*) de confianza, como se muestra en la Figura 3.8(c). En este caso, el monitor de la máquina virtual puede hacer uso de las instalaciones existentes proporcionadas por ese sistema operativo *host*.

3.3.1 VM y computación en la nube

Tres tipos de servicios en la nube [76]

- **Infraestructura como servicio** que cubre la infraestructura básica
- **Plataforma como servicio** que cubre servicios a nivel de sistema
- **Software como servicio** que contiene aplicaciones reales

Infraestructura como servicio (IaaS.) IaaS es la distribución de una infraestructura informática (recursos informáticos, de redes y de almacenamiento) como un servicio. Permite a los clientes escalar (agregar más recursos) o reducir (liberar recursos) cuando sea necesarios (y solo pagan por los recursos consumidos). Esta capacidad es llamado **elasticidad** y generalmente se logra a través de la virtualización de servidores, una tecnología que permite que varias aplicaciones se ejecuten en el mismo servidor físico que las máquinas virtuales, es decir, como si se ejecutaran en distintos servidores físicos. Luego, los clientes pueden solicitar instancias informáticas como máquinas virtuales y agregar y adjuntar almacenamiento según sea necesario. Un ejemplo de IaaS son los servicios web de **Amazon**.

Software como servicio (SaaS.) SaaS es la entrega de software de aplicación como un servicio. Generaliza el modelo anterior de proveedor de servicios de aplicaciones (ASP) mediante el cual la aplicación alojada es propiedad exclusiva, operada y mantenida por el ASP. Con SaaS, el proveedor de la nube permite al cliente utilizar aplicaciones alojadas (como con ASP) pero también proporciona herramientas para integrar otras aplicaciones,

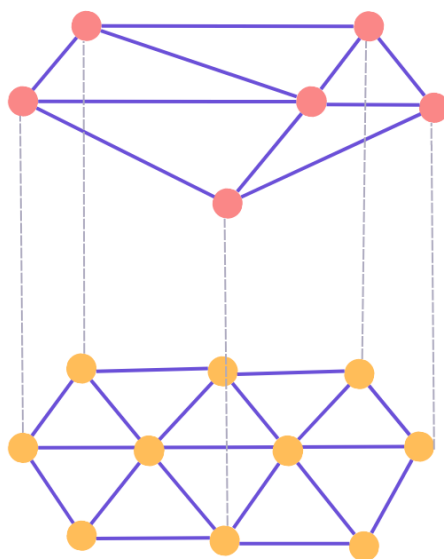


Figure 3.9: Red Superpuesta.

de diferentes proveedores o incluso desarrollado por el cliente (usando la plataforma en la nube). Ejemplo de una SaaS es el sistema **Salesforce CRM**.

Plataforma como servicio (PaaS.) PaaS es la entrega de una plataforma informática con herramientas de desarrollo y APIs como servicio. Permite a los desarrolladores crear e implementar aplicaciones personalizadas directamente en la infraestructura de la nube e integrarlas con aplicaciones proporcionadas como SaaS. Un ejemplo de PaaS es **Google Apps**.

3.4 Caso de Estudio: Virtualización de Redes

La Virtualización de redes ocupa de la construcción de redes virtuales diferentes a través de una red base existente como la red de Internet. Cada red virtual se puede diseñar para admitir una aplicación distribuida en particular [54].

Por ejemplo, una red virtual podría admitir la transmisión de multimedia, como en **Youtube**, **Netflix** y convivir con otras redes que admita un juego con múltiples jugadores en línea; todos estos ejemplos se ejecutan en la misma red subyacente. Una red virtual específica para una aplicación se puede construir sobre la red existente y se optimiza para esa aplicación en particular, sin cambiar las características de la red subyacente.

Las redes superpuestas son redes virtuales ([78]) que consta de nodos y enlaces virtuales, que se encuentra encima de una red subyacente (como una red IP), en la Figura 3.9 se muestra un esquema de una red superpuesta.

Las redes superpuestas ofrecen:

- un servicio que se adapta a las necesidades de una clase de aplicación o un servicio particular nivel superior, por ejemplo, las redes sociales;
- una operación más eficiente en un entorno de red determinado, por ejemplo, enrutamiento en una red específica;

- una función adicional, por ejemplo, multidifusión o comunicación segura.

Ejemplo de redes superpuesta se encuentra: Skype, redes sociales, redes inalámbricas y redes tolerantes a las interrupciones en el contexto de los dispositivos móviles y computación ubicua, y el soporte de superposición para transmisión multimedia.

3.4.1 Skype: un ejemplo de una red superpuesta

Skype es una aplicación P2P que ofrece voz sobre IP (*VoIP*) [2]. También incluye mensajería instantánea, videoconferencia e interfaces para el servicio de telefonía estándar a través de *SkypeIn* y *SkypeOut*.

Skype

El software fue desarrollado por Kazaa en 2003 y por lo tanto, comparte muchas de las características del intercambio de archivos entre pares de la aplicación Kazaa [23]

Skype es una red virtual en el sentido de que establece conexiones entre personas (suscriptores de Skype activos). No se requiere ninguna dirección IP o puerto para establecer una llamada.

Arquitectura de Skype Skype se basa en una infraestructura P2P que consta de máquinas de usuarios comunes (*hosts*) y supernodos: los supernodos son *hosts* de Skype ordinarios que tienen capacidades suficientes para llevar a cabo su función. Los supernodos se seleccionan a pedido, en función de criterios que incluyen ancho de banda disponible, accesibilidad (la máquina debe tener una dirección IP global y no oculto detrás de un enrutador habilitado para NAT, por ejemplo) y disponibilidad (según el tiempo que Skype se ha estado ejecutando en ese nodo). Esta estructura general se captura en la Figura 3.10 .

Conexión de usuario Los usuarios de Skype se autentican a través de un conocido servidor de inicio de sesión. Luego contacta con un supernodo seleccionado. Para lograrlo, cada cliente mantiene un caché de identidades de supernodo (es decir, pares de direcciones IP y números de puerto). En el primer inicio de sesión este caché se llena con las direcciones de alrededor de siete supernodos, y con el tiempo el cliente crea y mantiene un conjunto mucho más grande (quizás varios cientos).

Búsqueda de usuarios El objetivo principal de los supernodos es realizar la búsqueda eficiente de los índices globales de usuarios, que se distribuye entre los supernodos. La búsqueda es orquestada por el supernodo elegido por el cliente e implica una búsqueda en expansión de otros super nodos hasta que se encuentre el usuario especificado. En promedio, se contactan ocho supernodos. La búsqueda de un usuario suele tardar entre tres y cuatro segundos en completarse en los *hosts* que tienen una dirección IP global (y un poco más larga, de cinco a seis segundos, si están detrás de un enrutador habilitado para NAT). De acuerdo a los experimentos de [2], parece que los nodos intermediarios involucrados en la búsqueda, almacenan en caché los resultados para mejorar el rendimiento.

Conexión de voz Una vez que se encuentra al usuario requerido, Skype establece una conexión de voz entre las dos partes utilizando TCP para señalar solicitudes de llamada, y UDP o TCP para la transmisión de audio. Se prefiere UDP en vez de TCP. El software utilizado para codificar y decodificar audio juega un papel clave para proporcionar una excelente calidad de llamada que normalmente se logra con

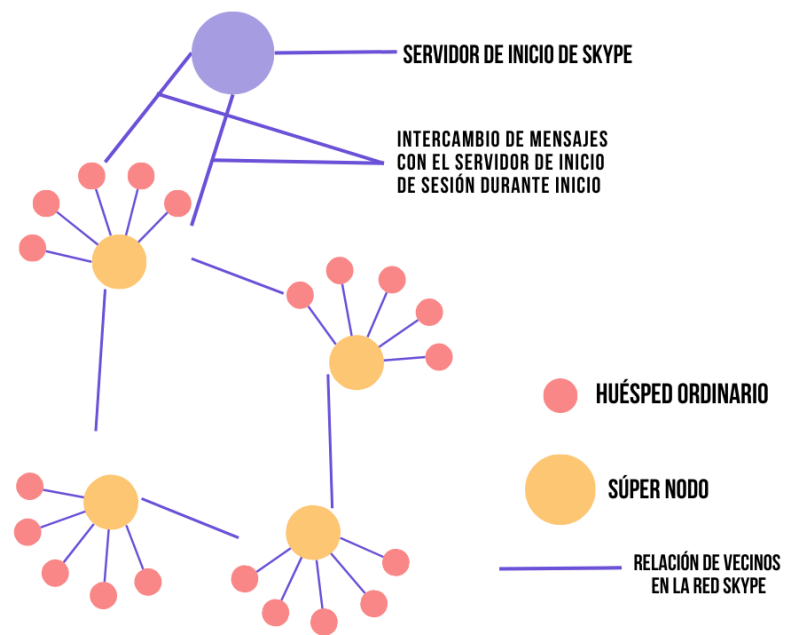


Figure 3.10: Arquitectura Skype.

Skype, y los algoritmos asociados se adaptan cuidadosamente a operar en entornos de Internet a 32 kbps y superiores.



4. Protocolos de la capa de Transporte

Este capítulo trata acerca de las interfaces de los programas de aplicación (API), en particular de las interfaces entre los programas de aplicación y el protocolo TCP/IP. En arquitecturas cliente-servidor, cuando se hace una solicitud, por ejemplo obtener una imagen, `image.png` de la página web `example.com`, los pasos que se siguen, a grandes rasgos, son los siguientes:

1. Obtener una dirección ip para `example.com` (descrito en la sección acerca del DNS, ver 2.4).
2. Abrir un *socket* a una dirección ip en un puerto determinado
3. Abrir una conexión TCP a una dirección ip en un puerto determinado
4. Hacer la solicitud HTTP GET, de la `image.png`
5. Obtener la imagen solicitada, `image.png`
6. Cerrar la conexión.

En lo que sigue, se detalla el concepto de *socket*.

4.1 Socket

La abstracción de la interfaz *socket* es el socket [53], [78]. Un socket es como el punto donde un proceso de aplicación local se conecta a la red. La interfaz define las operaciones de creación del socket, adjuntar el socket a la red, enviar/recibir mensajes a través del socket y cerrar el socket. como se ilustra en la Figura 4.1.

Para que un proceso reciba mensajes, su socket debe estar vinculado a un puerto local y a una de las direcciones de Internet en la computadora que se ejecuta. Los mensajes enviados a una dirección de Internet y número de puerto en particular solo pueden recibir un proceso cuyo socket está asociado con esa dirección de Internet y número de puerto.

Los procesos pueden usar el mismo socket para enviar y recibir mensajes. Cualquier proceso puede hacer uso de múltiples puertos para recibir mensajes, pero un proceso no puede compartir puertos con otros procesos en la misma computadora. Cada socket esta asociado con un protocolo particular, ya sea *UDP* o *TCP*.

Veamos como se implementan los socket en los protocolos de la capa de transporte, UDP y TCP.

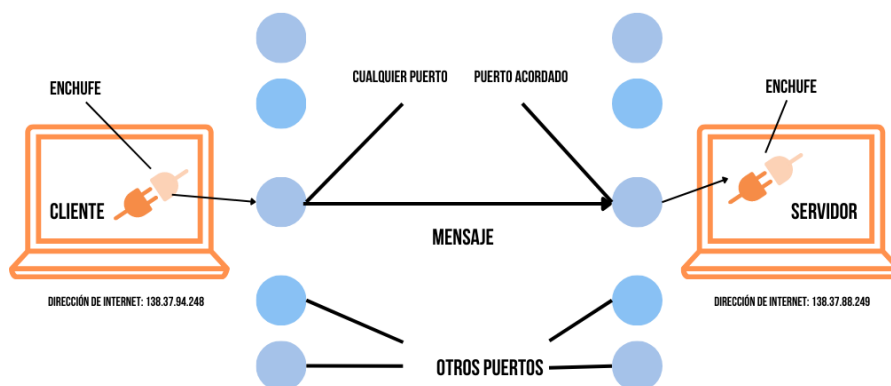


Figure 4.1: Socket.

4.2 UDP

UDP, User Datagram Protocol, es un protocolo de nivel de transporte basado en el intercambio de datagramas. Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera. No tiene confirmación de recepción del mensaje ni control de flujo de transmisión del mensaje, por lo que los paquetes pueden adelantarse unos a otros; y tampoco se sabe si ha llegado correctamente, ya que no hay confirmación de entrega o recepción.

Un datagrama enviado por UDP se transmite desde un proceso de envío a un proceso de recepción sin reconocimiento ni reintentos. Si ocurre una falla, el mensaje puede que no llegue. Para enviar o recibir mensajes, un proceso primero debe crear un socket vinculado a una dirección de Internet del *host* local y un puerto local. El servidor enlazará su socket a su puerto: y se da a conocer a los clientes para que puedan enviarle mensajes.

Datagrama

En la capa de transporte, la unidad de transferencia básica en las llamadas de internet son los datagramas de internet, llamados también datagramas IP o datagrama. La capa de transporte es responsable de brindar servicios a la capa de aplicación: obtener un mensaje de un programa de aplicación que se ejecuta en el host de origen; lo encapsula en un paquete de capa de transporte (llamado datagrama de usuario); y entregarlo al programa de aplicación correspondiente en el host de destino. Un datagrama está compuesto de dos partes: encabezado y cuerpo, en la Figura 4.2 se ilustra la estructura.

Revisar más de *UDP* en documento de *IBM*: [UDP](#)



Figure 4.2: Estructura del Datagrama

Los siguientes son algunas características relacionadas con la comunicación de datagramas [67]:

Tamaño del mensaje: la longitud total de un datagrama está medida en octetos.

- Longitud del encabezado (HLEN). El campo de longitud del encabezado de 4 bits (HLEN) define la longitud total del encabezado del datagrama en palabras de 4 bytes. El datagrama IPv4 tiene un encabezado de longitud variable. Cuando un dispositivo recibe un datagrama, necesita saber cuándo se detiene el encabezado y comienzan los datos, que están encapsulados en el paquete. Para que el valor de la longitud del encabezado (número de bytes) se ajuste a una longitud de encabezado de 4 bits, la longitud total del encabezado se calcula como palabras de 4 bytes. La longitud total se divide por 4 y el valor se inserta en el campo.
- Longitud total. Es un campo de 16 bits (encabezado más datos) del datagrama IP en bytes. Un número de 16 bits puede definir una longitud total de hasta 65.535. Sin embargo, el tamaño del datagrama es normalmente mucho menor que esto. Para encontrar la longitud de los datos de un datagrama, reste la longitud del encabezado de la longitud total. La longitud del encabezado se puede encontrar multiplicando el valor en el campo HLEN por 4:

$$\text{Longitud de datos} = \text{longitud total} - (\text{HLEN}) \times 4$$

Bloqueos: La implementación de sockets con datagramas normalmente proporcionan envíos sin bloqueo y recepciones con bloqueo (una recepción sin bloqueo es una opción en algunas implementaciones):

- La operación de envío termina cuando ha entregado el mensaje a los protocolos

UDP e IP subyacentes, que son responsables de transmitirlo al host de destino.

- A su llegada al destino, el mensaje se coloca en una cola asociada al puerto de destino que esta relacionado con el socket.
- El mensaje puede ser recogido de la cola por un invocación pendiente o futura a ese socket.
- Los mensajes se descartan en el destino si no hay procesos con un socket vinculado al puerto de destino.

Tiempos de espera: La recepción con bloqueo es adecuada cuando un servidor está esperando recibir solicitudes de sus clientes. Pero en algunos programas, no es apropiado que en procesos que han invocado una operación de recepción con espera, lo haga indefinidamente. Puede suceder fallas en el proceso de envío o el mensaje esperado puede haberse perdido. Para ello, se pueden establecer tiempos de espera en los sockets.

Recibir de cualquiera: el método de recepción no especifica un origen para los mensajes. En cambio, una invocación del método de recepción obtiene un mensaje dirigido a su socket desde cualquier origen. El método de recepción devuelve la dirección de Internet y el puerto local del remitente, permitiendo al destinatario comprobar de dónde procede el mensaje. Es posible conectar un *socket* de datagrama a un puerto remoto particular y una dirección de Internet, para que el *socket* pueda enviar mensajes y recibir mensajes de esa dirección.

4.2.1 Uso de UDP

Voz sobre IP y DNS: Es una opción popular porque no causa sobrecarga (*overhead*) en la red

4.3 TCP

A diferencia del protocolo *UDP*, *TCP* es un protocolo que ofrece un servicio de flujos de bytes, fiable y orientado a la conexión. Dicho servicio ha demostrado ser útil para una amplia variedad de aplicaciones porque libera a la aplicación de tener que preocuparse sobre datos perdidos o reordenados, [53].

El protocolo de control de transmisión (*Transmission Control Protocol* o *TCP*) es uno de los protocolos fundamentales en Internet. Los programas pueden usar *TCP* para crear “conexiones” entre sí a través de las cuales puede enviarse un flujo de datos.

TCP da soporte a muchas de las aplicaciones de Internet (navegadores, intercambio de ficheros, clientes *FTP*, etc.) y protocolos de aplicación, [78], [67] :

HTTP *HTTP*, protocolo de transferencia de hipertexto, se utiliza para la comunicación entre los navegadores web y servidores web.

FTP el Protocolo de transferencia de archivos permite que los directorios en una computadora remota naveguen y transferir archivos para ir de una computadora a otra a través de una conexión.

Telnet *Telnet* proporciona acceso mediante una sesión de terminal a una computadora remota.

SMTP el Protocolo de transferencia de correo se utiliza para enviar correo entre computadoras.

En la Figura 4.3 se muestra la posición de IP y de los protocolos de la capa de red, en conjunto con la grupo de protocolos de *TCP/IP* ya descritos (se han omitido los protocolos

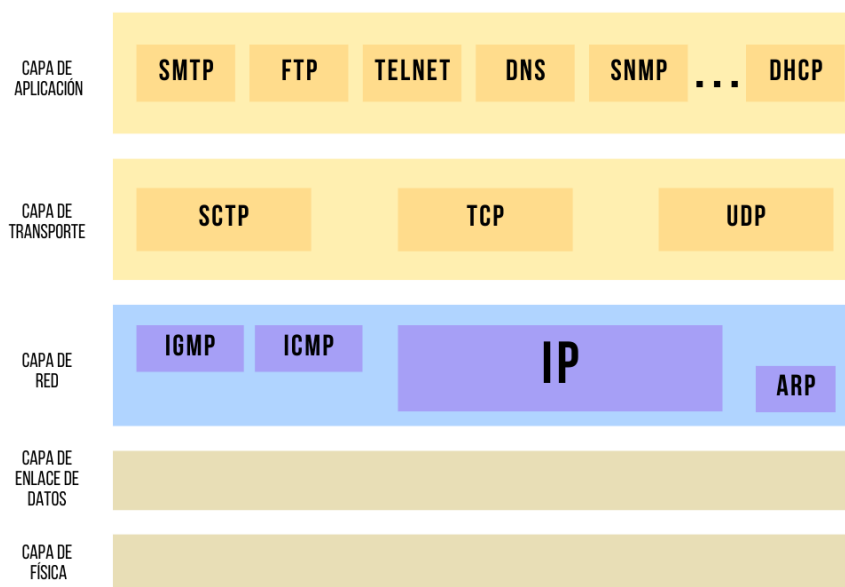


Figure 4.3: Protocolos TCP/IP

por debajo de la capa de Red).

Las características de TCP son las siguientes:

- Tamaños de mensaje:** la aplicación puede elegir cuántos datos escribe o lee en un segmento. Puede tratar conjuntos de datos muy pequeños o muy grandes. La implementación de una secuencia o segmento TCP decide cuántos datos recopilar antes de transmitirlo como uno o más paquetes IP. A su llegada, los datos se entregan a la aplicación según lo solicitado.
- Mensajes perdidos:** el protocolo TCP utiliza un esquema de reconocimiento del mensaje. Por ejemplo, el extremo emisor mantiene un registro de cada paquete IP enviado y el extremo receptor reconoce todas las solicitudes recibidas y envía un acuse de recibo. Si el remitente no recibe un acuse de recibo de un mensaje dentro de un tiempo de espera, entonces retransmite el mensaje.
- Control de flujo:** TCP intenta igualar las velocidades de los procesos de lectura y escritura en un segmento. Si el escritor es demasiado rápido para el lector, entonces es bloqueado hasta que el lector haya consumido suficientes segmentos de datos.
- Duplicación y ordenamiento de mensajes:** los identificadores de mensajes están asociados con cada paquete IP, que permite al destinatario detectar y rechazar duplicados, o reordenar los mensajes que no llegan en el orden del remitente.
- Destinos de mensajes:** un par de procesos de comunicación establecen una conexión antes de que puedan comunicarse a través de una secuencia. Una vez que se establece una conexión, Los procesos leen y escriben los *stream* de datos sin necesidad de utilizar direcciones de Internet o puertos. Establecer una conexión implica una solicitud de conexión del cliente al servidor seguido de una solicitud de aceptación del servidor al cliente antes que cualquier comunicación puede tener lugar.

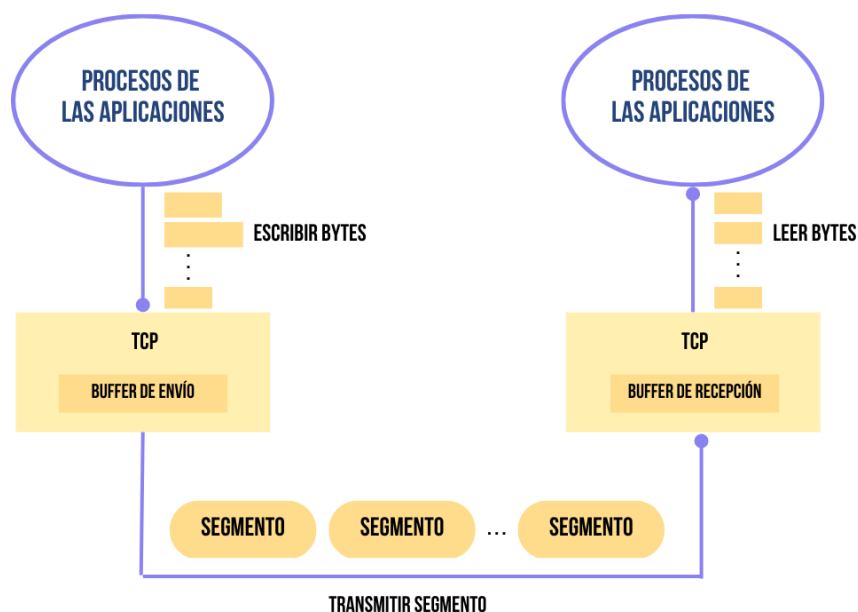


Figure 4.4: Intercambio de mensajes en TCP

4.3.1 Mensajes de TCP

TCP es un protocolo orientado a bytes, significa que el remitente escribe bytes en una conexión TCP y el receptor lee bytes de la conexión TCP, [78], [54]. TCP no transmite bytes individuales a través de Internet. En su lugar, TCP en el host origen almacena suficientes bytes del proceso de envío para llenar un paquete de tamaño razonable y luego envía este paquete a su par en el host destino. TCP en el host destino vacía el contenido del paquete en un búfer de recepción, y el proceso de recepción lee de este búfer en su tiempo libre.

En la Figura 4.4, muestra este intercambio de datos en una dirección; aunque, una única conexión TCP admite flujos de bytes que fluyen en ambas direcciones.

Los paquetes intercambiados entre pares TCP en la Figura 4.4 son segmentos, ya que cada uno lleva una parte del flujo de bytes.

La estructura del segmento se muestra en la Figura 4.5.

- **SrcPort** y **DstPort** identifican el origen y el destino de los puertos, al igual que en UDP. Estos campos, más la dirección de la fuente y las IP de destino, identifican cada conexión TCP.
- El campo **número de secuencia** contiene el número de secuencia para el primer byte de datos transportado en ese segmento.
- Los campos **Reconocimiento** y **AdvertisedWindow** llevan información sobre el flujo de datos que van en la dirección contraria.
- El campo **banderas** se utiliza para transmitir información de control entre los puntos de conexión TCP. Los indicadores posibles incluyen SYN, FIN, RESET, PUSH, URG y ACK. Por ejemplo, las banderas **SYN** y **FIN** se utilizan al establecer y terminar una conexión TCP, respectivamente.
- El campo **HdrLen** da la longitud del encabezado en palabras de 32 bits. Este campo

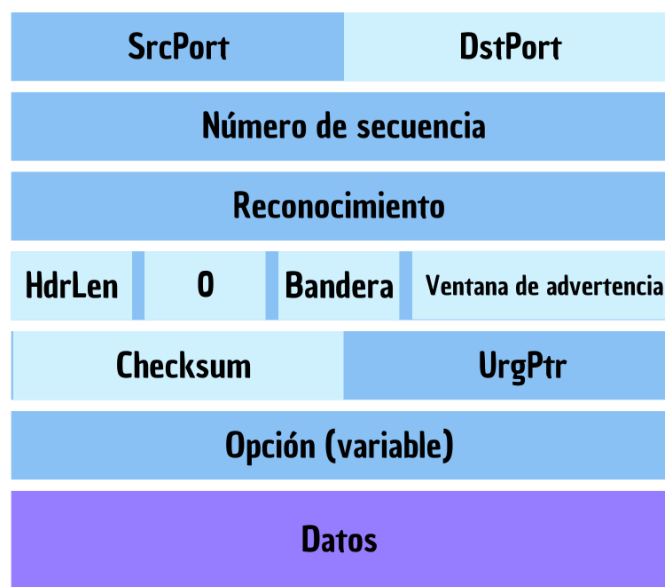


Figure 4.5: Estructura de mensajes en TCP

se conoce como campo de desplazamiento, ya que mide el desplazamiento desde el inicio del paquete hasta el comienzo de los datos.

- El campo **Checksum** se usa de la misma manera que para UDP: se calcula sobre el encabezado TCP, los datos TCP y el pseudoencabezado, que se compone de la dirección de origen, la dirección de destino y la longitud campos del encabezado IP. La suma de comprobación es necesaria para TCP tanto en IPv4 e IPv6.

4.4 Multidifusión

Una operación de multidifusión (*multicast*) es una operación que envía un mensaje de un proceso a cada uno de los miembros de un grupo de procesos, de manera que la membresía del grupo es transparente para el remitente. Hay un abanico de posibilidades en el comportamiento de una multidifusión. El protocolo de multidifusión más simple no ofrece garantías sobre envío de mensajes o mensajes perdidos.

Los mensajes de multidifusión proporcionan una infraestructura para construir sistemas distribuidos con las siguientes características:

1. Tolerancia a fallas basada en servicios replicados: un servicio replicado consta de un grupo de servidores para suplir un servicio en particular. Las solicitudes de los clientes se envían por multidifusión a todos los miembros del grupo de servidores, cada uno de los cuales realiza una operación idéntica. Cuando algunos de los miembros falla, los clientes aún pueden ser atendidos.
2. Descubrimiento de servicios en redes espontáneas: En el descubrimiento de servicios, se usan los mensajes de multidifusión por servidores y clientes para localizar los servicios de disponibles con el fin de registrar sus interfaces o buscar las interfaces de otros servicios en el Sistema distribuido.
3. Mejor rendimiento a través de datos replicados: los datos se replican para aumentar

el rendimiento de un servicio: en algunos casos, las réplicas de los datos se colocan en los computadores de usuarios. Cada vez que cambian los datos, el nuevo valor se envía por multidifusión a los procesos que gestionan las réplicas.

4. Propagación de notificaciones de eventos: la multidifusión a un grupo se puede utilizar para notificar a procesos cuando algo sucede. Por ejemplo, en Facebook, cuando alguien cambia su estado, todos sus amigos reciben notificaciones.

4.4.1 La multidifusión IP

La multidifusión IP se basa en el Protocolo de Internet (IP). Multidifusión IP permite al remitente transmitir un solo paquete IP a un conjunto de computadoras que forman un grupo de multidifusión. La multidifusión IP tiene las siguientes características:

Dirección de Grupos. Cada grupo de multidifusión es una dirección única de la clase D. Unas pocas direcciones son asignadas por la autoridad única de internet, otras direcciones son de uso privado.

Número de Grupos. IP proporciona direcciones hasta para 2^{28} grupos de multidifusión.

Membresía dinámica a grupos. Un host puede unirse o abandonar un grupo de IP multidifusión cuando lo requiera.

Uso del hardware. Si el hardware de la red subyacente soporta IP multidifusión, entonces IP usa el hardware de multidifusión para enviar los mensajes IP multidifusión. En caso contrario, usa difusión (cast) o transmisión (broadcast)

Reenvío en Redes. Debido a que miembros de grupos de multidifusión IP son adjuntos a varias redes físicas, se requiere el uso de enrutadores con la capacidad de multidifusión para el reenvío de multidifusión IP.

Semántica de la distribución Multidifusión IP usa la semántica mejor-esfuerzo para la distribución de mensajes: significa que un datagrama multidifusión puede perderse, retrasarse, duplicarse, o distribuirse fuera de orden

Membresía y transmisión. Un host arbitrario puede enviar datagramas al grupo de multidifusión. La membresía al grupo solo se usa para determinar si el datagrama recibido por el host puede ser enviado al grupo.

En el nivel de programación de aplicaciones, la multidifusión IP solo está disponible a través de UDP. Un programa de aplicación realiza multidifusión enviando datagramas UDP con multidifusión direcciones y números de puerto ordinarios. Puede unirse a un grupo de multidifusión haciendo que su socket se una al grupo, lo que le permite recibir mensajes al grupo. A nivel de IP, una computadora pertenece a un grupo de multidifusión cuando uno o más de sus procesos tiene sockets que pertenecen a ese grupo. Cuando llega un mensaje de multidifusión a una computadora, las copias se reenvían a todos los sockets locales que se han unido a la dirección de multidifusión especificada y están vinculados a el número de puerto especificado.

4.5 Caso de Estudio: ¿Como funciona un aplicación de Chat?

Este ejemplo se muestra como se usan los protocolos de comunicación en una aplicación Chat. Un chat es un tipo de comunicación en tiempo real que se realiza entre varios usuarios cuyas computadoras están conectadas a una red, generalmente Internet; los usuarios escriben mensajes en su teclado, y el texto aparece automáticamente y al instante

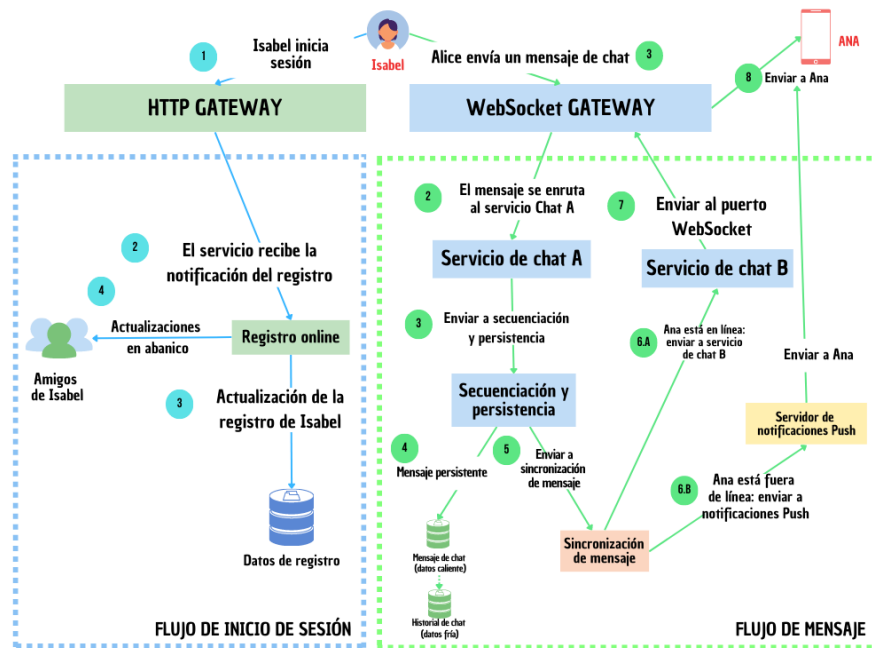


Figure 4.6: Chat. Adaptado de [42]

en el monitor de todos los participantes. Ejemplo de aplicaciones chat son Whatsapp, Google Chat, Skype, entre otras.

A continuación, se describe el funcionamiento, de una aplicación chat, de acuerdo a la Figura ??:

Flujo de inicio de sesión de usuario

- 1: Isabel inicia sesión en la aplicación de chat y establece una conexión de socket web con el lado del servidor.
- 2-4: El servicio recibe la notificación del registro de Isabel, actualiza su registro y notifica a los amigos de Isabel sobre su presencia.

Flujo de mensajería

- 1-2: Isabel envía un mensaje de chat a Ana. El mensaje de chat se dirige al servicio de chat A.
- 3 a 4: el mensaje de chat se envía al servicio de secuenciación, que genera una identificación única y se conserva en el almacén de mensajes.
- 5: El mensaje de chat se envía a la cola de sincronización de mensajes para sincronizar con el servicio de chat de Ana.
- 6: antes de reenviar el mensaje, el servicio de sincronización de mensajes verifica la presencia de Ana:
 1. Si Ana está en línea, el mensaje de chat se envía al servicio de chat B.
 2. Si Ana está desconectada, el mensaje se envía al servidor push y se envía al dispositivo de Ana.
- 7-8 si Ana está en línea, el mensaje de chat se envía a Ana a través del socket web.



5. Comunicación entre Procesos Remotos

Este capítulo continua el estudio de los protocolos para la comunicación entre procesos. Se detallan el Protocolo Solicitud Respuesta, Llamadas a Procedimientos Remotos e Invocación a Métodos Remotos.

5.1 Protocolo Solicitud-Respuesta

El protocolo solicitud-respuesta está diseñada para apoyar los roles e intercambios de mensajes en interacciones cliente-servidor. El protocolo solicitud-respuesta que se describe en la Figura 5.1, [91] [54], se basa en un trío de primitivas de comunicación: *doOperation*, *getRequest* y *sendReply*:

- *doOperation*: los clientes invocan operaciones remotas. Sus argumentos especifican el servidor remoto y la operación a invocar, junto con información adicional (argumentos) requerida por la operación.
- *getRequest* es utilizado por un proceso de servidor para adquirir solicitudes de servicio. Envía un mensaje de solicitud al servidor remoto y devuelve la respuesta.
- *sendReply* Cuando el servidor ha invocado la operación especificada, usa *sendReply* para enviar el mensaje de respuesta al cliente. Cuando el cliente recibe el mensaje de respuesta *doOperation* original se desbloquea y la ejecución del programa cliente continúa.

Ejemplo del protocolo solicitud-respuesta es **HTTP** y **TCP stream** ([93], [54]).

5.1.1 Estructura del Mensaje

La información a transmitir en un mensaje del protocolo solicitud-respuesta, se muestra en la Figura 5.2

- El primer campo indica si el mensaje es una Solicitud o un mensaje de Respuesta
- El campo *requestId*, contiene un identificador de mensaje generado por una operación en el cliente por cada mensaje de solicitud; el servidor copia estos *ID* en los mensajes de respuesta correspondientes. Esto permite que *doOperation* verifique que la respuesta es el resultado de la solicitud actual.
- El tercer campo es una referencia remota, es el identificador del objeto remoto.
- El cuarto campo es el conjunto de argumentos que definen las operaciones que se

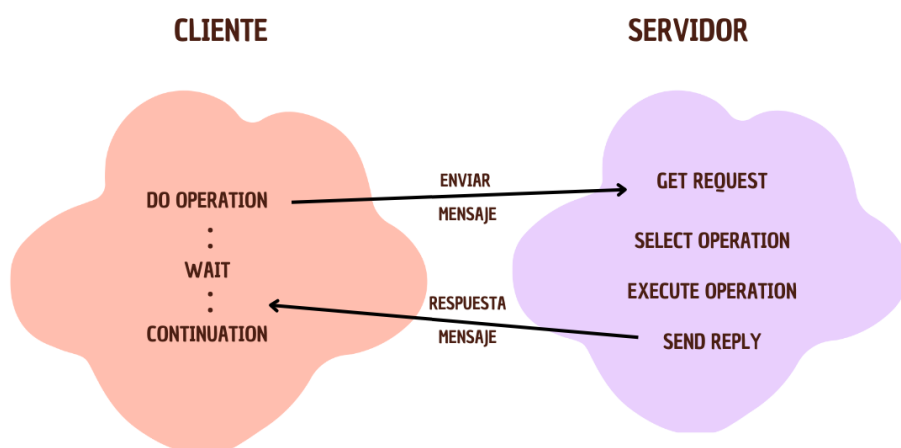


Figure 5.1: Protocolo Solicitud Respuesta.

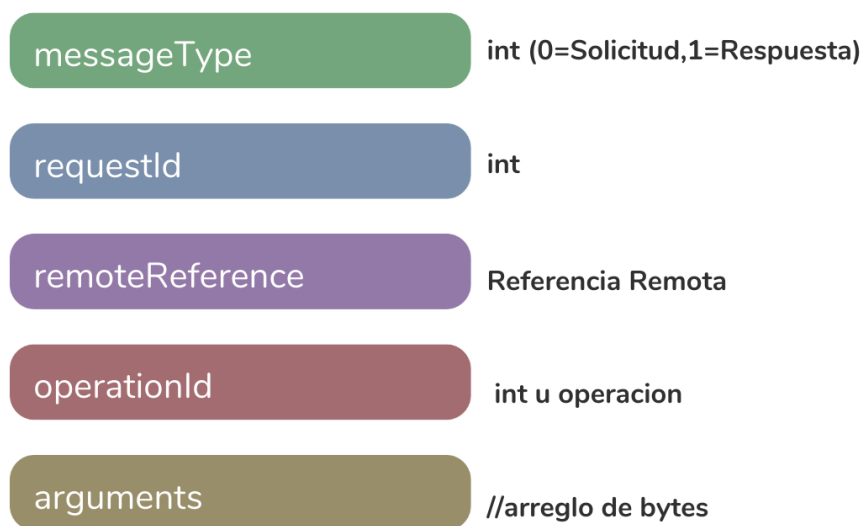


Figure 5.2: Estructura del mensaje.

van a invocar.

5.1.2 Fallas

Si el protocolo solicitud-respuesta se implementan sobre datagramas *UDP*, entonces sufren de los mismos fallos de comunicación que *UDP*. Es decir:

- Fallas por omisión del mensaje.
- No se garantiza que los mensajes se entreguen en el orden del remitente.

5.1.3 Características del Protocolo solicitud-respuesta

- **Tiempos de Espera.** Para las ocasiones en que un servidor falla o que se descarta una solicitud o un mensaje de respuesta, la *doOperation* usa un *timeout* mientras está esperando obtener mensaje de respuesta del servidor. Para compensar la posibilidad de pérdida de mensajes, *doOperation* puede reenviar el mensaje de solicitud repetidamente hasta que recibe una respuesta.
- **Mensajes Duplicados.** En los casos en que el mensaje de solicitud sea retransmitido, el servidor puede recibirlo más de una vez. Esto puede llevar a que el servidor ejecute una operación más de una vez para la misma solicitud. Para evitar esto, el protocolo está diseñado para reconocer mensajes sucesivos (del mismo cliente) con el mismo identificador de solicitud, para filtrar los duplicados.
- **Mensajes Perdidos.** Si el servidor ya ha enviado la respuesta cuando recibe una solicitud duplicada, deberá ejecutar la operación nuevamente para obtener el resultado, a menos que haya almacenado el resultado de la ejecución original.
- **Historial.** Para los servidores que requieren una retransmisión de respuestas sin repetir la ejecución de las operaciones, se puede usar un historial de los mensajes transmitidos. Una entrada en un historial contiene un identificador de solicitud, un mensaje y un identificador del cliente al que se envió la respuesta. Su propósito es permitir que el servidor retransmita los mensajes de respuesta cuando los procesos del cliente los soliciten.

5.1.4 Estilos del protocolo

Se pueden describir tres estilos de protocolos relacionados al protocolo Solicitud-Respuesta, que producen diferentes comportamientos:

- El protocolo Solicitud (R). El cliente envía un mensaje de solicitud único al servidor. El cliente puede procesar inmediato después de que se envíe el mensaje de solicitud, ya que no es necesario esperar un mensaje de respuesta. Este protocolo se implementa a través de los datagramas de UDP y, además, sufre de las mismas falles de comunicación.
- El protocolo de solicitud-respuesta (RR). El protocolo RR es útil para la mayoría de los intercambios de clientes-servidores. No se requieren mensajes de acuse de recibo especial, ya que el mensaje de respuesta de un servidor se considera un acuse de recibo del mensaje de solicitud del cliente.
- El protocolo Solicitud-Respuesta (RRA). El protocolo RRA se basa en el intercambio de tres mensajes: Solicitud, respuesta y reconocimiento. El mensaje de reconocimiento contiene el *requestId* del mensaje de respuesta. Esto permitirá al servidor descartar las entradas de su historial.

5.2 Llamados a Procedimientos Remotos (RPC)

Se analizan tres aspectos que son importante para comprender este concepto:

- el estilo de programación promovido por RPC - programación con interfaces;
- la semántica de llamadas asociada con RPC;
- la transparencia y su relación con las llamadas a procedimientos remotos.

5.2.1 Programación con Interfases

La interfaz de un módulo especifica los procedimientos y las variables a las que se pueden acceder a través de otros módulos. En sistemas distribuidos las interfaces son programas distribuidos donde los módulos se ejecutan en procesos separados. En el modelo cliente-servidor, en particular, cada servidor proporciona un conjunto de procedimientos que están disponibles para uso de los clientes, [93] [54].

- No hay detalle de implementación lenguaje de programación. Hay una evolución del software
- No hay acceso a variables mediante ejecución de procesos remotos ni mecanismos pase de parámetros (llamada por valor o referencia)
- Direcciones en procesos locales no son válidas en los procesos remotos
- Mecanismo RPC se integra con el lenguaje de programación e incluye la notación para la definición de interfaces con mensajes input/output
- Escrita en variedades de lenguajes, C++, Java, Python
- Lenguajes de definición de interfaces (IDL) están diseñados para permitir que los procedimientos implementados en distintos lenguajes puedan ser invocados por otros

5.2.2 Semántica de llamadas de RPC

La operación *doOperation* se puede implementar de diferentes maneras para proporcionar diferentes garantías de entrega en el mensaje (Figura 5.1). Las principales opciones son:

- Reintentar mensaje de solicitud: controlar si se retransmite el mensaje de solicitud hasta se recibe una respuesta o supone que el servidor ha fallado.
- Filtrado duplicado: controlar cuándo se usan las retransmisiones y si se debe filtrar solicitudes duplicadas en el servidor.
- Retransmisión de resultados: controlar si se debe mantener un historial de mensajes de resultados para permitir que los resultados perdidos se retransmitan sin volver a ejecutar las operaciones en el servidor.

Por ejemplo, con una semántica **Puede-ser**, el invocador de la llamada recibe un resultado, en cuyo caso el invocador de la llamada sabe que el procedimiento se ejecutó al menos una vez, o una excepción que le informa que no se recibió ningún resultado. La semántica **Al-menos-una-vez**, puede ser lograda mediante la retransmisión de mensajes de solicitud, que enmascara las fallas de omisión del mensaje de solicitud o resultado. La semántica puede sufrir lo siguiente tipos de falla:

- fallas de bloqueo cuando falla el servidor que contiene el procedimiento remoto;
- fallas arbitrarias: en los casos en que el mensaje de solicitud se retransmite, el servidor puede recibirlo y ejecutar el procedimiento más de una vez, posiblemente causando valores incorrectos para ser almacenados o devueltos.

Con una semántica **como-máximo-una-vez**, la persona que llama recibe un resultado, en cuyo caso la persona que llama sabe que el procedimiento se ejecutó exactamente una vez, o una excepción que le informa que no se recibió ningún resultado, entonces el

Medidas de Tolerancia de Fallas			Semántica de Llamadas
Retransmisión mensaje	Filtrado de duplicados	Re-ejecución o re-transmisión de mensajes	
No	No aplica	No aplica	Puede ser
Si	No	Re-ejecución	Al-menos-una-vez
Si	Si	Retransmisión de la respuesta	como-máximo-una-vez

Table 5.1: Semántica de Llamadas RPC. Adaptado de [54]

procedimiento ha sido ejecutados ya sea una vez o no lo ha sido.

5.2.3 Transparencia

La elección de si el RPC debe ser transparente también está disponible para los diseñadores de IDL. Por ejemplo, en algunos IDL, una invocación remota puede generar una excepción cuando el cliente no puede comunicarse con un procedimiento remoto. Esto requiere que el programa cliente maneje tales excepciones, permitiéndole lidiar con tales fallas. Un IDL también puede proporcionar una facilidad para especificar la semántica de llamada de un procedimiento. Esto puede ayudar al diseñador del servicio; por ejemplo, si se elige la semántica de llamada al menos una vez para evitar los gastos generales de una vez como máximo, las operaciones deben diseñarse para que sean idempotentes.

5.2.4 Implementación de RPC

La llamada a procedimiento remoto (RPC) es una técnica orientada a la construcción de aplicaciones distribuidas basadas en cliente-servidor. Se basa en la extensión de la llamada de procedimiento local de manera que el procedimiento llamado no necesita existir en el mismo espacio de direcciones que el procedimiento de llamada. Los dos procesos pueden estar en el mismo sistema, o pueden estar en diferentes sistemas con una red que los conecta.

En la Figura 5.3 se muestra lo que ocurre cuando se hace una llamada a un proceso remoto:

1. El entorno de llamada se suspende, los parámetros del procedimiento se transfieren a través de la red al entorno donde se ejecutará el procedimiento.
2. Cuando finaliza el procedimiento y produce sus resultados, estos se transfieren de vuelta al entorno de llamada, donde se reanuda la ejecución como si regresara de una llamada de procedimiento regular.

En la Figura 5.4 se muestra un esquema de los componentes de la arquitectura de RPC.

El cliente que accede a un servicio incluye un procedimiento *stub* para cada procedimiento definido en la interfaz de servicio. El procedimiento *stub* se comporta como un procedimiento local para el cliente, pero en lugar de ejecutar la llamada, ordena (empaqueta) el identificador del procedimiento y los argumentos en un mensaje de solicitud, que envía a través de su módulo de comunicación al servidor.

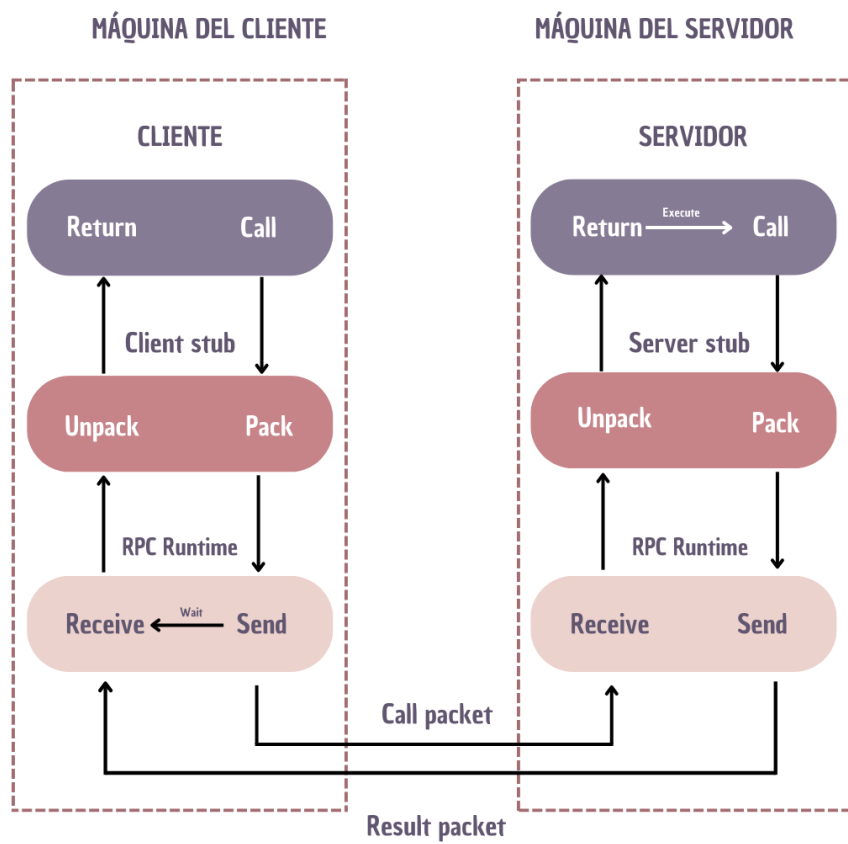


Figure 5.3: Llamada a Procedimiento Remoto.

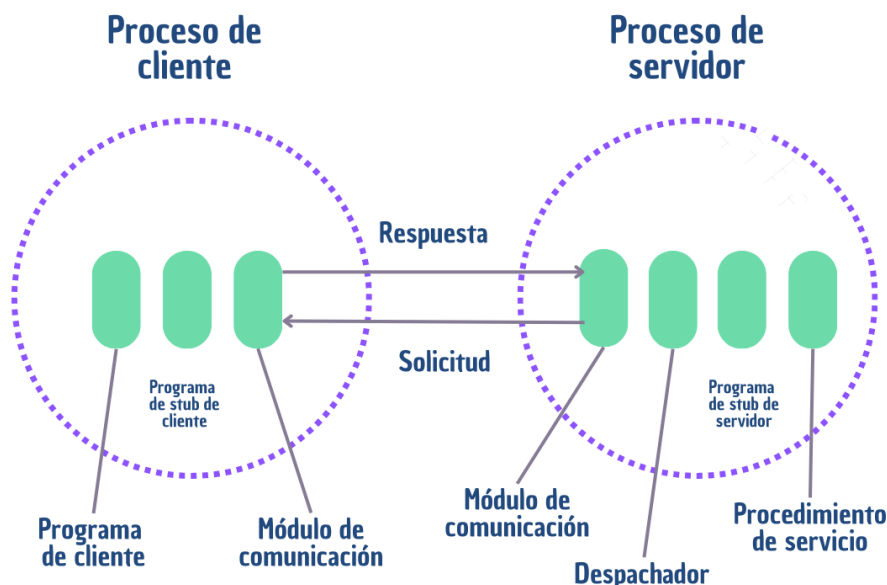


Figure 5.4: Arquitectura de RPC

Cuando llega el mensaje de respuesta, el módulo de comunicación desarma (desempaqueta) los resultados. El proceso del servidor contiene un despachador junto con un procedimiento de código auxiliar del servidor y un procedimiento de servicio para cada procedimiento en la interfaz de servicio. El despachador selecciona uno de los procedimientos *stub* del servidor, según el identificador de procedimiento en el mensaje de solicitud.

El procedimiento de *stub* del servidor luego desarma (desempaqueta) los argumentos en el mensaje de solicitud, llama al procedimiento del servicio correspondiente y calcula los valores de retorno para el mensaje de respuesta.

Los procedimientos de servicio implementan los procedimientos en la interfaz de servicio. Los procedimientos de *stub* de cliente y servidor y el despachador puede ser generado automáticamente por un compilador de interfaz a partir de la definición de interfaz del servicio.

RPC puede implementarse para tener una de las opciones de semántica de invocación o llamadas, generalmente se elige al menos una vez o como máximo una vez. Para lograr esto, el módulo de comunicación implementará las opciones de diseño deseadas en términos de retransmisión de solicitudes, tratamiento de duplicados y retransmisión de resultados.

5.3 Invocación a Métodos Remotos (RMI)

Los puntos en común entre RMI y RPC son los siguientes:

- Ambos admiten programación con interfaces.
- Están contruidos sobre protocolos de solicitud-respuesta y pueden ofrecer un rango de semántica de llamadas como al menos una vez y como máximo una vez.
- Ambos ofrecen un nivel similar de transparencia, es decir, las llamadas locales y remotas emplean la misma sintaxis, pero las interfaces remotas suelen exponer la distribución de la llamada subyacente, por ejemplo, admitiendo excepciones remotas.

Las siguientes diferencias conducen a una mayor expresividad cuando se trata de programación de aplicaciones y servicios distribuidos complejos:

- El programador puede utilizar todo el poder expresivo de la programación orientada a objetos para el desarrollo de software de sistemas distribuidos.
- Todos los objetos en un sistema basado en RMI tienen referencias de objeto únicas (ya sean locales o remoto), tales referencias de objeto también se pueden pasar como parámetros, ofreciendo así semántica de paso de parámetros significativamente más rica que en RPC.

Los siguientes conceptos están presentes en el modelo de objeto distribuido:

- Referencias a objetos remotos: otros objetos pueden invocar los métodos de un objeto remoto si tienen acceso a su referencia de objeto remoto.
- Interfaces remotas: cada objeto remoto tiene una interfaz remota que especifica qué de sus métodos se pueden invocar de forma remota.

5.3.1 Arquitectura RMI

La arquitectura se muestra en la Figura 5.5 ([93], [92], [54]):

Módulo de Comunicación Proporcionan semántica de invocación, como ejemplo, al menos-uno Transmite mensaje de solicitud/respuesta entre cliente y servidor Solicitud es (tipo de mensaje, IdSolicitud, ref objeto remoto, IdOperacion, Argumentos) En el servidor selecciona el despachador para la clase de objeto que se invoca. El despachador ubica la referencia local del objeto en el Módulo de Referencia Remota

Módulo de Referencia Remota Responsable de trasladar referencias de objetos locales a remotas y creación de referencias remotas Contiene una tabla de objetos remotos y una tabla para cada proxy local Actúa de la manera siguiente:

- 1era vez cuando se pasa un objeto remoto, el módulo de referencia remota crea una referencia al objeto remoto y se añade a la tabla de objetos remotos
- Cuando llega referencia a un objeto remoto, el módulo de referencia obtiene referencia al objeto local, la cual es un proxy o un objeto remoto. Si el objeto remoto no está en la tabla se crea el proxy y se añade el módulo de referencia remota

Criado (servant) Instancia de una clase que proporciona el cuerpo de un objeto remoto. Maneja el requerimiento remoto pasado por el esqueleto correspondiente. Se crea cuando se instancia el objeto remoto.

Proxy Proporciona que los métodos de invocación remota sean transparente al usuario comportandose como un objeto local. Esconde los detalles del empaquetamientos-desempaquetamiento de las referencias a objetos remotos.

Despachador Un servidor tiene un despachador y un esqueleto. Recibe la petición desde el módulo de comunicación. Usa el IdOperacion para seleccionar el método adecuado en el esqueleto

Esqueleto Implementa el método en la interface remota. Desempaqueta los argumentos e invoca el método correspondiente en el criado. Espera que la invocación se complete

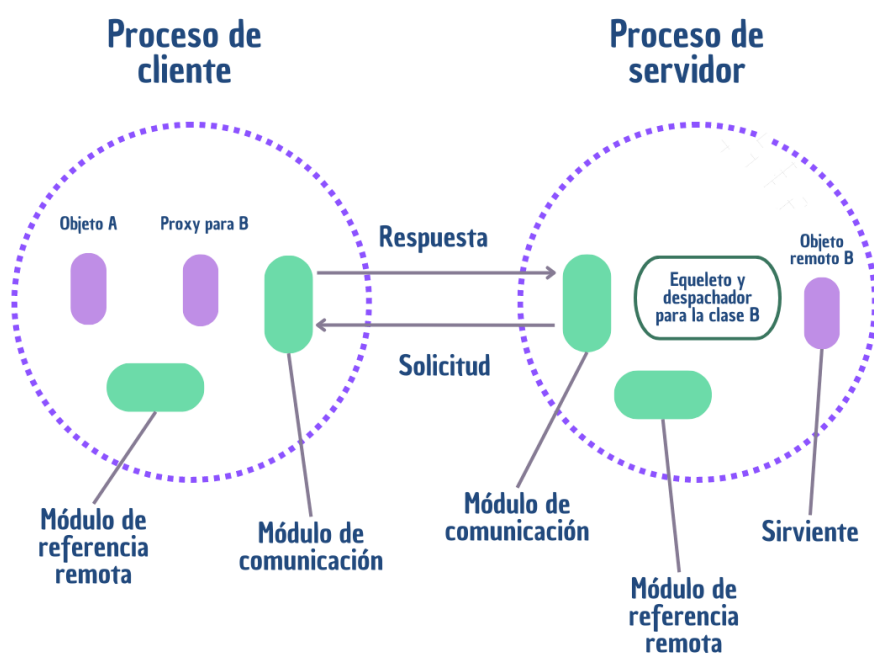


Figure 5.5: Arquitectura RMI.

y empaqueta los resultados

5.3.2 Colector de Basura Distribuida

Es un algoritmo distribuido cuya función es recolectar la basura distribuida. Establece una cooperación entre el colector local y un módulo añadido que colecciona la basura distribuida

Colector de basura distribuida:

1. Cada proceso servidor mantiene un conjunto de nombres de los procesos que proporcionan referencias a objetos remotos por cada uno de sus objetos remotos. Este conjunto se puede almacenar en una columna adicional de la tabla de objetos remotos.
2. Cuando un cliente C recibe una referencia remota a un objeto remoto en particular, B, hace una invocación `AddRef (B)` al servidor de ese objeto remoto y luego crea un proxy; el servidor agrega un apuntador C a B.
3. Cuando el recolector de basura de un cliente C nota que un proxy de un objeto remoto B no está accesible, hace una invocación `removeRef (B)` al servidor correspondiente y luego borra el proxy; el servidor elimina el apuntador C de B.
4. Cuando el apuntador B está vacío, el recolector de basura local del servidor recuperará el espacio ocupado por B a menos que existan apuntadores locales.

5.4 Caso de Estudio: API

Las tecnologías de comunicación entre procesos más utilizadas para las interacciones de solicitud y respuesta son **RPC**, **REST** y **GraphQL**. Por lo general, las API internas que se usan para las comunicaciones de servicio a servicio dentro de una organización se implementan con un marco RPC de alto rendimiento como gRPC.

gRPC

gRPC, Google Remote Procedure Call, es un sistema de llamada a procedimiento remoto de código abierto desarrollado inicialmente en Google.

Por el contrario, las API externas disponibles para el público tienden a basarse en **REST** o **GraphQL**. En lo que queda, presentaremos estas tecnologías.

5.4.1 ¿Que es un API?

Un desarrollador usa una API cuando escribe software que interactuará con un sistema de software externo cerrado. El sistema de software externo proporciona una API como un conjunto estándar de herramientas que todos los desarrolladores pueden usar. Ejemplos son las APIs de Google, YouTube, Twitter, entre otras.

5.4.2 ¿Que es REST?

REST (REST, Representational State Transfer) es un estilo de arquitectura de software propuesto por [65]. **REST** es una arquitectura orientada a los recursos en la que los usuarios gestionarían los recursos web realizando operaciones HTTP como GET, PUT, POST y DELETE. La red de recursos se puede considerar como una máquina de estado virtual y las acciones (GET, PUT, POST, DELETE) son cambios de estado dentro de la

máquina. Esto significa que cuando se usa el paradigma **REST** para construir un **API REST**, se tiene un medio uniforme para crear, leer y actualizar datos usando **URL HTTP** simples con un conjunto estándar de verbos **HTTP** [84], [62].

Mientras **REST** esta relacionado con una serie de restricciones, el término **RESTful** hace referencia a un API que se adhiere a los siguientes criterios:

- Arquitectura cliente-servidor compuesta de clientes, servidores y recursos, con la gestión de solicitudes a través de HTTP.
- Comunicación entre el cliente y el servidor sin estado (*stateless*).
- Uso del **caché** para almacenar datos y optimizar las interacciones entre el cliente y el servidor.
- Una interfaz uniforme entre los elementos, para que la información se transfiera de forma estandarizada: recursos identificables de manera que el cliente pueda manipularlos.
- Un sistema en capas que organiza en jerarquías cada uno de los servidores (los encargados de la seguridad, del equilibrio de carga, etc.) y que participan en la recuperación de la información solicitada.
- código por demanda, es decir, a solicitud del cliente.

Cuando el cliente envía una solicitud a través de una API de RESTful, esta transfiere una representación del estado del recurso requerido a quien lo haya solicitado o al extremo. La información se entrega por medio de HTTP en uno de estos formatos: JSON (JavaScript Object Notation), HTML, XLT, Python, PHP o texto sin formato. JSON es el lenguaje de programación más popular, ya que tanto las máquinas como las personas lo pueden comprender y no depende de ningún lenguaje, a pesar de que su nombre indique lo contrario.

5.4.3 GraphQL

GraphQL es un lenguaje de consultas para las API, desde la perspectiva del consumidor de esas API de datos. La palabra *graph* en **GraphQL** proviene del hecho de que la mejor manera de representar datos en el mundo real es con una estructura de datos similar a un grafo, [49], [47].

Como alternativa a **REST**, **GraphQL** permite que los desarrolladores creen consultas para extraer datos de varias fuentes en una sola llamada a la API. **GraphQL** proporciona un enfoque de API web en el que los clientes definen la estructura de los datos que devolverá el servidor. Esto puede impedir el almacenamiento en caché de los resultados de la consulta.

Problema de API REST

Una API REST es una colección de puntos finales donde cada punto final representa un recurso. Cuando un cliente necesita datos sobre múltiples recursos, tiene que realizar múltiples solicitudes a esa API REST y luego reunir los datos combinando las múltiples respuestas que recibe.



Comunicación Indirecta

6	Comunicación Indirecta	111
6.1	Grupos	111
6.2	Publicación-Suscripción	115
6.3	Colas	120
6.4	Casos de Estudio: Patrón y mensajería Pub/-sub	122
7	Arquitectura orientada a Servicios	125
7.1	Informática orientada a Servicios	125
7.2	Servicios Web	129
7.3	REST	141
7.4	Caso de Estudio: Monolitos vs Microservicios	148
8	Sistemas Punto a Punto	153
8.1	Introducción	153
8.2	Evolución de los sistemas P2P	155
8.3	Caso de Estudio. Pastry	160



6. Comunicación Indirecta

En comunicación indirecta la naturaleza del intermediario y del acoplamiento varían de un enfoque a otro y entre sistemas: se puede establecer conceptos de acoplamiento tanto temporal como espacial que determinan como el receptor y el emisor coinciden o no en el tiempo y en espacio.

En la tabla 6.1 se plasman las posibles variantes en este tipo de enfoque. Por ejemplo, las técnicas consideradas hasta ahora se basan en un acoplamiento directo o sistemas fuertemente acoplados, como las arquitecturas clientes-servidor. Son arquitecturas acopladas en tiempo y espacio. Por otra parte, las aplicaciones como el correo electrónico, son arquitecturas desacopladas en tiempo y acopladas en espacio, ver descripción en la parte superior, derecha de la tabla 6.1.

Mientras que la multidifusión puede catalogarse como arquitecturas acoplada en tiempo, desacopladas en espacio, descrita en la parte inferior e izquierda de la tabla 6.1. Y las arquitecturas que calzan en la categoría de desacoplamiento en espacio y tiempo, como las arquitecturas Pub-sub y colas de mensajes, ubicada en la parte inferior, derecha.

Acoplamiento

El concepto de Acoplamiento propuesto por Yourdon y Constantine en su libro, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, [95]

En lo que sigue, se estudian algunas de las arquitecturas de comunicación indirecta: Grupos, pub-sub y colas.

6.1 Grupos

La comunicación grupal es una abstracción sobre la comunicación de multidifusión y se puede implementar a través de multidifusión IP o una red de superposición equivalente, teniendo como valor adicional la gestión de pertenencia al grupo, detección de fallas y garantía de confiabilidad y pedidos, [92], [54].

	Acoplado en tiempo	Desacoplado en tiempo
Acoplado en espacio	Comunicación dirigida a un receptor o a varios receptores de un mensaje. Los receptores deben existir en ese momento del tiempo.	Comunicación dirigida a un receptor(es) de un mensaje. Receptores o emisores pueden no existir en ese momento del tiempo.
Des acoplado en espacio	Emisor no necesita saber la identidad del receptor o receptores. Emisor y receptor deben existir al mismo tiempo.	Emisor no necesita saber la identidad del receptor o receptores. Los receptores o emisores pueden no existir en ese momento del tiempo.

Table 6.1: Acoplamiento en Espacio y Tiempo. Adaptado de [54]

6.1.1 Modelo de Programación

En la comunicación grupal, el concepto central es el de un **grupo** con una **membresía de grupo** asociada y con los procesos de **unirse** o **salir** del grupo ver Figura 6.1. Los procesos pueden enviar un mensaje a este grupo y hacer que se propague a todos los miembros (o a un miembro) del grupo con ciertas garantías en términos de confiabilidad y orden. Así, la comunicación de grupo implementa la comunicación de multidifusión, así como también la comunicación unidifusión.

El uso de una única operación de envío permite un uso eficiente del ancho de banda. También es relevante en términos de garantías de entrega del mensaje ya que la operación de envío del mensaje se asocia a un proceso. Si ocurre una falla y algún mensaje no se entrega, se puede implementar el reenvío de mensajes.

6.1.2 Membresía del Grupo

Un servicio de membresía grupal tiene cuatro tareas principales:

Interfaz para cambios de membresía de grupo : el servicio de membresía proporciona operaciones para crear/destruir grupos de procesos y para agregar/retirar un proceso hacia o desde un grupo. En la mayoría de los sistemas, un solo proceso puede pertenecer a varios grupos al mismo tiempo (grupos superpuestos).

Detección de fallas : el servicio monitorea a los miembros del grupo no solo en caso de que se bloqueen, sino también en caso de que se vuelvan inaccesibles debido a una falla de comunicación. El detector marca los procesos como sospechosos o no sospechosos. El servicio utiliza el detector de fallas para tomar una decisión sobre la situación de la membresía del grupo: excluye un proceso de la membresía si se sospecha que ha fallado o si se ha vuelto inalcanzable.

Notificación de cambios en la membresía del grupo : el servicio notifica a los miembros del grupo cuando se agrega un proceso, o cuando se excluye un proceso (por falla o cuando el proceso se retira deliberadamente del grupo).

Difusión de direcciones de grupo : cuando un proceso difunde un mensaje en forma múltiple, proporciona el identificador de grupo en lugar de una lista de procesos del grupo. El servicio de administración de membresía expande el identificador a la membresía del grupo actual para su entrega. El servicio puede coordinar la entrega de multidifusión con cambios de membresía controlando la expansión de direcciones. Es decir, puede decidir sistemáticamente dónde enviar un mensaje

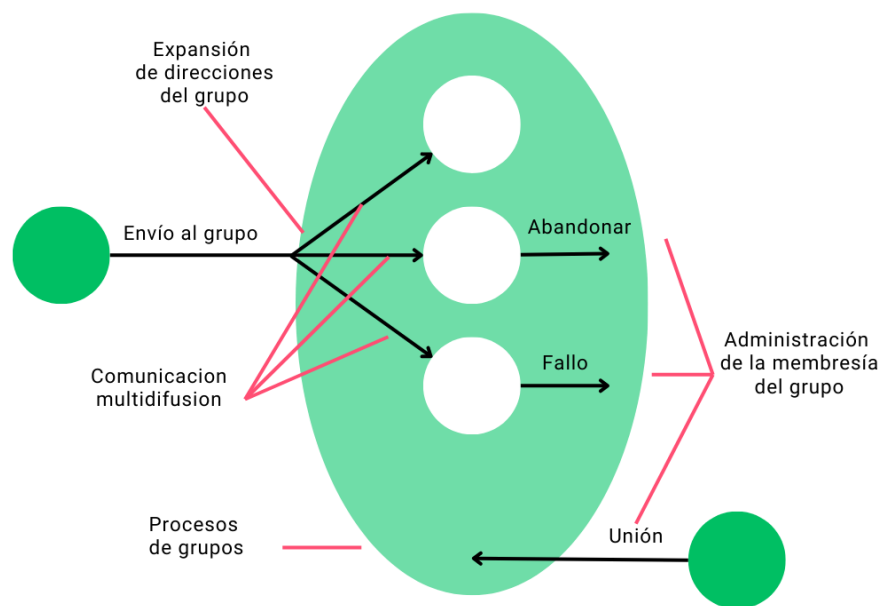


Figure 6.1: Model de Programación. Tomado de [54]

determinado, aunque la membresía pueda cambiar durante la entrega.

6.1.3 Características de la arquitectura de Grupos

Grupos de procesos La mayor parte del trabajo en servicios grupales se centra en grupos donde las entidades comunicantes son Procesos. El nivel de servicio proporcionado por los grupos de procesos es similar al de los sockets.

Grupos de objetos Es una colección de objetos (instancias de la misma clase) que procesan el mismo conjunto de invocaciones al mismo tiempo, y cada una devuelve respuestas. Los clientes invocan operaciones en un único objeto local, que actúa como proxy del grupo. El proxy utiliza un sistema de comunicación grupal para enviar las invocaciones a los miembros del grupo de objetos. Los parámetros de objeto y los resultados se empaquetan como en RMI y las llamadas asociadas se envían automáticamente a los objetos/métodos de destino.

Grupos abiertos Un grupo está abierto si los procesos fuera del grupo pueden enviarle mensajes, Figura 6.2. Los grupos abiertos son útiles, por ejemplo, para entregar mensajes a grupos de procesos interesados.

Grupos cerrados Se dice que el grupo está cerrado si solo los miembros del grupo pueden enviar mensajes multidifusión. Un proceso en un grupo cerrado se entrega a sí mismo cualquier mensaje que difunde al grupo, ver Figura 6.2. Los grupos cerrados de procesos son útiles, por ejemplo, para servidores que cooperan para enviarse mensajes entre sí que solo ellos deberían recibir.

Grupos Superpuestos Entidades (procesos u objetos) pueden ser miembros de varios grupos

Grupos no Superpuestos Implican que la membresía no se superpone (es decir, cualquier proceso pertenece como máximo a un grupo)

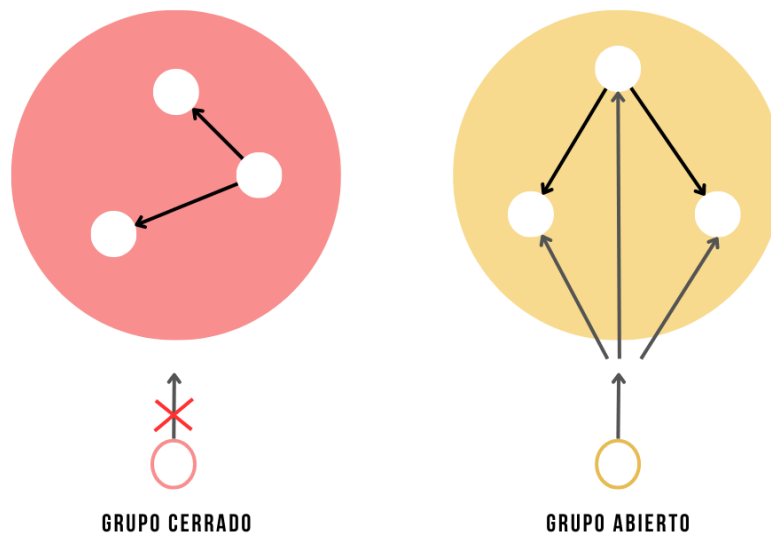


Figure 6.2: Grupos cerrados y abiertos. Tomado de [54]

6.1.4 Confiabilidad y ordenamiento en multidifusión

Otras características atribuida a los grupos esta la confiabilidad y el ordenamiento de los mensajes.

Confiabilidad La confiabilidad en la comunicación se ha definido en términos de las siguientes propiedades:

Integridad El mensaje recibido es el mismo que el enviado, y no se entregan dos veces.

Validez Cualquier mensaje saliente es eventualmente entregado. La interpretación de multidifusión confiable se basa en estas propiedades, con integridad definida en términos de entregar el mensaje correctamente como máximo una vez, y la validez interpretado como que garantiza que un mensaje enviado finalmente será entregado.

Acuerdo Para extender la semántica y cubrir la entrega a múltiples receptores, la propiedad *acuerdo* indica que si el mensaje se entrega a un proceso, entonces se entrega a todos los procesos del grupo.

Ordenamiento Además de las garantías de fiabilidad, la comunicación grupal exige garantías adicionales en términos del orden relativo de los mensajes entregados a múltiples destinos. Para contrarrestar los retrasos y desorden en la entrega, los servicios de comunicación grupal ofrecen multidifusión ordenada, con la opción de una o más de las siguientes propiedades:

Orden FIFO : ordenamiento primero en entrar, primero en salir (FIFO) se ocupa de preservar el pedido desde la perspectiva del remitente del proceso, en el sentido de que si un proceso envía un mensaje antes que otro, se entregará en este orden en todos los procesos del grupo.

Ordenamiento causal : el ordenamiento causal tiene en cuenta las relaciones causales entre mensajes, en el sentido de que si un mensaje ocurre antes que otro mensaje en el sistema distribuido, esta relación causal se conservará en la entrega de los

mensajes asociados en todos los procesos

Ordenamiento total : En el orden total, si un mensaje se entrega antes que otro mensaje en un proceso, se conservará el mismo orden en todos los procesos.

6.2 Publicación-Suscripción

Las principales entidades en un sistema publicación-suscripción son los **editores** y **suscriptores** de contenido [89]. Un **publicador** (*publisher*) detecta un evento y luego lo publica en forma de notificación. Una **notificación** encapsula información relacionada con el evento observado. La notificación denota que ha ocurrido un evento observado. Un **evento** representa cualquier transición de estado discreta que ha ocurrido y se señala desde una entidad a un número de otras entidades.

Un **suscriptor**, por ejemplo, podría expresar interés en todos los eventos relacionados con este libro de texto, como la disponibilidad de una nueva edición o actualizaciones del sitio web relacionado. La tarea del sistema de publicación y suscripción es hacer coincidir las suscripciones con los eventos publicados y garantizar la entrega correcta de las notificaciones de eventos. Un evento dado se entregará potencialmente a muchos suscriptores y, por lo tanto, en publicación y suscripción se usan paradigmas de comunicaciones de uno a muchos.

6.2.1 Aplicaciones de los sistemas de publicación-suscripción

Los sistemas de publicación-suscripción se utilizan en una variedad de dominios de aplicación, en particular los relacionados con la difusión de eventos a gran escala. Ejemplos incluyen [89]:

- **GUI**, en las que los sistemas pub/sub se aplica como el pegamento que conecta los diversos componentes entre sí. Un ejemplo es el patrón de diseño **MVC** (Modelo Vista Controlador) muy utilizado en GUIs y su componente el patrón de observador.
- **Push de información**, en el que se publica el contenido al usuario. Este es un requisito para aplicaciones que dependen de datos en tiempo real o casi en tiempo real.
- **Filtrado de información y entrega dirigida** utilizada por los servicios de alerta y presencia (Google Alerts, etc.), tiendas de aplicaciones, servicios de corretaje de RSS, etc. Los ejemplos incluyen **XMPP Pub/sub**, **Pubsubhubbub**, **Facebook Messenger and Chat** y **Twitter**.
- **Plano de señalización**, en el que pub/sub asegura que los eventos asíncronos se entregan en tiempo real o casi en tiempo real desde la publicación de componentes hasta la suscripción de componentes. Las aplicaciones de ejemplo incluyen sistemas industriales y tácticos. **DDS** (Data Distribution Systems) es el estándar clave para estos sistemas.
- La arquitectura orientada a servicios (**SOA**) y las aplicaciones comerciales se basan en publicación/suscripción en el bus de servicios empresariales (**ESB**). El **ESB** normalmente se implementa con un *broker* de mensaje XML.
- **Procesamiento de Eventos Complejos (CEP)** para análisis de datos. CEP se utiliza ampliamente en varias aplicaciones comerciales, por ejemplo, comercio algorítmico y detección de fallas.
- **Computación en la nube**, en la que pub/sub y colas de mensajes se utilizan para conectar los componentes de la nube.
- **Internet de las cosas**, en el que pub/sub conecta los sensores y actuadores entre sí

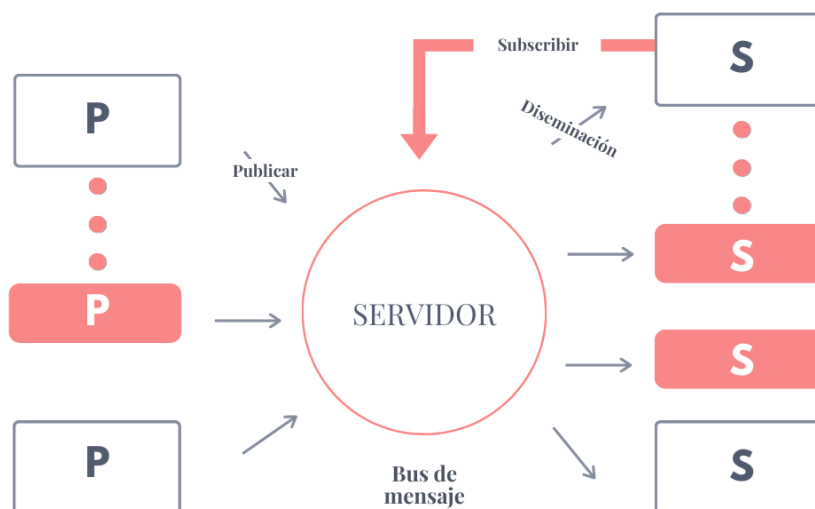


Figure 6.3: Paradigma Publicación Suscripción.

con recursos de Internet.

- Juegos multijugador en línea, en los que pub/sub se usa para sincronizar el estado del juego en jugadores y servidores

6.2.2 Modelo de Programación

El modelo de programación en los sistemas de publicación-suscripción se basa en un pequeño conjunto de operaciones, ver Figura 6.3. Los editores difunden un evento e a través de una operación de *publicacion*(e) y los suscriptores expresan su interés en un conjunto de eventos a través de suscripciones. En particular, logran esto a través de una operación de *suscripcion*(f) donde f se refiere a un filtro, es decir, un patrón definido sobre el conjunto de todos los eventos posibles.

La expresividad de los filtros está determinada por modelo de la suscripción. Los suscriptores pueden revocar este interés mediante la correspondiente operación de *cancelacion* – *suscripcion*(f). Cuando los eventos llegan a un suscriptor, los eventos se entregan usando una operación de *notificacion*(e).

Algunos sistemas complementan el conjunto de operaciones anuncios. Con los anuncios, los editores tienen la opción de declarar la naturaleza de los eventos futuros a través de una operación de *anuncios*(f).

Modelo de Suscripción La expresividad de los sistemas de publicación-suscripción está determinada por el **modelo de suscripción**, que cuentan con una serie de esquemas o filtros [89] [54]:

Basado en canales : en este enfoque, los editores publican eventos en canales con nombre y los suscriptores luego se suscriben a uno de estos canales para recibir todos los eventos enviados a ese canal. Este esquema es el único que se define en un canal físico.

Basado en temas : se asume que cada notificación se expresa en términos de varios campos, con un campo que denota el tema. Las suscripciones se definen en función del tema de interés. Este enfoque es equivalente a los enfoques basados en canales con la diferencia de que los temas se definen implícitamente, pero se declaran como parte de un campo en el enfoque basados en temas.

Basado en contenidos : son una generalización de enfoques basados en temas que permiten la expresión de suscripciones en una variedad de campos en solo una notificación del evento. Más específicamente, un filtro basado en contenido es una consulta definida en términos de composiciones de restricciones sobre los valores de los atributos del evento. Por ejemplo, un suscriptor podría expresar interés en eventos relacionados con el tema de los sistemas de publicación-suscripción, donde el sistema en cuestión es el *Servicio de eventos CORBA* y el autor es *Tim Kindberg* o *Gordon Blair*.

Basado en tipo : las suscripciones se definen en términos de tipos de eventos y la coincidencia se define en términos de tipos o subtipos del filtro dado. Este enfoque puede expresar una variedad de filtros, desde un filtrado basado en nombres de tipo generales hasta consultas más detalladas que definen atributos y métodos de un objeto dado. Estos filtros detallados son similares en expresividad a los enfoques basados en contenido.

6.2.3 Consideraciones de Implementación

Implementaciones Centralizadas El procesamiento de eventos y notificaciones se puede implementar fácilmente en editores y con un intermediario centralizado, ver Figura 6.4.

En la implementación centralizada, ocurre que:

- El enfoque más simple es centralizar la implementación en un solo nodo con un servidor en ese nodo que actúa como un intermediario de eventos.
- Los publicadores publican eventos y (opcionalmente) envían anuncios al corredor, y los suscriptores envían suscripciones al corredor y reciben notificaciones a cambio.
- La interacción con el corredor se realiza a través de una serie de mensajes punto a punto; esto se puede implementar mediante el paso de mensajes o la invocación remota.

Este enfoque es sencillo de implementar, pero el diseño carece de resiliencia y escalabilidad, ya que el nodo centralizado representa un punto único de posibles fallas del sistema y un cuello de botella para el rendimiento.

Implementaciones Distribuidas En este esquemas, el nodo centralizado es reemplazado por una red de corredores que cooperan para ofrecer la funcionalidad deseada. En la visión distribuida, el sistema pub-sub distribuido se implementa como una red de intermediarios o enrutadores en la capa de aplicación que se comunican mediante el uso de las primitivas de capa inferior, normalmente TCP/IP. En la Figura 6.5 se esquematiza una red pub-sub distribuida donde cada componente de la red de corredores son componentes intermediarios o enrutadores.

Dicho enfoque tienen el potencial de sobrevivir a las fallas de los nodos y se ha demostrado que pueden funcionar bien en implementaciones a escala de Internet. Como alternativa, es posible tener una implementación completamente nodo-a-nodo (*peer-to-peer*) de un sistema de publicación-suscripción. En este enfoque, no hay distinción entre editores, suscriptores y corredores; todos los nodos actúan como intermediarios,

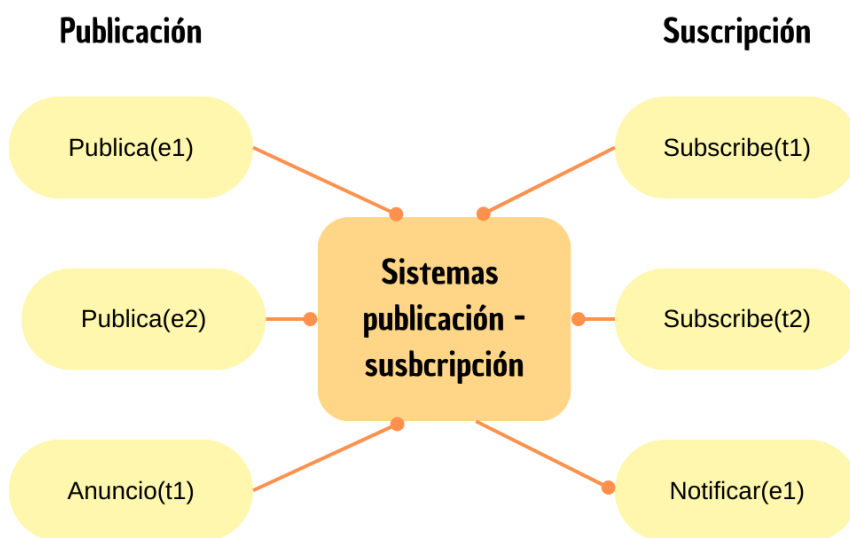


Figure 6.4: Publicación-Suscripción centralizado.

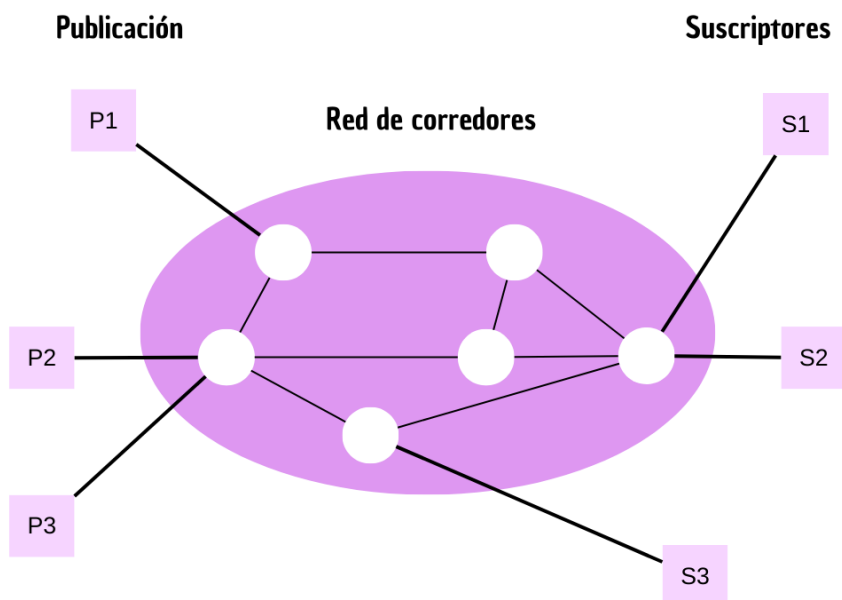


Figure 6.5: Publicación-Suscripción distribuido. Tomado de [54]

implementando de manera cooperativa la funcionalidad de enrutamiento de los eventos requeridos

6.2.3.1 Enfoques de Implementación

Existe una variedad de enfoques de implementación [89] [54]:

Inundación (Flooding) el enfoque más simple se basa en Inundación. Opera de la siguiente manera:

- Se envía una notificación de evento a todos los nodos de la red y luego se realiza el emparejamiento con el suscriptor del evento.
- Como alternativa, la inundación se puede utilizar para enviar suscripciones a todos los posibles publicadores; la coincidencia se realiza en el publicador y los eventos coincidentes se envían directamente a los suscriptores relevantes mediante la comunicación punto a punto. La inundación se puede implementar utilizando una función de difusión o multidifusión subyacente.
- Alternativamente, los intermediarios pueden organizarse en un gráfico acíclico en el que cada uno envía notificaciones de eventos entrantes a todos sus vecinos. Este enfoque tiene el beneficio de la simplicidad, pero puede resultar en una gran cantidad innecesaria de tráfico de red.

Filtrado : Se conoce como enrutamiento basado en filtrado. Los corredores envían notificaciones a través de la red solo donde hay una ruta a un suscriptor válido. Funciona así:

- La propagación de la información de suscripción se realiza a través de la red hacia los editores potenciales y luego se almacena el estado asociado en cada corredor.
- Específicamente, cada nodo debe mantener una **lista de vecinos** que contenga a todos los vecinos conectados en la red de corredores, una **lista de suscripción** que contenga todos los suscriptores conectados directamente atendidos por este nodo, y una **tabla de enrutamiento**. Esta tabla de enrutamiento mantiene una lista de vecinos y suscripciones válidas para ese camino.
- Este enfoque exige una implementación de la coincidencia en cada nodo en la red de corredores: en particular, en la función de coincidencia lleva a cabo la notificación de eventos y una lista de nodo junto con la suscripción y las devoluciones asociadas en el conjunto de nodos donde la notificación coincide con la suscripción.

El algoritmo específico para este enfoque de filtrado se captura en las Figura 6.6 y opera de la siguiente manera:

- Cuando un corredor recibe en la solicitud de publicación de un nodo dado, debe pasar esta notificación a todos los nodos conectados donde hay una suscripción con coincidencia al evento y también decide dónde propagar este evento a través de la red de corredores.
- Las líneas 2 y 3 logran el primer objetivo al igualar el evento contra la lista de suscripción y luego reenviar el evento a todos los nodos con suscripciones coincidentes (la lista de coincidencias).
- Las líneas 4 y 5 luego usan la función de coincidencia nuevamente, esta vez coincide con el evento contra la tabla de enrutamiento y reenvía solo a las rutas que conducen a una suscripción (la Lista FWD).
- Los corredores también deben lidiar con los eventos de suscripción entrantes.

```

upon receive publish(event  $e$ ) from node  $x$                                 1
    matchlist := match( $e$ , subscriptions)                                2
    send notify( $e$ ) to matchlist;                                         3
    fwdlist := match( $e$ , routing);                                        4
    send publish( $e$ ) to fwdlist -  $x$ ;                                     5
upon receive subscribe(subscription  $s$ ) from node  $x$                     6
    if  $x$  is client then                                                  7
        add  $x$  to subscriptions;                                         8
    else add( $x$ ,  $s$ ) to routing;                                           9
    send subscribe( $s$ ) to neighbours -  $x$ ;                             10

```

Figure 6.6: Algoritmo de Filtrado. Tomado de [54]

Si el evento de suscripción es de un suscriptor inmediato conectado, entonces esta suscripción se registra en la tabla de suscripciones (líneas 7 y 8).

- De lo contrario, el corredor es un nodo intermediario; Este nodo ahora sabe que existe un camino hacia esta suscripción y, por lo tanto, se agrega una entrada apropiada a la tabla de enrutamiento (línea 9).
- En ambos casos, este evento de suscripción se pasa a todos los vecinos aparte del nodo de origen (línea 10).

Encuentros (Rendezvous) : Para comprender este enfoque, es necesario ver el conjunto de todos los eventos posibles como un espacio de eventos y dividir la responsabilidad de este espacio de eventos entre el conjunto de agentes de la red. En particular, este enfoque define los nodos de encuentro, que son nodos intermediarios responsables de un subconjunto determinado del espacio de eventos. Para lograr esto, un algoritmo de enrutamiento basado en encuentros debe definir dos funciones:

- En primer lugar, SN toma una suscripción determinada, s , y devuelve uno o más nodos de encuentro que asumen la responsabilidad de esa suscripción. Cada uno de estos nodos de encuentro mantiene una lista de suscripción como en el método de filtrado anterior, y reenvía todos los eventos coincidentes al conjunto de nodos de suscripción.
- En segundo lugar, cuando se publica un evento e , la función $EN(e)$ también devuelve uno o más nodos de encuentro, esta vez que corresponde a la coincidencia del evento e con las suscripciones en el sistema.
- Tenga en cuenta que tanto $SN(s)$ como $EN(e)$ devuelven más de un nodo si la confiabilidad es un problema.
- Tenga en cuenta también que este enfoque solo funciona si la intersección de $SN(s)$ y $EN(e)$ no está vacía para una e dada que coincide con s .

El código correspondiente para el enrutamiento basado en citas se muestra en la Figura 6.7

6.3 Colas

Mientras que los grupos y publicación-suscripción proporcionan un estilo de comunicación de uno a varios, las colas de mensajes proporcionan un servicio punto a punto utilizando el concepto de cola de mensajes como una indirección, logrando así las propiedades deseadas de desacoplamiento de espacio y tiempo. Son punto a punto en el sentido de que el

```

upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist <- match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;
upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;

```

Figure 6.7: Algoritmo de Encuentros. Tomado de [54]

remitente coloca el mensaje en una cola y luego es eliminado por un solo proceso. Las colas de mensajes también se conocen como *Middleware orientado a mensajes*.

6.3.1 Modelo de programación

El modelo de programación que ofrecen las colas de mensajes es muy sencillo. Ofrece un acercamiento a la comunicación en sistemas distribuidos a través de colas. En particular, los procesos productores pueden enviar mensajes a una cola específica y otros procesos (consumidores) pueden recibir mensajes de esta cola. Se admiten tres estilos de recepción:

- una recepción con bloqueo, que se bloqueará hasta que esté disponible un mensaje apropiado;
- una recepción sin bloqueo (una operación de sondeo), que verificará el estado de la cola y devolverá un mensaje si está disponible, o una indicación de no disponible en caso contrario;
- una operación de notificación, que emitirá una notificación de evento cuando un mensaje esté disponible en la cola asociada.

6.3.2 Características

Entre sus características:

- Varios procesos pueden enviar mensajes a la misma cola y, del mismo modo, varios receptores pueden eliminar mensajes de una cola.
- La política de colas es el primero en entrar, primero en salir (FIFO), pero la mayoría de las implementaciones de colas de mensajes también admiten el concepto de prioridad, con los mensajes de mayor prioridad entregados primero. Los procesos de consumidor también pueden seleccionar mensajes de la cola según las propiedades de un mensaje.
- Un mensaje consta de un destino (un identificador único que designa la cola de destino), metadatos asociados con el mensaje, incluidos campos como la prioridad del mensaje y el modo de entrega, y también el cuerpo del mensaje.

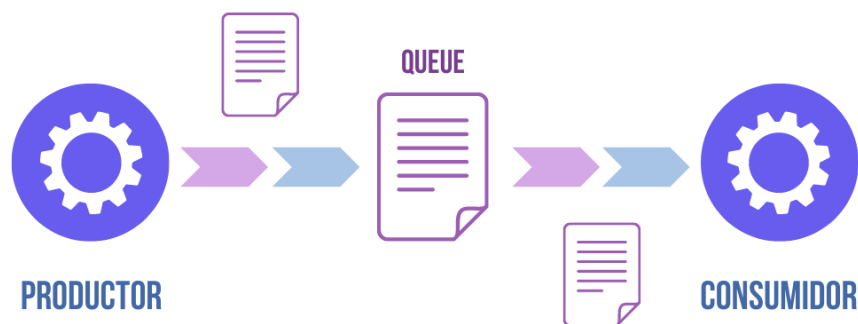


Figure 6.8: Modelo de Programación.

- Los mensajes son persistentes, es decir, las colas de mensajes almacenarán los mensajes indefinidamente (hasta que se consuman) y también enviarán los mensajes al disco para permitir una entrega confiable.
- Cualquier mensaje enviado se recibe eventualmente (validez) y el mensaje recibido es idéntico al enviado, y ningún mensaje se entrega dos veces (integridad). Por lo tanto, los sistemas de cola de mensajes garantizan que los mensajes se entregarán (y se entregarán una vez), pero no pueden decir nada sobre el momento de la entrega.

6.4 Casos de Estudio: Patrón y mensajería Pub/sub

6.4.1 Mensajería Publicación Suscripción

En arquitectura de software, **Mensajería Pub/Sub**, también conocido como **patrón Pub/Sub** [35], es un patrón de mensajería que proporciona un marco para el intercambio de mensajes en el que los remitentes de mensajes, llamados editores, no programan los mensajes para que se envíen directamente a receptores específicos, llamados suscriptores, sino que clasifican los mensajes publicados en clases sin saber qué suscriptores, si los hay. De igual forma, los suscriptores expresan interés en una o más clases de mensajes y solo reciben mensajes que son de su interés, sin saber qué editores, si los hay.

Los sistemas de Mensajería Pub/sub es un hermano del paradigma de la cola de mensajes y, por lo general, es una parte de un sistema de middleware orientado a mensajes. La mensajería Pub/sub permite el acoplamiento flexible y la escala entre el remitente de mensajes (editores) y los receptores (suscriptores) en los agentes de mensajería a los que se suscriben. Algunas de las tecnologías comunes que utilizan Pub/Sub Messaging son RabbitMQ, Kafka, Redis, etc.

Los mensajes se envían desde un editor a los suscriptores a medida que están disponibles.

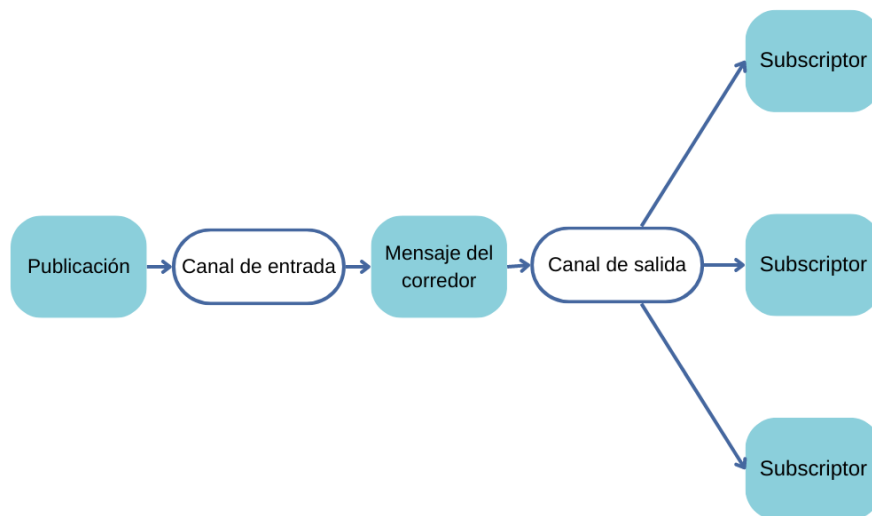


Figure 6.9: Patrón Publicación-Suscripción.

Los editores son servicios que envían mensajes al intermediario de mensajes. Luego, los suscriptores se suscriben a los corredores de mensajes que les interesan para escuchar estos mensajes.

En la Figura 6.9 se muestra los componentes de este sistema.

La mensajería de publicación/suscripción tiene los siguientes beneficios [1]:

- Acoplamiento bajo entre los componentes, lo que hace que su sistema sea más modular y flexible.
- Alta escalabilidad (en teoría, Pub/Sub permite que cualquier cantidad de editores se comunique con cualquier cantidad de suscriptores).
- Mejora la fiabilidad. La mensajería asíncrona ayuda a que las aplicaciones continúen funcionando sin problemas bajo cargas mayores y manejen fallas intermitentes de manera más efectiva.
- Permite el procesamiento diferido o programado. Los suscriptores pueden esperar para recoger los mensajes hasta las horas de menor actividad, o los mensajes pueden enrutarse o procesarse de acuerdo con un cronograma específico.
- Permite una integración entre sistemas que utilizan diferentes plataformas, lenguajes de programación o protocolos de comunicación, así como entre sistemas locales y aplicaciones que se ejecutan en la nube.
- Comunicación asíncrona basada en eventos que es ideal para aplicaciones de baja latencia en tiempo real.

Patrón Pub/Sub

Lectura recomendada: [Publish-Subscribe Pattern: The Most Used Patterns in JavaScript](#)

6.4.2 Rabbit MQ

RabbitMQ es un software de intermediario (broker o corredores) de mensajes de código abierto (a veces llamado middleware orientado a mensajes) que originalmente implementó el Protocolo de cola de mensajes avanzado (AMQP) y desde entonces se ha ampliado con una arquitectura para admitir el Protocolo de mensajería orientada a texto en streaming (**STOMP**), MQ Telemetry Transport (**MQTT**) y otros protocolos.

Escrito en **Erlang**, el servidor RabbitMQ se basa en el marco de Open Telecom Platform para la agrupación en clústeres y la conmutación por error. Las bibliotecas de cliente para interactuar con el intermediario están disponibles para los principales lenguajes de programación. El código fuente se publica bajo la licencia pública de Mozilla.

Erlang

Erlang es un lenguaje de programación que se utiliza para construir sistemas de software en tiempo real escalables masivamente con requisitos de alta disponibilidad. Algunos de sus usos se encuentran en telecomunicaciones, banca, comercio electrónico, telefonía informática y mensajería instantánea. Más de **Elang** en esa dirección.

Rabbit MQ

Conozca acerca de este tema en **Rabbit MQ**



7. Arquitectura orientada a Servicios

La arquitectura del software es la estructura(s) del sistema que incluye componentes de software, las propiedades visibles externamente de esos componentes, las relaciones entre ellos y las restricciones sobre su uso. Es una abstracción de los elementos de tiempo de ejecución de un sistema de software durante alguna fase de su operación. Un sistema puede estar compuesto por varios niveles de abstracción y fases de operación, cada una con su propia arquitectura de software. [65].

7.1 Informática orientada a Servicios

Informática orientada a Servicios representa una plataforma informática distribuida de nueva generación. Incluye su propio paradigma de diseño y principios de diseño, patrón de diseño, lenguajes de patrones, un modelo arquitectónico distintivo y conceptos, tecnologías y marcos relacionados. [61], [63]

7.1.1 Elementos de Informática orientada a Servicios

La informática orientada a servicios (Service-oriented computing, SOC) es un paradigma informático que utiliza los servicios como elementos para apoyar el desarrollo rápido y de bajo costo de aplicaciones/soluciones distribuidas. La promesa de la informática orientada a servicios es un mundo de servicios cooperativos que se acoplan libremente para crear procesos comerciales dinámicos y aplicaciones ágiles que puede abarcar organizaciones y plataformas informáticas, y que puede adaptarse de forma rápida y autónoma a requisitos cambiantes. ([51], [58], [60], [27])

Para construir el modelo de servicio, informática orientada a servicios se basa en la arquitectura orientada a servicios (SOA), que es una forma de reorganizar las aplicaciones de software y la infraestructura en un conjunto de servicios que interactúan.

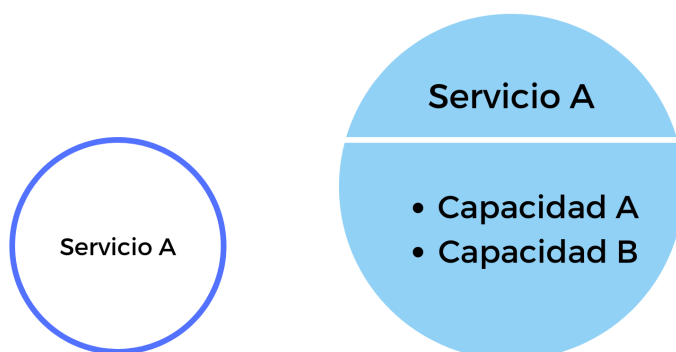


Figure 7.1: Servicios y sus capacidades

Elementos del paradigma de la informática orientada a servicios

Elementos relacionados con el paradigma de la informática orientada a servicios:

- SOA,
- servicios,
- orquestación de servicios y
- coordinación de transacciones de servicio

Arquitectura orientada a Servicios Arquitectura orientada a Servicios establece un modelo arquitectónico orientado a mejorar la eficiencia, la agilidad y la productividad de una empresa mediante el uso de la informática orientada a servicios. La plataforma informática orientada a servicios gira en torno al paradigma de diseño de orientación de servicio y su relación con SOA. Una implementación de SOA puede consistir en una combinación de tecnologías, productos, API, extensiones de infraestructura de soporte y otros elementos.

Servicios y orientación a servicios La orientación a servicios es un paradigma de diseño compuesto por un conjunto específico de principios de diseño. La aplicación de estos principios al diseño de solución lógica da como resultado una solución orientada al servicio. La orientación a servicio usa la separación de problemas y da forma a la solución. Aplicar la orientación a servicios da como resultado una solución que se puede clasificar de forma segura como **orientada a servicios** y unidades que se califican como **servicios**.

El **servicio** es la unidad fundamental de la solución orientada a servicios. Cada servicio tiene su propio contexto funcional y capacidades relacionadas con este contexto. En la Figura 7.1 esta ilustrado la notación usada para modelar un servicio (izquierda) y de describir un servicio con sus capacidades (derecha).

En [40], un servicio es recurso abstracto que representa una capacidad de realizar tareas que constituyen una funcionalidad coherente desde el punto de vista de los proveedores de las entidades y de las entidades solicitantes.

Aquellas capacidades adecuadas para la invocación de programas de consumo externos se expresan a través de un **contrato de servicio** publicado. Un contrato de servicio establece las cualidades que son (o deberían ser) visibles para los negocios entre los



Figure 7.2: Contrato de Servicios.

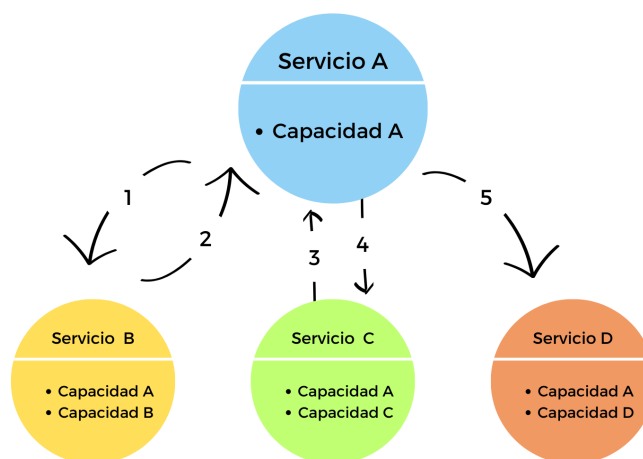


Figure 7.3: Composición de Servicios. Adaptado de [60]

participantes del servicio, en lugar de modelar interacciones en un nivel detallado. Las partes de un contrato de servicios se ilustran en la Figura 7.2.

Composición de Servicios Una composición de servicios es un agregado coordinado de servicios. Es comparable a una aplicación tradicional, ya que su alcance funcional suele asociarse con la automatización de un proceso empresarial principal. Un ejemplo de modelado de composiciones de servicios esta en la Figura 7.3. Es una composición de servicio compuesta por cuatro servicios. Las flechas indican una secuencia de modelados de intercambios de mensajes. La flecha 5 representa una entrega de datos asíncrona unidireccional del Servicio A al Servicio D.

Inventario de Servicios un inventario de servicios es una colección de servicios que representa una empresa o un segmento de ella, ver Figura 7.4. Son creados a través de procesos top-down que resultan en la definición del inventario de servicio. Los principios de diseño y los estándares de diseño aplicados al inventario establece el grado de interoperabilidad interservicios.

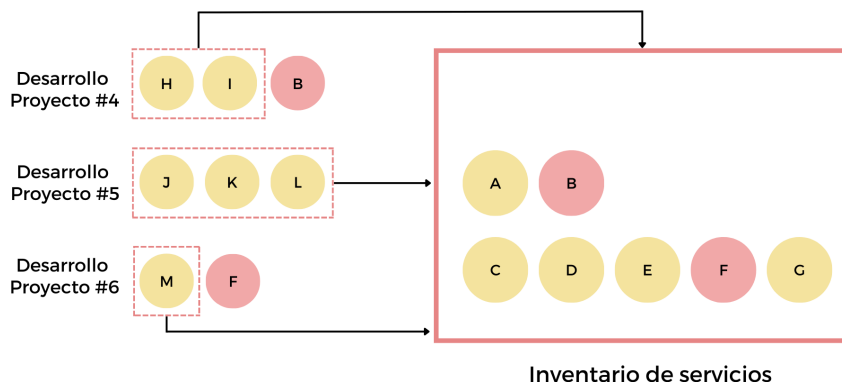


Figure 7.4: Inventario de Servicios

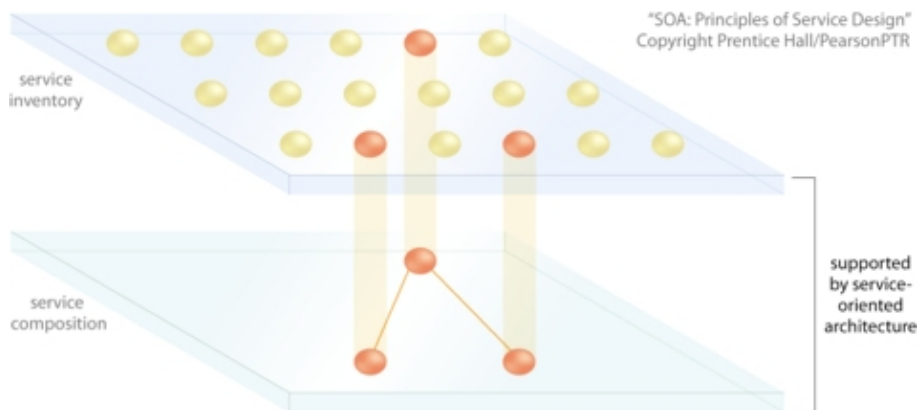


Figure 7.5: Vista Física de Servicios. Tomado de [60]

Vista Física de Servicios La solución orientada al servicio se implementa como servicios y servicio compuestos diseñadas de acuerdo con principios de diseño de orientación de servicio, Figura 7.5. Un servicio puede ser invocado por múltiples programas de consumidores. Los servicios compuestos puede formar la base de un inventario de servicios que se administran independientemente de su propio entorno de despliegue físico. Se pueden automatizar múltiples procesos comerciales mediante los servicios compuestos. Debido a que los servicios son reutilizables, no pertenecen a ninguna aplicación.

7.1.2 Principios de diseño de orientación de servicio

Contrato de servicio estandarizado Los servicios expresan su propósito y capacidades en un contrato de servicio. El principio de diseño del contrato requiere consideraciones de diseño de la interfaz del servicio. Diseño incluye la manera en que los servicios expresan la funcionalidad, definición de tipos y modelos de datos, las políticas, etc.

Acoplamiento El acoplamiento es una conexión o relación entre dos cosas. Comparable a un nivel de dependencia. La orientación al servicio aboga por la creación de un tipo específico de relación dentro y fuera de los límites del servicio, que reduzca las dependencias entre el contrato, implementación y consumidores del servicio. Acoplamiento débil o

debilmente acoplado promueve el diseño independiente, evolución e implementación de un servicio, al tiempo que garantiza la interoperabilidad con los consumidores del servicio.

Abstracción Este principio enfatiza la necesidad de ocultar la mayor cantidad de detalles subyacentes de un servicio como sea posible. Es importante para habilitar y preservar la relación débilmente acoplada y, también en el posicionamiento y diseño de composiciones de servicio. Los metadatos son relevantes cuando se evalúan los niveles de abstracción apropiados.

Reutilización de servicio La reutilización se enfatiza fuertemente dentro de la orientación del servicio; es una parte central de los procesos de análisis y diseño de servicios, y también forma la base para los modelos claves de servicio. El principio de la reutilización del servicio enfatiza el posicionamiento de los servicios como recursos empresariales con contextos funcionales agnósticos.

Autonomía del servicio La autonomía es necesaria para que los servicios puedan llevar a cabo sus capacidades de manera consistente y confiable. Es necesaria la aplicación de principios de diseño para aumentar la confiabilidad y predictibilidad del comportamiento de un servicio. Se considera niveles de aislamiento y consideraciones de normalización del servicio para lograr una medida adecuada de autonomía.

Servicio sin estado El manejo de información con estado puede comprometer la disponibilidad y escalabilidad de un servicio. Los servicios se diseñan para permanecer con estado solo cuando sea necesario. El principio de servicios sin estado requiere que se evalúen la idoneidad de la arquitectura tecnológica para proporcionar las opciones de delegación y diferimiento de la administración de los estados.

Descubrimiento de servicio Para que los servicios se posicionen como activos, deben identificarse y comprenderse cuando se presenten oportunidades de reutilización. El diseño del servicio debe tener en cuenta la *calidad de las comunicaciones* del servicio y sus capacidades individuales.

7.2 Servicios Web

De acuerdo a [40], un servicio web es un sistema software diseñado para soportar la interacción máquina-a-máquina, a través de una red, de forma interoperable. Cuenta con una interfaz descrita en un formato procesable por un equipo informático (específicamente en **WSDL**), a través de la que es posible interactuar con el mismo mediante el intercambio de mensajes **SOAP**, típicamente transmitidos usando serialización XML sobre HTTP conjuntamente con otros estándares web.

La Figura 7.6 indica los puntos principales sobre la arquitectura de comunicación en la que operan los servicios web: un servicio web se identifica mediante un URI y los clientes pueden acceder a él mediante mensajes formateados en XML [54]. SOAP se utiliza para encapsular estos mensajes y transmitirlos a través de HTTP u otro protocolo, por ejemplo, TCP o SMTP. Un servicio web implementa descripciones de servicios para especificar la interfaz y otros aspectos del servicio en beneficio de los clientes potenciales.

Las características distintivas de los servicios web se listan a continuación:

- Uso de SOAP
- Uso de XML: la representación del mensaje en XML para protocolos de comunicación y datos. El formato XML es fácil de leer y entender, lo que facilita la creación y el mantenimiento de archivos XML. Su desventaja es el espacio que ocupa la representación textual y el tiempo de procesamiento (parse).



Figure 7.6: Infraestructura de Servicios Web.

- Combinación con otros servicios web para construir otras funcionalidades
- Patrones de comunicación: RR síncronos RR asíncronos, combinaciones de ellos.
- Acoplamiento Débil: minimizar las dependencias entre servicios para tener una arquitectura flexible
- Uso de interfaz de servicio web
- Tendencia a la simplicidad: ejm, REST usa interfaces mínimas (GOOGLE)
- Uso de múltiples paradigmas de comunicación (RR, Asíncronos, indirectos). Esto afecta el nivel de acoplamiento.

7.2.1 SOAP

SOAP (SOAP, Service Oriented Architecture Protocol) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML [40].

Su propósito es establecer una interacción asíncrona entre el cliente y el servidor. Define un esquema con XML, donde se representa el contenido de una solicitud – respuesta de un mensaje. Se usa con protocolos HTTP, SMTP, TCP, UDP. La estructura de un mensaje SOAP se muestra en el listado 7.1:

Algoritmo 7.1: Estructura de mensaje SOAP

```

<soap:Envelope>
  <soap:Header>
    <header element>
    </header element>
  </soap:Header>
  <soap:Body>
    <body element>
    </body element>
  </soap:Body>
</soap:Envelope>

```

Un mensaje SOAP puede ser un documento o un soporte a la comunicación cliente-

servidor:

- Un documento se coloca directamente en el interior del elemento de cuerpo junto con una referencia a un esquema XML que contiene la descripción del servicio (nombres y tipos utilizados en el documento).
- Para la comunicación cliente-servidor, el elemento del cuerpo *body* contiene ya sea una solicitud (*Request*), ver listado 7.2 o una respuesta (*Replay*), ejemplo en el listado 7.3.

Algoritmo 7.2: Solicitud en mensaje SOAP. Tomado de [40]

```
<soap:Body>
  <getProductDetails xmlns="http://warehouse.example.com"
    <productId>827635</productId>
  </getProductDetails>
</soap:Body>
</soap:Envelope>
```

Algoritmo 7.3: Respuesta en mensaje SOAP. Tomado de [40]

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com"
      <getProductDetailsResult>
        <productName>Toptimate 3-Piece
        <productId>827635</productId>
        <description>3-Piece luggage set
        Black Polyester.</description>
        <price>96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP: Protocolo de Transporte del mensaje Se requiere un protocolo de transporte para enviar un mensaje SOAP a su destino. Los mensajes SOAP son independientes del tipo de transporte utilizado. HTTP se usa para especificar la dirección de destino del mensaje. De igual manera, se usa HTTP para retornar el contenido de un respuesta a una solicitud SOAP. En el listado 7.4: las líneas de la 1 a la 4 es el encabezado de HTTP, a partir de la 4 se detalla el mensaje SOAP. [54]

Algoritmo 7.4: Protocolo de Transporte en SOAP. Tomado de [54]

```
POST /examples/stringer
Host: www.cdk4.net
Content-Type: application/soap+xml
Action: http://www.cdk4.net/examples/stringer#exchange

<env:envelope xmlns:env = namespace URI for SOAP envelope
  <env:header> </env:header>
  <env:body> </env:body>
</env:Envelope>
```



Figure 7.7: Servicio Web con SOAP y HTTP

En cuanto a la confiabilidad en la comunicación del mensaje usando SOAP, debido a que SOA usa HTTP sobre TCP por lo que esta afectado por el modelo de fallas de TCP ya descrito en capítulo 4

Búsqueda de un servicio en la web En la Figura 7.7 se muestra cómo es la búsqueda de un servicio en la web y los actores involucrados en ello:

1. ¿ Donde consigo un servicio X? la pregunta se hace al directorio de servicios, el **UDDI**
2. El **UDDI** responde: el Servidor A es capaz de hacer X.
3. ¿ Cómo invoco su servicio?
4. Mire el **WSDL** que esta disponible
5. Hago la solicitud X usando el protocolo SOAP
6. Hago la solicitud de la operacion X usando el protocolo **SOAP**

7.2.2 Descripción de Servicios e Interfaces Web

La interfaz de servicio web se hacen necesarias para permitir la comunicación de los clientes con los servicios. La definición de las interfaces son proporcionadas por la descripción de servicios

La descripción de Servicios especifica como los mensajes son comunicados y la ubicación de ese servicio o el URI del servicio. Es un acuerdo entre cliente y servidor de como se ofrece un servicio. Se usa para generar procedimientos (stub) del cliente que implementa el comportamiento del cliente

7.2.3 Lenguaje de Descripción del Servicio

El Lenguaje para Descripción de Servicios Web (WSDL, Web Service Description Language) se usa para describir los servicios. WSLD define, usando esquema XML, la representación de los componentes de la descripción de servicios. De igual manera, contempla la definición de nombre de elementos, tipos, mensajes, interfaz, enlaces y servicios.

Los elementos en la descripción del servicio WSDL

- **Abstracta:** conjunto de definiciones de los tipos usados por el servicio

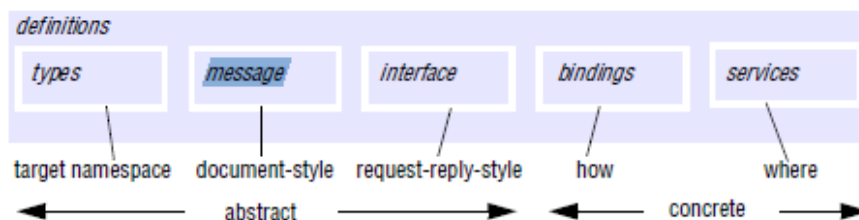


Figure 7.8: IDL

- Tipo: `<element name="isFilled" type="boolean"/>`
- Mensaje: descripción del mensaje que se intercambiará. También describe los tipos de mensajes
- **Concreta:** enlace y servicio, ambas dependen del protocolo que se use en SOAP.

7.2.4 Descripción de los elementos de WSDL

Elemento *Definitions* Todas las partes de las descripciones abstractas y concretas se alojan dentro de un elemento raíz que establece un constructo padre llamado *Definitions*. Por lo tanto, lo que hay entre los elementos *Definitions* representa el ámbito de un contrato de servicio Web de un servicio Web, ver listado 7.5.

Algoritmo 7.5: Elemento *Definitions*. Tomado de [40]

```
<definitions name="OrdenCompra" targetNamespace=
    ...
</definitions>
```

Este elemento tiene dos atributos opcionales: *name* y *target-Namespace*. Pueden añadirse atributos adicionales como los atributos *xmlns* utilizados para establecer un rango de prefijos para los espacios de nombres existentes relevantes para este contrato. Estos prefijos se utilizan para identificar y distinguir elementos de distintos orígenes (con diferentes espacios de nombres) que residen en la misma definición WSDL.

Atributo *targetNamespace* El atributo *targetNamespace* establece el valor del espacio de nombres asociado a todos los elementos con nombre definidos en el documento WSDL. Por ejemplo, el nombre de un elemento *portType* definido en la definición WSDL está automáticamente asociado con este espacio de nombres de destino, y por lo tanto será distinguible de nombres idénticos en diferentes definiciones WSDL.

Atributo *xmlns* Al igual que otro elemento XML, el elemento *definitions* puede contener múltiples variaciones del atributo *xmlns* para establecer una serie de prefijos de espacio de nombres. Estos prefijos son necesarios cuando se necesita mezclar elementos de diferentes orígenes en el mismo documento. Una definición WSDL básicamente siempre acabará conteniendo elementos de lenguajes distintos de WSDL (como SOAP y XML Schema) y definiciones diferentes (como tipos definidos en otros documentos de esquema XML).

Algoritmo 7.6: Atributo *xmlns*. Tomado de [60]

```
<definitions name="OrdenCompra" targetNamespace=
    "http://prueba.com/contract/po"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://prueba.com/contract/po"
```

```

xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
xmlns:po="http://prueba.com/schema/po"
xmlns:soap11="http://schemas.xmlsoap.org/ws"
xmlns:soap12="http://schemas.xmlsoap.org/ws"

```

Por ejemplo, los atributos *xmlns* que pueden ser necesarios:

- El atributo *targetNamespace* se establece en <http://prueba.com/contract/po>, lo que significa que todos los elementos con nombre en la definición **WSDL** pertenecerán a este espacio de nombres
- El espacio de nombres predeterminado se establece en el espacio de nombres **WSDL**, lo que indica que el espacio de los elementos estándar WSDL (tipos, portType, mensaje, etc.) no requerirán ningún prefijo cuando se utiliza en el documento, como se muestra a continuación:

```

<message name="msgOrdenCompraRequest">
    <part name="OrdenCompra" element="OrdenCompra" type="tns:OrdenCompra" />
</message>

```

- El prefijo *tns*: está asociado con <http://prueba.com/contract/po>, que también es el espacio de nombres de destino. Esto permite referirse a elementos que pertenecen a la espacio de nombres de destino en el documento WSDL a través de este prefijo.
- El prefijo *po*: permite referirse a elementos del esquema XML que fueron declarados en el espacio de nombre <http://prueba.com/schema/po>, como por ejemplo:

```

<message name="msgEnviarOrdenRequest">
    <part name="OrdenCompra" element="OrdenCompra" type="po:OrdenCompra" />
</message>

```

- Los prefijos *soap11* y *soap12* están vinculados al espacio de nombres estándar para SOAP 1.1 y SOAP 1.2, respectivamente y se utilizan exclusivamente dentro de la descripción **concreta** referida en la Figura 7.8.

Elemento *Documentation* Cualquier parte de una definición WSDL se puede anotar con comentarios legibles por humanos a través del uso del elemento de *documentation*. Cualquier parte o elemento WSDL puede ser documentado, y su contenido puede ser texto arbitrario o incluso otros elementos XML.

```

<documentation>
    Esta es una entidad de servicio responsable
</documentation>

```

Estructura del mensaje WSDL En el nivel abstracto, un servicio web se define en términos de su interfaz pública. La descripción abstracta establece esta interfaz a través de un conjunto de elementos relacionados. La estructura de un mensaje WSDL se ve en el listado 7.7

Algoritmo 7.7: Estructura de mensaje WSDL

```

<definitions>
    <types>
        data type definitions .....
    </types>

```

```

<message>
    definition of the data being communicated ....
</message>
<portType>
    set of operations .....
</portType>
<binding>
    protocol and data format specification ....
</binding>
</definitions>

```

La estructura de la descripción abstracta ([40] [63]):

Elemento <types> Define los tipos de datos (esquema XML) utilizados por el servicio web. Un servicio web necesita definir sus entradas y salidas y cómo se asignan dentro y fuera de los servicios. El elemento <types> se encarga de definir los tipos de datos que utiliza el servicio web. Los tipos son documentos XML o partes de documentos.

El elemento de tipos describe todos los tipos de datos utilizados entre el cliente y el servidor.

WSDL no está ligado exclusivamente a un sistema de escritura específico.

WSDL utiliza la especificación de esquema XML W3C como opción predeterminada para definir los tipos de datos.

Si el servicio utiliza solo tipos simples integrados en el esquema XML, como cadenas y números enteros, no se requiere el elemento de tipos.

WSDL permite que los tipos se definan en elementos separados para que los tipos se puedan reutilizar con múltiples servicios web.

Algoritmo 7.8: Descripción de elemento <types>. Tomado de [40]

```

<types>
    <schema targetNamespace = "http://example.com"
        xmlns = "http://www.w3.org/2000/10/XMLSchema"
            <element name = "TradePriceRequest"
                <complexType>
                    <all>
                        <element name = "ticker" type = "string" />
                    </all>
                </complexType>
            </element>

            <element name = "TradePrice">
                <complexType>
                    <all>
                        <element name = "price" type = "float" />
                    </all>
                </complexType>
            </element>
        </schema>

```

</types>

Elemento <message> El elemento <message> describe los datos que se intercambian entre los proveedores de servicios web y los consumidores.

Cada Servicio Web tiene dos mensajes: entrada y salida.

La entrada describe los parámetros del servicio web y la salida describe los datos de retorno del servicio web.

Cada mensaje contiene cero o más parámetros <part>, uno para cada parámetro de la función del servicio web. Cada parámetro <part> se asocia con un tipo concreto definido en el elemento contenedor <types>.

Por ejemplo, en el listado 7.9 se definen dos elementos de mensaje: un mensaje de solicitud *HelloRequest* y el segundo representa un mensaje de respuesta *HelloResponse*.

Algoritmo 7.9: Descripción del elemento <message>. Tomado de [40]

```
<message name = "HelloRequest">
  <part name = "firstName" type = "xsd:string"/>
</message>

<message name = "HelloResponse">
  <part name = "greeting" type = "xsd:string"/>
</message>
```

Elemento <portType> El elemento <portType> describe las operaciones que se pueden realizar y los mensajes implicados. Define un servicio web, las operaciones que se pueden realizar y los mensajes involucrados. En el tipo de operación solicitud-respuesta, **WSDL** define cuatro tipos que se detallan en la tabla 7.1.

One-Way	La operación puede recibir un mensaje pero no devolverá una respuesta
Request-Response	La operación puede recibir una solicitud y devolverá una respuesta.
Solicit-Response	La operación puede enviar una solicitud y esperará una respuesta.
Notification	La operación puede enviar un mensaje pero no esperará una respuesta

Table 7.1: Operación Solicitud-Respuesta en WSDL. Tomado de [40]

Elemento <binding> Los enlaces **WSDL** definen el formato del mensaje y los detalles del protocolo para cada tipo de puerto de un servicio web. El elemento <binding> tiene dos atributos: nombre y tipo.

El atributo de nombre define el nombre del enlace y el atributo de tipo apunta al puerto para el enlace, en el código 7.10 es el puerto "glossaryTerms".

El elemento *soap:binding* tiene dos atributos: estilo y transporte. El atributo de estilo puede ser "rpc" o "document". En el código 7.10 se usa documento. El atributo de transporte define el protocolo **SOAP** a utilizar. En el listado 7.10 se usa HTTP.

El elemento de operación define cada operación que expone *portType*.

Para cada operación se debe definir la acción SOAP correspondiente. También debe especificar cómo se codifican la entrada y la salida. En el ejemplo 7.10 se usa "literal".

Algoritmo 7.10: Binding. Tomado de [40]

```

<message name="getTermRequest">
    <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
    <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
    <operation name="getTerm">
        <input message="getTermRequest"/>
        <output message="getTermResponse"/>
    </operation>
</portType>

<binding type="glossaryTerms" name="b1">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation>
        <soap:operation soapAction="http://schemas.xmlsoap.org/soap/envelope/getTerm"/>
        <input><soap:body use="literal"/></input>
        <output><soap:body use="literal"/></output>
    </operation>
</binding>

```

Servicio El elemento `<service>` define los puertos admitidos por el servicio web. Para cada uno de los protocolos compatibles, hay un puerto. El elemento servicio es una colección de puertos.

Los elementos del servicio proporcionan información al cliente: dónde acceder al servicio, por cuál puerto se accede al servicio web, y cómo se definen los mensajes de comunicación.

El elemento de servicio incluye un elemento de documentación para proporcionar documentación legible por humanos. En el listado 7.11 es un extrato con la definición de servicios.

Algoritmo 7.11: Definición de Servicio en WSDL. Tomado de [40]

```

<service name = "Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding = "tns:Hello_Binding" name = "Hello_Port">
        <soap:address
            location = "http://www.examples.com/SayHello/">
        </port>
    </service>

```

Los atributos *binding* del elemento *port* asocian la dirección del servicio con un elemento vinculante definido en el servicio web; como en el ejemplo 7.12

Algoritmo 7.12: Enlaces y Puertos en la definición de servicios. Tomado de [40]

```
<binding name = " Hello_Binding" type = "tns:Hello_PortType">
  <soap:binding style = "rpc"
    transport = "http://schemas.xmlsoap.org/soap/http"/>
    <operation name = "sayHello">
      <soap:operation soapAction = "sayHello"/>

      <input>
      <soap:body
        encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
        namespace = "urn:examples:helloservice" use = "literal"/>
      </input>

      <output>
      <soap:body
        encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
        namespace = "urn:examples:helloservice" use = "literal"/>
      </output>
    </operation>
  </binding>
```

Ejemplo de un archivo WSDL En el listado 7.13 se muestra un ejemplo de un archivo WSDL con las partes detalladas anteriormente.

Algoritmo 7.13: Ejemplo de un mensaje WSDL. Tomado de [40]

```
<definitions name = "HelloService"
  targetNamespace = "http://www.examples.com/wsdl/HelloService/"
  xmlns = "http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns = "http://www.examples.com/wsdl/HelloService/"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

  <message name = "HelloRequest">
    <part name = "firstName" type = "xsd:string"/>
  </message>

  <message name = "HelloResponse">
    <part name = "greeting" type = "xsd:string"/>
  </message>

  <portType name = "Hello_PortType">
    <operation name = "sayHello">
      <input message = "tns:HelloRequest"/>
      <output message = "tns:HelloResponse"/>
    </operation>
  </portType>

  <binding name = "Hello_Binding" type = "tns:Hello_PortType">
```

```

        <soap:binding style = "rpc"
            transport = "http://schemas.xmlsoap.org
            <operation name = "sayHello">
        <soap:operation soapAction = "sayHello"/>
            <input>
                <soap:body
                    encodingStyle = "http://schemas
                    namespace = "urn:examples:hello
                    use = "encoded"/>
            </input>

            <output>
                <soap:body
                    encodingStyle = "http://schemas
                    namespace = "urn:examples:hello
                    use = "encoded"/>
            </output>

        </operation>
    </binding>

    <service name = "Hello_Service">
        <documentation>WSDL File for HelloService </documentation>
        <port binding = "tns:Hello_Binding" name = "Hello_Port">
            <soap:address
                location = "http://www.examples.com/Say
            </port>
        </service>
    </definitions>

```

7.2.5 Universal Description, Discovery and Integration service

La especificación UDDI (Universal Description, Discovery, and Integration) define un modo de publicar y encontrar información sobre servicios Web. UDDI tiene dos funciones: es un protocolo basado en SOAP que define cómo se comunican los clientes UDDI con registros y es un conjunto en particular de registros duplicados globalmente.

En el registro de un servicio intervienen cuatro tipos de estructuras de datos principales, ver la Figura 7.9

- El tipo de datos *businessEntity* contiene información sobre la empresa que tiene un servicio publicado.
- El tipo de datos *businessService* es una descripción de un servicio Web.
- El tipo de datos *bindingTemplate* contiene información técnica para determinar el punto de entrada y especificaciones de construcción para invocar un servicio Web.
- El tipo de datos *tModel* proporciona un sistema de referencia que ayuda a descubrir servicios Web y actúa como una especificación técnica de un servicio Web.

UDDI proporciona un API para la ubicación del servicio:

- *get xxx get – businessDetail, get – ServiceDetail...* para la obtención de un registro en particular.
- *find xxx find – business, find – Service...* usado para la búsqueda del servicio en

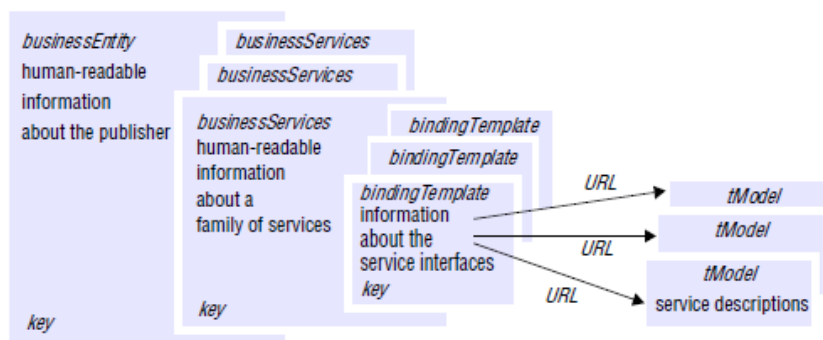


Figure 7.9: Directorio de Servicios

el UDDI.

El servidor posee un URI para identificar el API del UDDI.

7.2.6 Seguridad XML

Seguridad XML es un conjunto de diseños de la W3C ([40]) para registro, manejo de claves y encriptación. Se relaciona con el servicio web en el documento **SOAP** donde se describe la integridad, confidencialidad y autenticación del mensaje. En el documento **SOAP** se definen las etiquetas que indican cuales secciones del documento están encriptadas o seguras: el documento SOA puede encriptarse en su totalidad o parte de ello y puede protegerse el acceso al documento o parte de ello.

La seguridad del servicio web describe tres mecanismos principales:

- Cómo firmar mensajes SOAP para asegurar la integridad.
- Cómo encriptar mensajes SOAP para asegurar la confidencialidad.
- Cómo adjuntar tokens de seguridad para determinar la identidad del remitente.

La especificación de la W3C ([40]) permite una variedad de formatos de firma del documento, algoritmos de cifrado y múltiples dominios de confianza, y está abierta a varios modelos de algoritmos de token de seguridad, como:

- certificados X.509
- tickets de Kerberos,
- ID de usuario/credenciales de contraseña,
- Autenticación SAML (SAML, Security Assertion Markup Language), y
- tokens personalizados.

Entre los requerimientos para la especificación del algoritmo para la seguridad del servicio web:

- Se debe especificar un algoritmo adecuado para la implementación de la seguridad XML.
- El algoritmo usado para encriptación y autenticación de un documento debe ser seleccionado de un conjunto y los nombres de los algoritmos deben ser referenciados dentro del documento XML.
- Los documentos XML seguros se firman y/o se encriptan mucho antes de cualquier consideración sobre quién los accederá.
- El estándar especifica un conjunto de algoritmos que se proporcionarán en cualquier implementación de seguridad XML: Al menos un algoritmo de cifrado y uno de

firma.

- Los algoritmos utilizados deben seleccionados de ese conjunto y los nombres de los algoritmos en uso se deben referenciar dentro del documento XML.
- La seguridad XML define los nombres de los elementos que se pueden usar para especificar el URI del algoritmo en uso para la firma o el cifrado.

7.2.7 Coordinación

La infraestructura SOAP admite interacciones de solicitud-respuesta entre clientes y servicios web. Las aplicaciones generan solicitudes que deben realizarse en un orden particular. Ejemplo, al reservar un vuelo, la información de precio y disponibilidad se recopila antes de realizar las reservas.

Cuando un usuario interactúa con páginas web en un navegador, para reservar un vuelo o hacer una oferta en una subasta, la interfaz proporcionada por el navegador controla la secuencia en el que se realizan las operaciones. En un servicio web que realiza reservas, el servicio web debe trabajar a partir de una descripción de la forma apropiada de proceder cuando se interactúa con otros servicios para, por ejemplo, alquiler de coches y reservas de hotel como así como reservas de vuelos

El propósito de la coordinación de servicios web:

- para generar esquemas de código para un nuevo servicio que quiera participar;
- como base para generar mensajes de prueba para un nuevo servicio;
- promover un entendimiento común de la colaboración entre servicios;
- analizar la colaboración, por ejemplo, para identificar posibles situaciones de punto muerto

El estándar W3C ([40]) establece los siguientes aspectos relacionados con la coordinación de servicios web:

- composición jerárquica y recursiva de coreografías;
- capacidad de agregar nuevas instancias de un servicio existente y nuevos servicios;
- caminos concurrentes, caminos alternativos y la capacidad de repetir una sección de un coreografía;
- tiempos de espera variables, por ejemplo, diferentes períodos para mantener reservas;
- excepciones, por ejemplo, para tratar con mensajes que llegan fuera de secuencia y de usuario acciones tales como cancelaciones;
- interacciones asíncronas (devoluciones de llamada);
- paso por referencia, por ejemplo, para permitir que una empresa de alquiler de coches consulte a un banco para una verificación de crédito en nombre de un usuario;
- marcado (*timestamp*) de las transacciones que tienen lugar, por ejemplo, para permitir la recuperación;
- la capacidad de incluir documentación legible por humanos.

7.3 REST

REST: Representational State Transfer La transferencia de estado representacional (Rest, representational state transfer) es un estilo de arquitectura de software para sistemas hipertexto distribuidos como la World Wide Web. El término se originó en el año 2000, en una tesis doctoral sobre la web escrita por Roy Fielding, uno de los principales autores de la especificación del protocolo HTTP y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo de software. [65], [62] [58] [48]

REST establece un conjunto de principios para la arquitectura de servicios:

- Clientes usan URL y HTTP (GET, PUT, DELETE, POST)
- Recursos son representados en XML
- Énfasis en manipulación de recursos de datos sobre interfaces

Agente El agente es un programa dirigido por eventos que no proporciona una interfaz técnica publicada. Está diseñado para actuar como un intermediario capaz de interceptar mensajes en tiempo de ejecución. Cuando se intercepta un mensaje, el servicio agente puede realizar un procesamiento activo o pasivo sobre el mensaje. La lógica de procesamiento del servicio agente se considera activa cuando termina alterando el contenido del mensaje, mientras que la lógica de procesamiento pasivo no lo hace.

7.3.1 Restricciones de servicios REST

A continuación se indican las características y restricciones que debe cumplir una arquitectura en REST:

Cliente - Servidor

- Requiere que un servicio ofrezca una o más capacidades y escuche las solicitudes sobre estas capacidades.
- Un consumidor invoca una capacidad enviando el mensaje de solicitud correspondiente, y el servicio rechaza la solicitud o realiza la tarea solicitada antes de enviar un mensaje de respuesta al consumidor.
- Las excepciones que impiden que la tarea continúe son elevadas al consumidor, y el consumidor es responsable de tomar medidas correctivas.
- La solución debe someterse a un proceso por el cual está sujeta la separación de solicitudes.
- Esto divide la solución en unidades que abordan solicitudes definidas. Estas unidades se componen para formar la solución en tiempo de ejecución.
- El conocimiento requerido del consumidor sobre un servicio y el conocimiento requerido del servicio de sus consumidores se limitan a los contenidos del contrato técnico compartido.

Sin estado

- Cada solicitud debe contener toda la información necesaria para que el servicio entienda el significado de la solicitud, y todos los datos de estado de la sesión deben devolverse al consumidor del servicio una vez finalizada cada solicitud.
- La lógica del consumidor debe estar diseñada para preservar los datos del estado entre solicitudes y emitir solicitudes que contengan datos de estado.
- La solicitud debe contener todos los datos de estado necesarios para que el servicio la procese, y el servicio debe ser capaz de "olvidar" los datos de estado al emitir la respuesta sin comprometer la interacción general.
- Las solicitudes intermedias indican que el servicio está *en reposo* y, usa menos recursos de CPU, memoria o red.
- El servicio no almacena datos en una instancia de tiempo de ejecución de un consumidor de servicios. Puede almacenar datos de su propio contexto funcional.

Cache

- Los mensajes de respuesta del servicio a sus consumidores se etiquetan como almacenados o no en caché.
- Así, el servicio, el consumidor o uno de los componentes del middleware pueden almacenar en caché la respuesta para su reutilización en solicitudes posteriores.

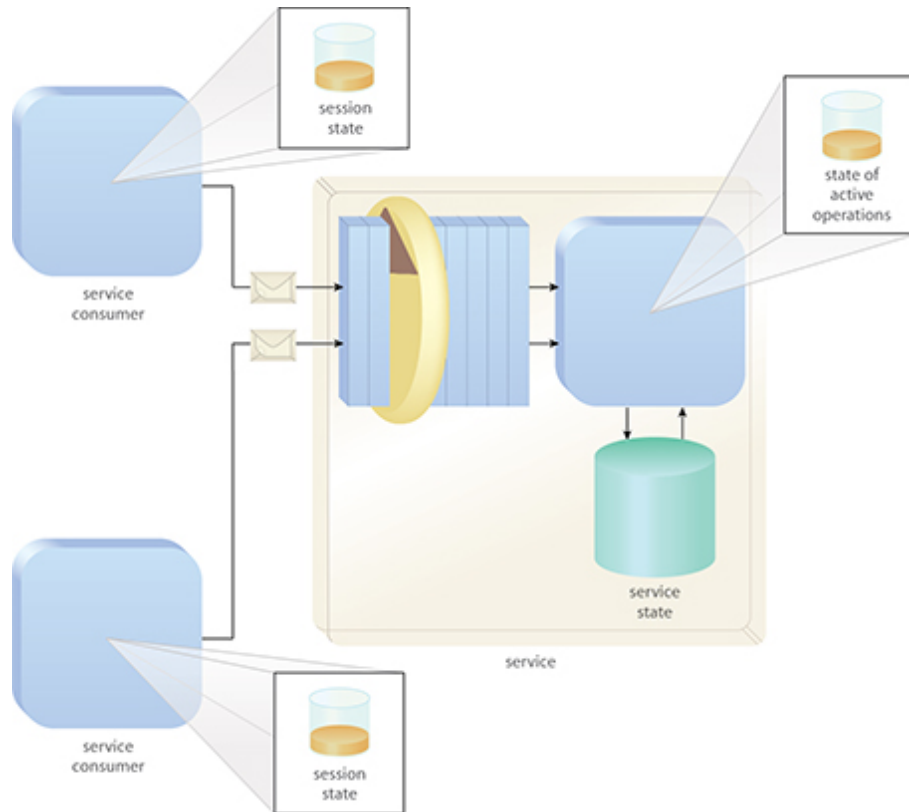


Figure 7.10: Servidores sin estado

- Los servicios están diseñados para producir metadatos para control de caché y devolverlos en mensajes de respuesta.
- Un repositorio de caché intermediario o del lado del consumidor permite al consumidor reutilizar los datos de respuesta almacenables en caché para los mensajes de solicitud posteriores.
- Los mensajes de solicitud deben ser comparables para determinar si son equivalentes o no.
- Los contratos incluyen declaraciones explícitas sobre la capacidad de almacenamiento en caché de las respuestas, o permitir que los metadatos para control de caché se incluyan en las respuestas.

Interfaz

- Los consumidores acceden a las capacidades del servicio a través de métodos, tipos y una sintaxis común del identificador de recursos que están estandarizados en muchos consumidores y servicios.
- Para consumidores y servicios se establece un contrato uniforme con los tipos de métodos genéricos y reutilizables, y la sintaxis del identificador de recursos.
- El procesamiento de mensajes del consumidor está diseñado para estar estrechamente unido al contrato uniforme.
- El procesamiento de mensajes del consumidor está diseñado para ser desacoplada o poco unida a las capacidades y recursos específicos del servicio.
- Los recursos pueden proporcionar enlaces a otros recursos que el consumidor del servicio puede "descubrir" y, opcionalmente, acceder dinámicamente en tiempo de ejecución.

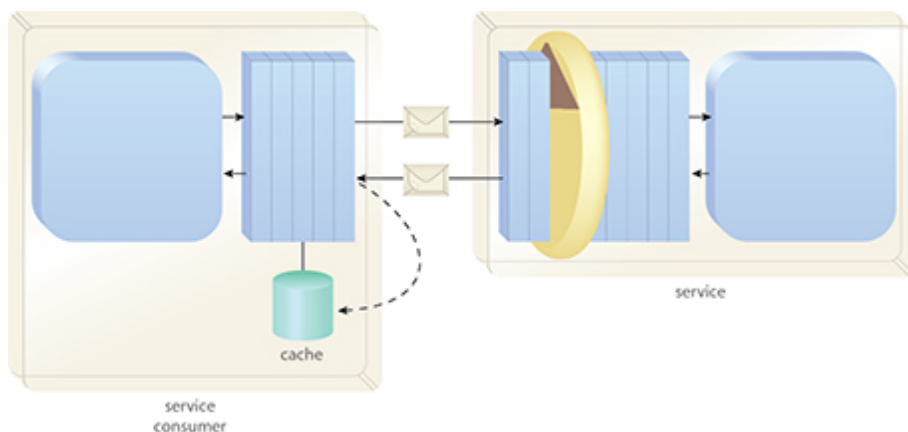


Figure 7.11: Cache. Tomado de [60]

- Los consumidores pueden "aprender" dinámicamente nuevos tipos de medios para procesar formatos de recursos previamente desconocidos.

Sistema de capas

- Las capas pueden estar compuestas por consumidores y servicios con contratos publicados o componentes de middleware basados en eventos (intermediarios) que establecen capas de procesamiento entre los consumidores y los servicios.
- Los consumidores están diseñados para invocar servicios sin el conocimiento de qué otros servicios también pueden invocarlos.
- Los intermediarios se agregan para realizar el procesamiento del mensaje en tiempo de ejecución sin conocimiento de cómo esos mensajes pueden procesarse más allá de la siguiente capa de procesamiento.
- La arquitectura de la solución está diseñada para permitir que se agreguen nuevas capas de middleware o eliminar capas antiguas de middleware sin cambiar el contrato técnico
- Los mensajes de solicitud/respuesta no deben revelar de qué capa proviene el mensaje a sus destinatarios.

Código-Por-Demanda

- Las arquitecturas de consumo de servicios incluyen un entorno de ejecución de la lógica proporcionada por un servicio. Esta lógica diferida se puede usar para extender la funcionalidad del consumidor o para especializarlo temporalmente.
- Los consumidores de servicios están diseñados para procesar la lógica descargada por los servicios en tiempo de ejecución.
- Los servicios toman decisiones explícitas sobre si ejecutarán la lógica ellos mismos o diferirán la ejecución de esa lógica a sus consumidores.

7.3.2 Metas de un servicios REST

Las metas de un servicio REST esta relacionada con lo siguiente:

Rendimiento La latencia de la red, el ancho de banda de la red limitado, y las redes no confiables pueden afectar el rendimiento al perder, reordenar o retrasar paquetes o mensajes y exigir que se vuelvan a intentar o reenviar

Escalabilidad La necesidad de que una arquitectura admita instancias o interacciones simultáneas. Se identifican cuatro enfoques:

- escalar, aumentar la capacidad de los servicios, los consumidores y los disposi-

tivos de red

- escalamiento, distribución de carga entre servicios y programas
- suavizar, igualar el número de interacciones durante los períodos pico y no pico para optimizar la infraestructura
- desacoplamiento del consumo de recursos finitos como la memoria de los consumidores concurrentes

Sencillez El objetivo de diseño sencillo se basa en la aplicación adecuada de la separación de solicitudes.

La sencillez es un foco para REST porque tiene un impacto en cómo los servicios se definen, descubren y finalmente se usan (o reutilizan) y además determina la facilidad con la que se pueden desarrollar los servicios de manera independiente. La principal contribución de REST a la sencillez es la restricción de contrato uniforme

Modificabilidad La facilidad para hacer cambios en una arquitectura. Se divide en las siguientes capacidades:

- evolución: refactorizar y volver a implementar componente del servicio, consumidor o middleware, sin afectar otras partes.
- extensibilidad: agregar funcionalidad a la arquitectura (incluso mientras se ejecutan las soluciones).
- personalización: modificar temporalmente partes de una solución para realizar tipos de tareas especiales.
- configurabilidad: modificar permanentemente partes de una arquitectura.
- reutilización: agregar nuevas soluciones a una arquitectura que reutilice servicios existentes, middleware, sin modificaciones.

Visibilidad Es la capacidad de las partes de una arquitectura para monitorear y regular la interacción entre otras partes de la misma arquitectura. Se traduce en el establecimiento de servicio agente basados en middleware que realizan un seguimiento de los mensajes transmitidos entre los servicios y los consumidores. El enfoque de estos servicio de agentes es mejorar el control administrativo sobre la arquitectura y optimizar su desempeño.

Portabilidad La facilidad con la que los servicios y las soluciones pueden trasladarse de una ubicación implementada a otra. Las consideraciones que se deben tener en cuenta incluyen el nivel de estandarización compatible en todos los entornos, la capacidad de mantener tanto los datos como la lógica agrupados, y qué tan portátil puede ser un determinado programa de software.

Confiabilidad Es el grado en que las soluciones y servicios (y la infraestructura subyacente) son susceptibles de fallar. La confiabilidad de una arquitectura se puede mejorar al evitar puntos únicos de falla, al utilizar mecanismos de conmutación por error y, funciones de monitoreo que pueden anticiparse y responder dinámicamente a las condiciones de falla.

7.3.3 Contratos de servicios no-Rest vs servicios REST:

Contratos de servicios no-Rest En la Figura 7.12 se muestra un esquema de como trabaja un contrato de servicio SOA usando como ejemplo un servicio de emisión de facturas::

- Emite un mensaje **SOAP** para solicitar datos de factura invocando *getInvoice*, predefinido en la definición **WSDL** contrato de servicio de Factura.
- Recibe datos solicitado de facturas en un mensaje de respuesta **SOAP** emitido por

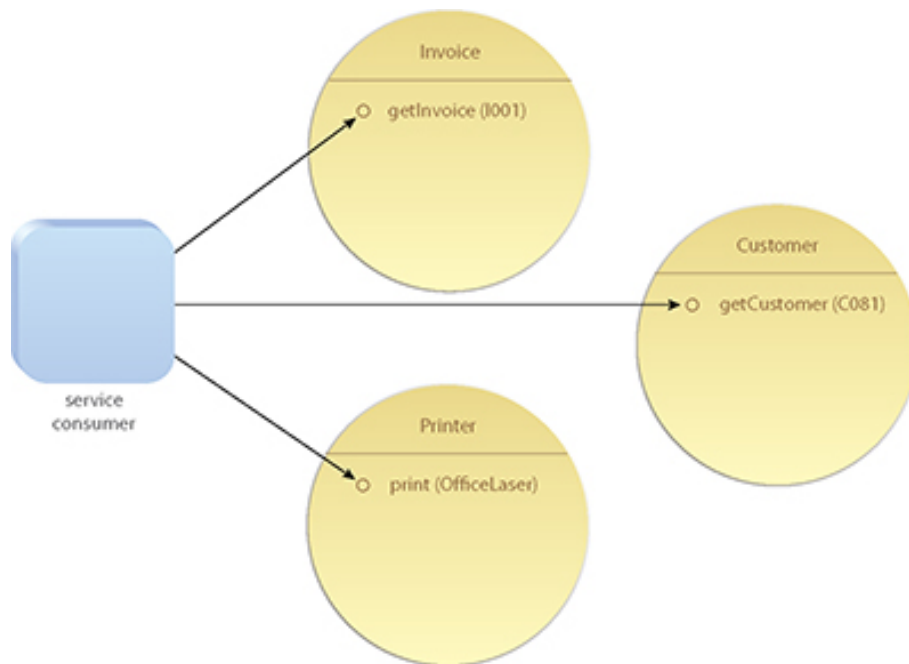


Figure 7.12: Contrato SOAP. Tomado de [60]

getInvoice del servicio de Factura.

- Emite un mensaje **SOAP** para solicitar datos del cliente invocando la capacidad *GetCustomer*, predefinido en la definición **WSDL** del contrato de servicio al cliente.
- Recibe los datos solicitados del cliente en un mensaje de respuesta SOAP emitido por la capacidad del servicio *GetCustomer* del servicio al cliente.
- Emite un mensaje **SOAP** solicitando que se agregue la dirección del cliente a la cola de impresión invocando la capacidad del servicio de impresión del servicio de impresora, como está predefinido en la definición WSDL del contrato de servicio de la impresora.
- Recibe un mensaje de respuesta **SOAP** que indica que la acción fue exitosa (o no).

En el listado 7.3.3 es un ejemplo de un mensaje SOAP simple que es enviado a un contrato de servicio basado en WSDL para solicitar una factura en función de su ID:

Solicitud en contrato de servicio SOAP

```

<soap:Envelope>
  <soap:Header>
    <wsa:To>
      http://invoice/
    </wsa:To>
  </soap:Header>
  <soap:Body>
    <getInvoice>
      <invoice-id>
        I123
      </invoice-id>
    </getInvoice>
  </soap:Body>

```

```
</soap:Envelope>
```

El ejemplo de mensaje de respuesta devuelto por el servicio web, en el listado 7.3.3, proporciona un mensaje de respuesta en SOAP que contiene el contenido de la factura.

Respuesta en contrato de servicio SOAP

```
<soap:Envelope>
  <soap:body>
    <getInvoiceResponse>
      <invoice>
        ... invoice content ...
      </invoice>
    </getInvoiceResponse>
  </soap:body>
</soap:Envelope>
```

Contratos de servicios Rest: En la Figura 7.13 se detallan las acciones relacionadas con el proceso de solicitud-respuesta del servicio de emisión de facturas usando un contrato REST:

- Emite una solicitud de datos de factura accediendo al servicio de Factura, utilizando un identificador de recursos y procesado mediante el método **HTTP GET**.
- Recibe los datos de factura solicitados en un mensaje de respuesta **HTTP** emitido por el servicio Factura.
- Emite una solicitud de datos de clientes accediendo al servicio de atención al cliente, utilizando un identificador de recursos y procesado mediante el método **HTTP GET**.
- Recibe los datos solicitados del cliente en un mensaje de respuesta **HTTP** emitido por el servicio al cliente.
- Emite una solicitud para que la dirección del cliente se agregue a la cola de impresión, accediendo al servicio de la impresora utilizando un identificador de recursos y procesado a través del método **HTTP POST**.
- Recibe una respuesta **HTTP** indicando que la acción fue exitosa (o no).

El intercambio con un servicio REST se basa en el uso de métodos HTTP, ejemplo en el listado 7.14:

Algoritmo 7.14: Solicitud en contrato de servicio REST

```
GET http://invoice/invoice/I123 HTTP/1.1
Accept: application/vnd.com.example.invoice+xml
```

El mensaje HTTP de respuesta, listado 7.15 que se devuelve al consumidor contiene un código de éxito o falla, el enunciado del encabezado del tipo de medio y el mismo fragmento XML que el mensaje de respuesta del servicio web.

Algoritmo 7.15: Respuesta en contrato de servicio REST

```
Content-Type: application/vnd.com.example.invoice+xml
<Invoice>
  ... invoice content ...
</Invoice>
```

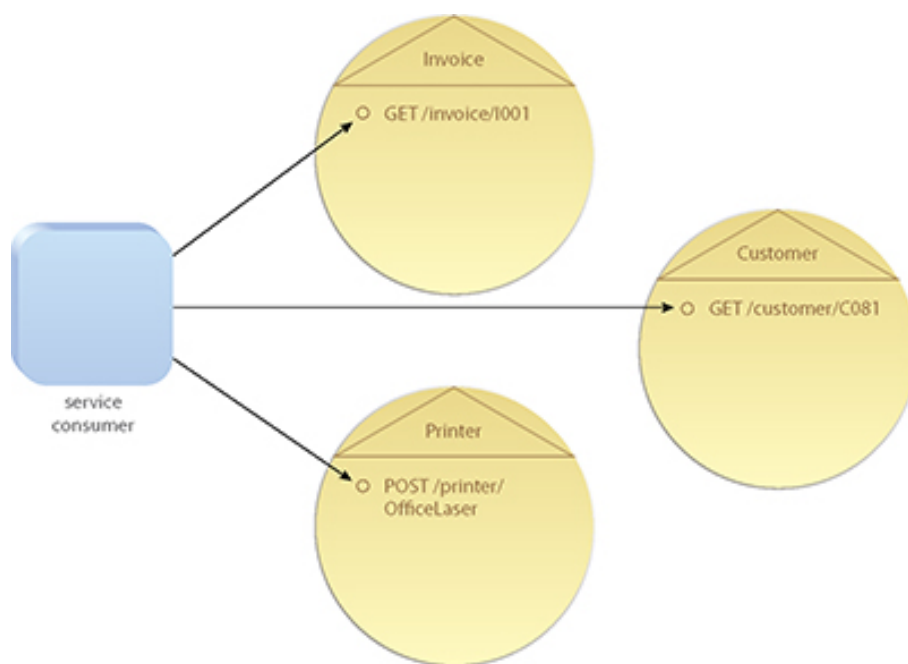


Figure 7.13: Contrato REST. Tomado de [60]

7.4 Caso de Estudio: Monolitos vs Microservicios

7.4.0.1 Monolitos

Las aplicaciones monolíticas [82] [60] consisten en un solo proceso que abarca una sola capa de aplicación, soportando las reglas de negocio, la manipulación de los datos y en ocasiones la interfaz de usuario. Los datos pueden ser almacenados físicamente en una ubicación remota, pero la lógica para su acceso y procesamiento es parte de la aplicación.

Como se observa en la Figura 7.14, en el núcleo de una aplicación monolítica está la lógica de negocio, que definen los servicios que ofrece el sistema. Alrededor del núcleo están los adaptadores que se conectan con servicios externos, que incluyen componentes de acceso a base de datos, de mensajería web y que exponen la interfaz de programación de la aplicación, API.

Ventajas y desventajas de aplicaciones monolíticas. Entre las ventajas se destaca:

- **Pruebas Unitarias:** las aplicaciones monolíticas son fáciles de probar, debido a su popularidad en los últimos años, existen variedades de herramientas para realizar las pruebas unitarias, y debido a que estas aplicaciones corren bajo un mismo proceso, con sólo ejecutar el proceso se pueden hacer las pruebas necesarias.
- **Implementación:** son fáciles de implementar ya que, por lo general, es necesario copiar un único archivo en un directorio.
- **Comunicación:** ya que en las aplicaciones monolíticas todo está construido en un solo programa, no hay necesidad de una comunicación complicada a través de la red.

Y la desventajas de su uso:

- **Desarrollo:** las aplicaciones exitosas tienen el hábito de crecer con el tiempo, este crecimiento se refleja en el incremento de los requerimientos, el alcance de la aplicación y en la ampliación del equipo de desarrolladores, así como aumento de la complejidad del código.



Figure 7.14: Monolito. Tomado de [83]

- **Despliegue continuo:** Para realizar cambios en producción, es necesario redespargar toda la aplicación, independientemente de lo que se necesite modificar, resultando difícil su manejo.
- **Escalabilidad:** cada uno de los módulos de un monolito posee requerimientos de hardware distintos, en consecuencia, la capacidad de escalar de manera independiente se convierte en una tarea compleja.
- **Resistencia a fallas:** ya que todos los módulos que componen las aplicaciones corren sobre un mismo proceso, una falla en cualquiera de estos módulos, puede hacer fallar el proceso completo.
- **Heterogeneidad Tecnológica:** los módulos de la aplicación corren bajo un mismo proceso, y estos están escritos en un mismo lenguaje de programación con un framework específico, haciendo costoso adoptar nuevas tecnologías.

7.4.0.2 Microservicios

El estilo arquitectónico de microservicio [82] [60] se enfoca en desarrollar una sola aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros. Estos servicios se basan en capacidades de negocios y son desplegados independientemente mediante cualquier mecanismo de despliegue automatizado.

En la Figura 7.15 se observa el esquema de la aplicación presentada en la Figura 7.14, adoptando un enfoque orientado a microservicios, donde se puede ver que cada módulo de la aplicación monolítica ahora se ha convertido en un microservicio independiente, que interactúa con otros.

Ventajas y Desventajas de los Microservicios Entre sus ventajas destaca:

- **Heterogeneidad tecnológica:** con un sistema compuesto de múltiples microservicios colaboradores, se puede decidir utilizar diferentes tecnologías dentro de cada uno.
- **Resistencia:** en las aplicaciones basadas en microservicios: si en un microservicio

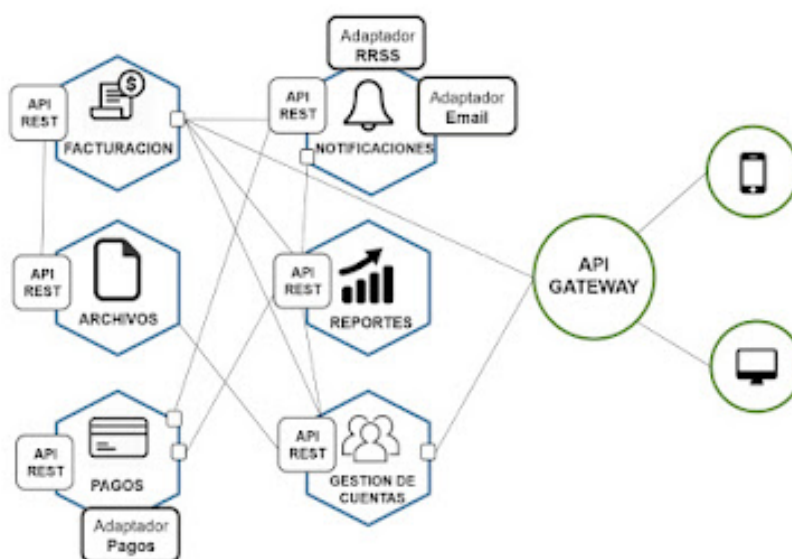


Figure 7.15: Microservicios. Tomado de [83]

se presenta una falla, esta solo afectará a dimono microservicio, y no a todo el sistema, existen mecanismos de resistencia a fallas para que los otros microservicios continúen trabajando.

- **Escalabilidad:** en microservicios, la escalabilidad se puede aplicar a los microservicios que la necesiten, esto permite que se enfoquen los recursos necesarios a los microservicios que lo requieran.
- **Fácil despliegue:** a diferencia de las aplicaciones monolíticas, que para realizar un cambio en producción se requiere el despliegue de toda la aplicación como una pieza independientemente del tamaño del cambio, en los microservicios, el despliegue es por microservicio; un cambio en un microservicio es independiente de los demás que conformen el sistema.
- **Reemplazabilidad:** En un sistema con microservicios individuales de pequeño tamaño, el costo de reemplazarlos o borrarlos, es menor, que reemplazar alguna funcionalidad de un sistema monolítico.

Los microservicios tienen muchas ventajas, pero no son perfectos; tienen asociados las dificultades que acompañan a los sistemas distribuidos. Entre sus desventajas:

- **Complejidad:** por el hecho de que son sistemas distribuidos. Los desarrolladores necesitan elegir e implementar un mecanismo de comunicación entre procesos; también el código para manejar fallos parciales.
- **Bases de datos:** para microservicios se puede manejar varios enfoques para base de datos; es común el crear una base de datos para cada microservicio, lo cual conlleva a transacciones con múltiples bases de datos. Las transacciones en este tipo de sistema, conllevan a gestionar varias bases de datos pertenecientes a diferentes microservicios.
- **Pruebas unitarias:** las pruebas unitarias las cuales son complejas, ya que en este tipo de aplicaciones es necesario acoplar el funcionamiento de los módulos o funciones del sistema que requieran conexiones con otros componentes de la red (bases de

datos o microservicios).



8. Sistemas Punto a Punto

8.1 Introducción

Un sistema punto a punto (P2P, peer-to-peer) es un sistema distribuido en el que todos los elementos interconectados tienen el mismo papel. A diferencia del modelo cliente-servidor, un sistema P2P proporciona acceso a recursos de información ubicados en computadores a través de una red.

En [37] indican que los sistemas P2P son un sistema autoorganizado de iguales, con entidades autónomas (pares) que tiene como objetivo el uso compartido de recursos distribuidos en un entorno en red evitando los servicios centrales.

Es un servicio descentralizado y auto-organizado, con un equilibrio dinámico en el almacenamiento y en el procesamiento de recursos de información. En P2P se distribuye las cargas de trabajo entre los equipos participantes, mientras estos equipos se unen y abandonan el servicio. Esto posibilita el compartir los datos y recursos a muy gran escala.

En los sistemas P2P se eliminan los servidores y su infraestructura. Esto proporcionar servicios y aplicaciones distribuidas utilizando datos y recursos computacionales disponibles en computadoras personales y estaciones de trabajo en Internet. Su diseño garantiza que cada usuario proporciona sus recursos al sistema. Los nodos tienen las mismas capacidades y responsabilidades.

Su correcto funcionamiento no depende de la existencia de cualquier administración centralizada. Están diseñados para ofrecer un grado limitado de anonimato a los proveedores y usuarios de los recursos.

Características

- Su funcionamiento eficiente se basa en su algoritmo para la ubicación de datos y el acceso equilibrado a los mismos.
- Recursos volátiles: sus propietarios no garantizan que vayan a permanecer encendidos, conectados o libres de fallos.
- No garantiza el acceso a recursos individuales. Puede diseñarse para minimizar la probabilidad de fallo de acceso a un recurso replicado.
- Puede usar algoritmos de consenso en la replicación para minimizar fallos (Ejemplo, generales bizantinos).

Uso de los sistemas P2P Algunas áreas donde los sistemas P2P pueden ser de utilidad [4]:

- Red web comunitaria. Cualquier grupo con intereses comunes, incluida una familia o aficionados, puede usar listas y un sitio web para crear su propia intranet.
- Comercio electrónico. P2P puede agregar nuevas capacidades, incluida la conexión y habilitación de los enlaces de una cadena de suministro, distribuir información, contenido o software de manera más eficaz y mantener los elementos de información en su nodo original con un directorio central o una capacidad de búsqueda.
- Juegos. Una infraestructura P2P proporciona una base natural para el desarrollo de juegos comunitarios en línea que no están controlados centralmente. Los desarrolladores pueden centrarse en las características del juego en lugar de la interfaz del protocolo de comunicaciones.
- Los motores de búsqueda. Se puede encontrar información fresca y actualizada buscando directamente en el espacio donde reside el elemento deseado.
- Protección contra el virus. Las relaciones entre los nodos de la comunidad P2P permiten la colaboración en la detección y advertencia de virus, así como la cuarentena automática de la comunidad contra nuevos ataques.
- Servicios adicionales. Hay casos en los que es deseable colocar los datos más cerca del cliente que los solicita. Los módulos de capacitación en línea que contienen segmentos de video, por ejemplo, brindan el efecto deseado cuando los archivos de datos grandes se encuentran cerca del alumno en línea. Múltiples clientes que ofrecen espacio de almacenamiento pueden brindar un servicio más flexible y confiable en comparación con un servidor.
- Desarrollo colaborativo. El alcance puede variar desde el desarrollo de productos de software hasta la redacción de un documento. a aplicaciones como renderizar gráficos.

sistemas P2P no estructurado y estructurado Entre los principales retos [37] de los sistemas P2P radica en la autoorganización descentralizada de un sistema distribuido y en lograr un alto nivel de calidad de servicio sin necesidad de servicios centralizados. Hay dos enfoques principales que se han desarrollado para resolver este problema: sistemas P2P no estructurado y estructurado:

- sistemas P2P no estructurados:
Las primeras aplicaciones para compartir archivos basadas en P2P usaban los llamados enfoques no estructurados. En este enfoque los sistemas dependían de búsquedas a través de un servidor central que almacenaba las ubicaciones de todos los elementos de datos. Solo después de buscar la ubicación de un elemento de datos a través del servidor, los datos se transfirieron directamente entre pares. Otras implementaciones usan algoritmos de inundación (ver algoritmos de inundación en cap. 6 en sección 6.2.3.1). Este enfoque tiene como desventaja la dificultad en escalamiento, además de ser un cuello de botella con respecto a recursos como la memoria, la potencia de procesamiento y el ancho de banda, mientras que los enfoques basados en inundaciones muestran un enorme consumo de ancho de banda en la red.
- sistemas P2P estructurados: Este enfoque estructurado está basado en tablas hash distribuidas, DHT (DHT, Distributed Hash Table). Son un tipo de tablas hash que almacenan pares clave-valor y permiten consultar el valor asociado a una clave, donde los datos se almacenan de forma distribuida en nodos y proveen un servicio eficiente de búsqueda que permite encontrar el valor asociado a una clave.

Las tablas hash distribuidas (DHT) [41] proporcionan una estructura de indexación distribuidas, es decir, el almacenamiento de datos distribuidos y direccionables por contenido; así como escalabilidad, confiabilidad y tolerancia a fallos. Por lo general, un elemento de datos se puede recuperar de la red con una complejidad de $O(\log N)$. De igual manera, agregar nuevo contenido o pares a la red y manejar fallas, comúnmente tienen una complejidad de $O(\log N)$ y $O(\log_2 N)$, respectivamente.

8.2 Evolución de los sistemas P2P

La evolución de los sistemas P2P también se pueden estudiar desde el punto de vista de las generaciones. A saber, hay tres generaciones, la 1era generación basada en servicios de intercambio de música como Napster; la 2da generación en aplicaciones de archivos compartidos, por ejemplo Freenet [7], Gnutella, Kazaa [23], BitTorrent [8]; y la 3era generación basado en tablas hash distribuidas, como Pastry [31], Tapestry [43], CAN [29], Chord [38] y Kademlia [25].

8.2.1 1-era Generación: Napster

Napster fue una aplicación para compartir archivos entre pares que se lanzó escrita por Shawn Fanning [11] y se lanzó el 1 de junio de 1999 con énfasis en la distribución de archivos de audio digital. Las canciones de audio compartidas en el servicio generalmente se codificaban en formato MP3. A medida que el software se hizo popular, la empresa se encontró con dificultades legales por la infracción de los derechos de autor. Cesó sus operaciones en 2001 después de perder una ola de juicios y se declaró en quiebra en junio de 2002 [54] [11] [12].

La arquitectura de Napster se centra en un servidor central o servidor de índices, ver Figura 8.1 que contenían los Índices unificados para cada archivo. El modo de operación de Napster era de la siguiente manera:

1. El usuario accede al servidor de índices para buscar la ubicación del archivo solicitado.
2. El archivo de índices proporciona al usuario una lista de nodo que contiene ese archivo.
3. El usuario hace la solicitud directa del archivo a un nodo seleccionado de la lista proporcionada.
4. El nodo que contiene el archivo le proporciona al usuario el archivo seleccionado.
5. Posteriormente, el nodo que obtuvo el archivo nuevo pasa a formar parte de la lista índices del servidor como un poseedor del archivo.

Napster aprovechó las características especiales de la aplicación, como que los archivos de música nunca se actualizan y no se requieren garantías con respecto a la disponibilidad de archivos individuales. La ventaja de los sistemas centralizados es que son simples de implementar y localizan los archivos de manera rápida y eficiente. Su principal desventaja es que son vulnerables a la censura, acciones legales, vigilancia, ataques maliciosos y fallas técnicas, ya que el contenido compartido, o al menos las descripciones del mismo y la capacidad de acceder a él, están controlados por una sola institución, empresa o usuario.

Napster utilizó un índice unificado (replicado) de todos los archivos de música disponibles. Es probable que el descubrimiento y el direccionamiento de objetos ocasionaran cuellos de botella. Además la coherencia entre las réplicas no era fuerte. Estos sistemas se consideran intrínsecamente no escalables, ya que seguramente habrá limitaciones en el tamaño de la

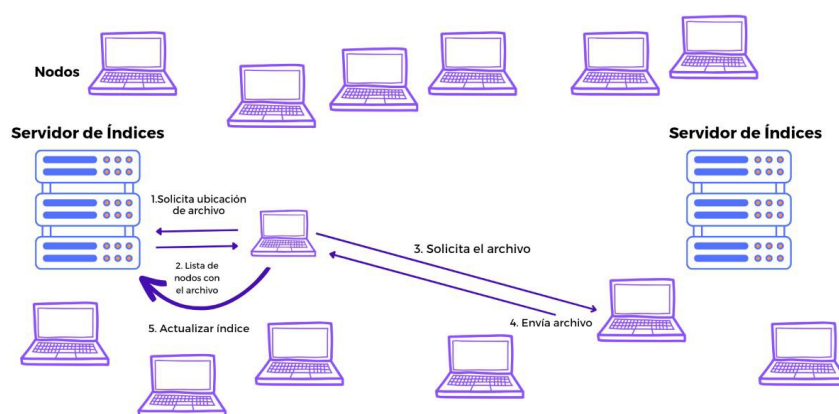


Figure 8.1: Arquitectura de Napster

base de datos del servidor y su capacidad para responder a las consultas y la dificultad de mantener requisitos de consistencia mínimos.

8.2.2 2-da Generación. Gnutella

En Gnutella como enfoque no estructurado, no hay un control general sobre la topología o la ubicación de los objetos dentro de la red. No existe una coordinación central de las actividades de la red. Los usuarios se conectan entre sí directamente de forma *ad hoc* a través de una aplicación de software [12] [73].

Similitudes entre Gnutella y Napster

- Los usuarios colocan los archivos que desean compartir en sus discos duros y los ponen a disposición a todos los demás para descargarlos de igual a igual.
- Los usuarios ejecutan una parte del software Gnutella para conectarse a la red Gnutella.

Diferencias entre Gnutella y Napster

- No existe una base de datos central que conozca todos los archivos disponibles en la red Gnutella. En cambio, todas las máquinas de la red se informan entre sí sobre los archivos disponibles mediante un enfoque de consulta distribuida.
- Hay muchas aplicaciones cliente diferentes disponibles para acceder a la red Gnutella.

Arquitectura de Gnutella En la Figura 8.2 se muestra un esquema de la arquitectura Gnutella. Los nodos son iguales al resto de los nodos pero algunos ofrecen otro servicio. Los **Ultrapeers** son algunos nodos son designados para tener recursos adicionales Las **hojas** se conectan a un pequeño número de ultrapeers que están fuertemente conectados a otros nodo (con más de 32 conexiones cada uno). Esto reduce el número máximo de saltos para una búsqueda exhaustiva.

Este estilo de arquitectura se conoce como arquitectura híbrida y es el enfoque adoptado en Skype (ver cap. 3 en sección 3.4.1).

La estrategia de búsqueda que usa Gnutella se basa en lo siguiente.

1. El nodo solicitante inicia la búsqueda del archivo A
2. Envía mensaje a todos los nodos vecinos solicitando el archivo A

3. En caso de no tener la ubicación del archivo A, los vecinos reenvían el mensaje al resto de sus vecinos en la red.
4. Los nodos que tienen el archivo A inician un mensaje de respuesta
5. El mensaje de respuesta a la consulta se propaga hacia atrás hasta el nodo solicitante
6. Con la ubicación conocida, se descarga el archivo A.

Las hojas y los ultrapeers utilizan el Protocolo de enrutamiento de consultas (QRP, Query Request Protocol) para intercambiar una tabla de enrutamiento de consultas (QRT, Query Routing Table). El QRT consta de una tabla de palabras clave hash que los nodo hoja envía a sus ultrapeers. Un nodo hoja envía su QRT a cada uno de los ultrapeers a los que está conectado, y los ultrapeers fusionan el QRT de todas sus hojas más su propio QRT (si comparten archivos) y lo intercambian con el suyo propio. Luego, el enrutamiento de consultas se realiza mediante el hash de las palabras de la consulta y viendo si todas coinciden en el QRT. Los ultrapeers hacen esa verificación antes de reenviar una consulta a un nodo hoja y también antes de reenviar la consulta a un ultrapeers del mismo nivel, siempre que este sea el último salto que la consulta puede realizar.

El protocolo QRP fue diseñado para reducir el número de consultas emitidas por nodos. El protocolo se basa en el intercambio de información sobre los archivos contenidos en los nodos y solo reenvía las consultas por rutas donde el sistema cree que habrá un resultado positivo. En lugar de compartir información sobre archivos directamente, el protocolo produce un conjunto de números hash en las palabras individuales a partir de un nombre de archivo. El protocolo QRP Gnutella versión 0.4 emplea estos tipos de mensaje:

- Mensajes de difusión:
 - Ping: mensaje de inicio
 - Consulta: patrón de búsqueda y TTL (time-to-live)
- Mensajes de retropropagación:
 - Pong: responde a un ping, contiene información sobre el par.
 - Respuesta a la consulta: contiene información sobre la computadora. que tiene el archivo necesario
- Mensajes de nodo a nodo: con protocolo HTTP
 - GET: devolver el archivo solicitado
 - PUSH: envíame el archivo

8.2.3 3-era Generación. Algoritmos DHT

8.2.3.1 Chord

Chord [38] [17] [41] es un protocolo de búsqueda y un algoritmo para tablas hash distribuidas estructuradas. Los nodos están organizados lógicamente en un anillo de tal forma que un elemento de datos con llave k se mapea hacia el nodo con el identificador más pequeño $id \geq k$. El nodo se conoce como sucesor de la clave k , y se denota como $succ(k)$.

La Figura 8.3 muestra un círculo de identificadores inicializado con $n = 6$, es decir, $2^6 = 64$ identificadores, diez nodos y siete elementos de datos. El sucesor de la clave K5, es decir, el nodo contiguo en el sentido de las agujas del reloj es el nodo N8, donde se encuentra K5. El sucesor de K43 es N43 ya que sus identificadores son iguales. La estructura circular módulo $2^6 = 64$ da como resultado que K61 esté ubicado en N8

Espacio Chord El espacio de nodos en Chord se considera circular, lo que significa que entre los nodos ubicados entre los identificadores 0 y $2^n - 1$ se consideran nodos vecinos. La distancia entre dos nodos se basa en las direcciones. Para calcular esta distancia

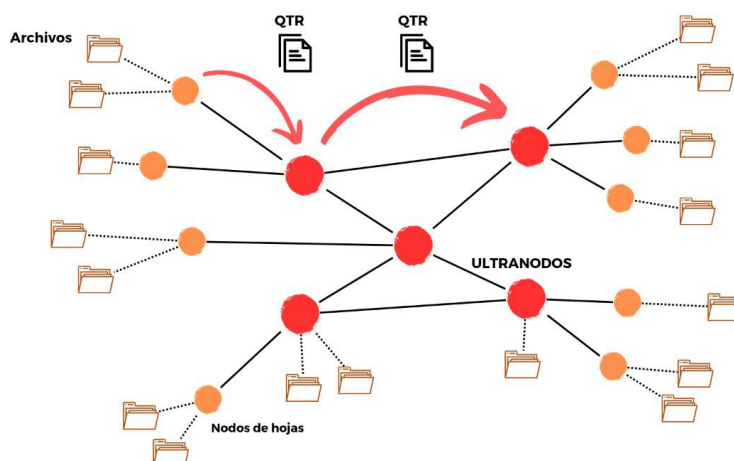


Figure 8.2: Arquitectura de Gnutella

Chord utiliza la diferencia numérica entre los dos identificadores o GUID (GUID, Global Unique Identifier) como distancia.

Noten que la distancia calculada del nodo A al nodo B no es la misma que la distancia calculada del nodo B al nodo A. El cálculo de la distancia entre un nodo A y un nodo B varía dependiendo de si el GUID de A es mayor o menor que el GUID de B:

- Si $GUIDA \leq GUIDB$, entonces $dist(A, B) = GUIDB - GUIDA$
- Si $GUIDA \geq GUIDB$, entonces $dist(A, B) = GUIDB + 2^n - GUIDA$

Tablas de enrutamiento Para que los pares en una red Chord puedan encontrarse entre sí, cada par necesita conocer la dirección IP de su vecino más cercano en términos de los GUID de los nodos en la red Chord. Con sólo conocer el par vecino más cercano en la red es posible encontrar cualquier par en la red. El algoritmo de búsqueda de Chord se detalla a continuación:

1. Si el vecino más cercano es el par GUID que está buscando, la búsqueda finalizó exitosamente.
2. De lo contrario, solicite al vecino más cercano que le devuelva la dirección del par que está buscando o el par más cercano que conoce al par objetivo, si el vecino no conoce al par objetivo:
 - a. Si el par devuelto GUID es el par que está buscando, la búsqueda finalizó correctamente.
 - b. Si el par vecino no conoce a ningún par más cercano al par objetivo que él mismo, no devuelve información del par. La búsqueda finaliza sin éxito.
3. De lo contrario, repita 2) pero envíe la solicitud al par que acaba de recibir de la solicitud de búsqueda anterior.

La búsqueda de pares eventualmente preguntará a todos los pares en la red, uno a la vez, hasta que se encuentre el par que busca, o el último par diga que el par más cercano que conoce al par objetivo es el par que busca en el mismo por de origen, es decir, después de una ronda completa en la red. Entonces el tiempo de búsqueda es de $O(N)$, lo que significa que el tiempo de búsqueda crecerá linealmente con la cantidad de pares en la red Chord.

Para mejorar el tiempo de búsqueda, en la tabla de enrutamiento de Chord se mantiene

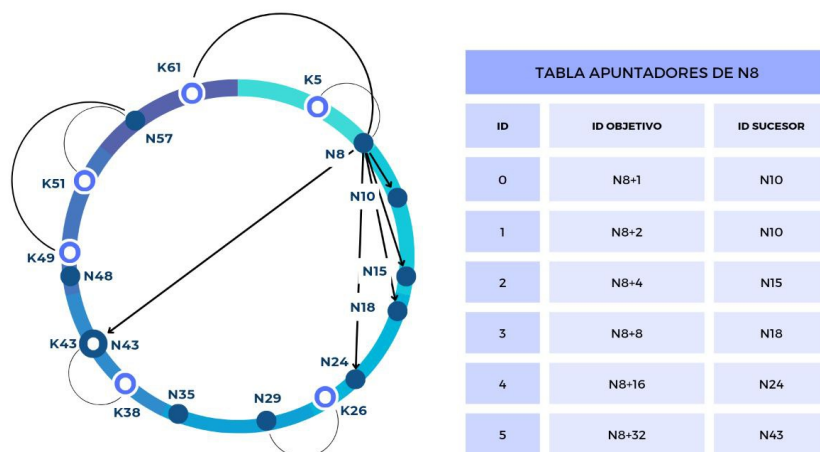


Figure 8.3: Tabla de enrutamiento Chord de 6 bits
. Adaptado de [17]

referencias a pares que están exponencialmente cada vez más distantes según su distancia GUID del nodo de referencia. Primero, la tabla de enrutamiento contendrá una referencia al vecino cercano con una distancia GUID de 1. Luego, una referencia al par con una distancia GUID de 2, 4, 8, 16, y así. usando la distancia exponencial de 2^n .

Por ejemplo en la Figura 8.3, en el nodo n , la entrada de la tabla en la fila i identifica el primer nodo que sucede a n por al menos $2^i - 1$, es decir, $\text{succ}(n + 2^i - 1)$, donde $1 \leq i \leq l$. El segundo apuntador del nodo N8 ($8 + 2^1 = 10$) es el nodo N10 y el tercer dedo ($8 + 2^2 = 12$) es el nodo N15.

Las líneas de puntos indican qué nodos albergan qué claves. Las líneas negras representan a quien apunta el nodo N8.

Protocolo Chord El protocolo Chord asigna a cada dirección IP del nodo y clave de archivo un identificador mediante hash consistente. Esos identificadores están ordenados en un anillo de números del 0 hasta $(2^n) - 1$, donde n es un número elegido por quien inicia la red Chord P2P. El uso de $n = 8$ daría como resultado que los GUID de acordes vayan de 0 a 255. Los valores n comunes son 64, 128, 160 y más; esto depende de lo que se espera que se use el número de nodos que se unan a la red de manera que cada nodo pueda obtener un id único.

La asignación de claves para un nodo se realiza de la siguiente manera:

- Al asignar la clave K a un nodo cuál sería el primer nodo cuyo identificador es igual a K en el espacio de identificadores.
- El nodo sucesor de una K (tecla a) es el primer nodo que encontró en el sentido de las agujas del reloj desde K en el anillo de acordes. por ejemplo: $\text{sucesor}(2) = 3$ cuando un par ingresa al ring, ciertas claves que fueron asignadas al sucesor ahora se asignan a un nuevo par. Cuando un par sale, las claves que fueron asignadas al par saliente ahora se asignan al sucesor.

Tabla de enrutamiento Chord Cuando un nodo quiere unirse al sistema, comienza por generar un identificador aleatorio, id . Observe que si el espacio identificador es lo suficientemente grande, entonces proporcionar el generador del número aleatorio es de buena calidad; la probabilidad de generar un identificador que ya está asignado a un nodo

real es cercana a cero. Entonces, el nodo puede simplemente realizar la búsqueda en un id , lo cual devolverá la dirección de la red de $succ(id)$. En ese punto, el nodo de unión puede simplemente contactar a $succ(id)$ y a su predecesor, e insertarse él mismo en el anillo. Desde luego, este esquema requiere que cada nodo también almacene información sobre su predecesor. La inserción provoca además que cada elemento de datos, cuya llave está ahora asociada con el nodo id , sea transferido desde $succ(id)$. En términos sencillos: el id del nodo informa su punto de inicio a su predecesor y a su sucesor, y transfiere sus elementos de datos a $succ(id)$.

8.3 Caso de Estudio. Pastry

Pastry desarrollado en el año 2001 por los investigadores de Microsoft Research, Antony Rowstron y Peter Druschel [31]. Se trata de un protocolo que se define como un sistema P2P, completamente descentralizado, escalable y autoorganizado. Emplea un protocolo de transporte UDP en la mayoría de los casos.

Los nodos de Pastry forman una red superpuesta descentralizada, autoorganizada y tolerante a fallos dentro de Internet. Pastry proporciona enrutamiento de solicitudes eficiente, ubicación determinista de objetos y equilibrio de carga de manera independiente de la aplicación. Además, Pastry tiene mecanismos que admiten y facilitan la replicación de objetos, el almacenamiento en caché y la recuperación de fallas de aplicaciones específicas [17].

Cada nodo en la red Pastry tiene un identificador único global, GUID (GUID, Global Unique Identifier) en un espacio de identificador circular de 128 bits. Se supone que el identificador del nodo se genera aleatoriamente y que cada GUID tiene la misma probabilidad de ser elegido. Un nodo con un GUID similar puede estar geográficamente lejos. Dada una clave, PASTRY puede entregar un mensaje al nodo con el GUID más cercano a la clave dentro de $\log_2 N$ pasos, donde b es un parámetro de configuración (normalmente $b = 4$) y N es el número de nodos de Pastry activos en la red superpuesta.

Supongamos que queremos encontrar el nodo en la red PASTRY con el GUID más cercano a una clave determinada. Tenga en cuenta que GUID y la clave son secuencias de 128 bits. Tanto el GUID como la clave pueden considerarse como una secuencia de dígitos con base 2^b .

Cada nodo PASTRY tiene un estado que consta de:

- una tabla de enrutamiento utilizada en la primera fase del enrutamiento (largas distancias).
- conjunto de vecindad M contiene el GUID y las direcciones IP de los nodos en $|M|$ más cercanos (según una métrica) al nodo considerado.
- un conjunto de hojas L contiene el GUID y las direcciones IP de los nodos en $|L|/2$ cuyos GUID son numéricamente más cercanos y más pequeños que el GUID actual, y los nodos $|L|/2$ cuyos GUID son numéricamente más cercano y mayor que el GUID actual.

8.3.0.1 Tabla de enrutamiento

8.3.0.2 Enrutamiento circular

Un paquete se puede enrutar a cualquier dirección en el espacio de claves, ya sea que haya un nodo con ese GUID de nodo o no, ver Figura 8.4. Cada hoja contiene un par GUID y dirección IP. El paquete se enruta hacia su lugar adecuado en el anillo circular y el par cuyo

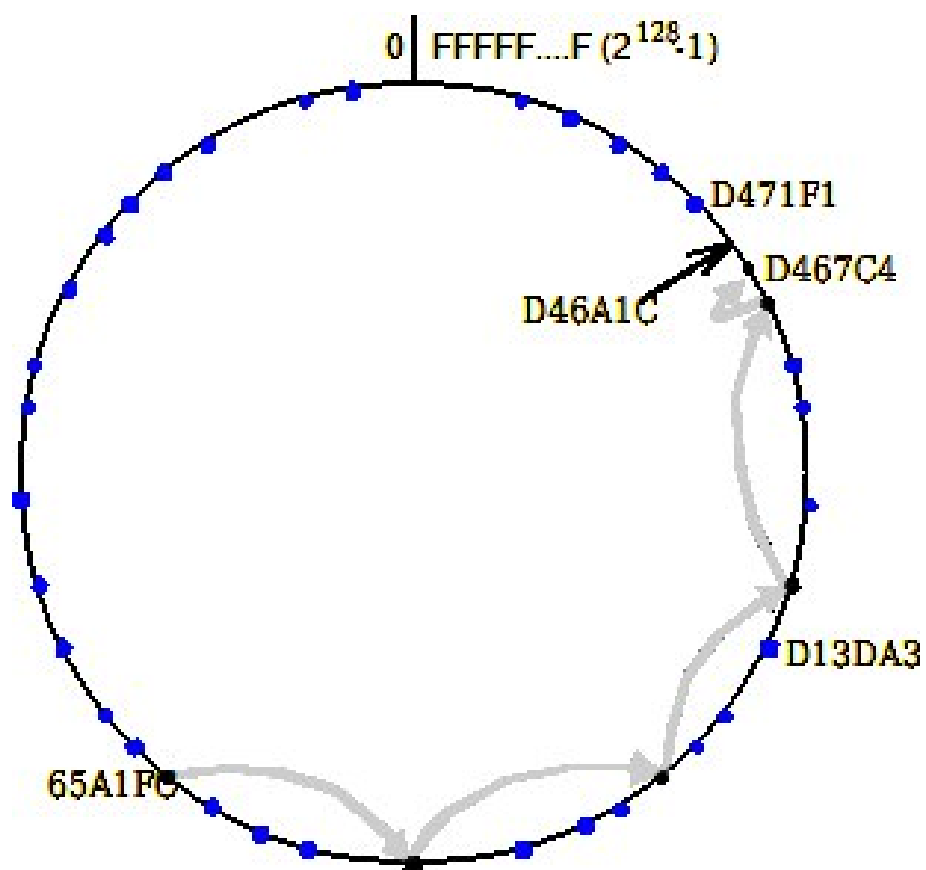


Figure 8.4: Anillo Pastry. Tomado de [54]

GUID esté más cercano al destino deseado recibirá el paquete. El destino más cercano es un nodo cuyo NodeId comparte con la clave un prefijo que es al menos un dígito más largo que el que la clave comparte con el nodo actual. Si no se conoce dicho nodo, el mensaje se reenvía a un nodo que comparte el mismo prefijo del nodo real, pero su NodeId está numéricamente más cerca de la clave.

8.3.0.3 Tabla de enrutamiento

La tabla de enrutamiento es una tabla $\log_{2^b}(N) \times (2^b - 1)$ donde b es el parámetro de configuración y N es el número de nodos PASTRY en la red. Las $(2^b - 1)$ entradas en la fila n cada una se refiere a un nodo cuyo GUID comparte el presente nodo GUID en los primeros n dígitos pero cuyo $(n + 1)$ ésimo dígito tiene uno de los $(2^b - 1)$ posibles valores distintos del $(n + 1)$ ésimo dígito en la identificación del nodo actual. Ejemplo de una tabla de enrutamiento esta en la Figura 8.5.

Si el enrutamiento circular falla en la búsqueda del par, entonces el par consulta a continuación su tabla de enrutamiento con el objetivo de encontrar la dirección del par que comparte un prefijo del GUID más largo con la dirección de destino del propio par buscado. Si no existe, elegirá un par de su lista de contactos con el mismo prefijo de longitud cuyo GUID de nodo esté numéricamente más cerca del destino y enviará el paquete a ese par. Dado que el número de dígitos correctos en la dirección siempre aumenta o permanece igual (y si permanece igual, la distancia entre el paquete y su destino se reduce), el protocolo de enrutamiento converge.

Más en detalle, para manejar un mensaje M que se diriga a un nodo D (donde $R[p, i]$ es el elemento en la columna i , fila p de la tabla de enrutamiento) el algoritmo de búsqueda:

1. Si $L_{-|L|/2} \leq D \leq L_{+|L|/2}$ (//establece si el destino está dentro de la hoja, o es el nodo actual).
2. Dirigir M al elemento de L_i de la hoja de conjunto con GUID más cercano a D o al nodo actual A .
3. else (// usa la tabla de enrutamiento para despachar M a un nodo con un GUID cercano)
4. Búsqueda p , la longitud del prefijo común más larga de D y A , i , el $(p + 1)$ ésimo dígito hexadecimal de D .
5. Si $(R[p, i] \neq \text{nulo})$ dirigir M hacia $R[p, i]$ (// enrutar M a un nodo con el prefijo común más largo).
6. else (// no hay ninguna entrada en la tabla de enrutamiento)
7. Avance M a cualquier nodo en L o R con un prefijo común de longitud p con una GUID que es numéricamente más cerca

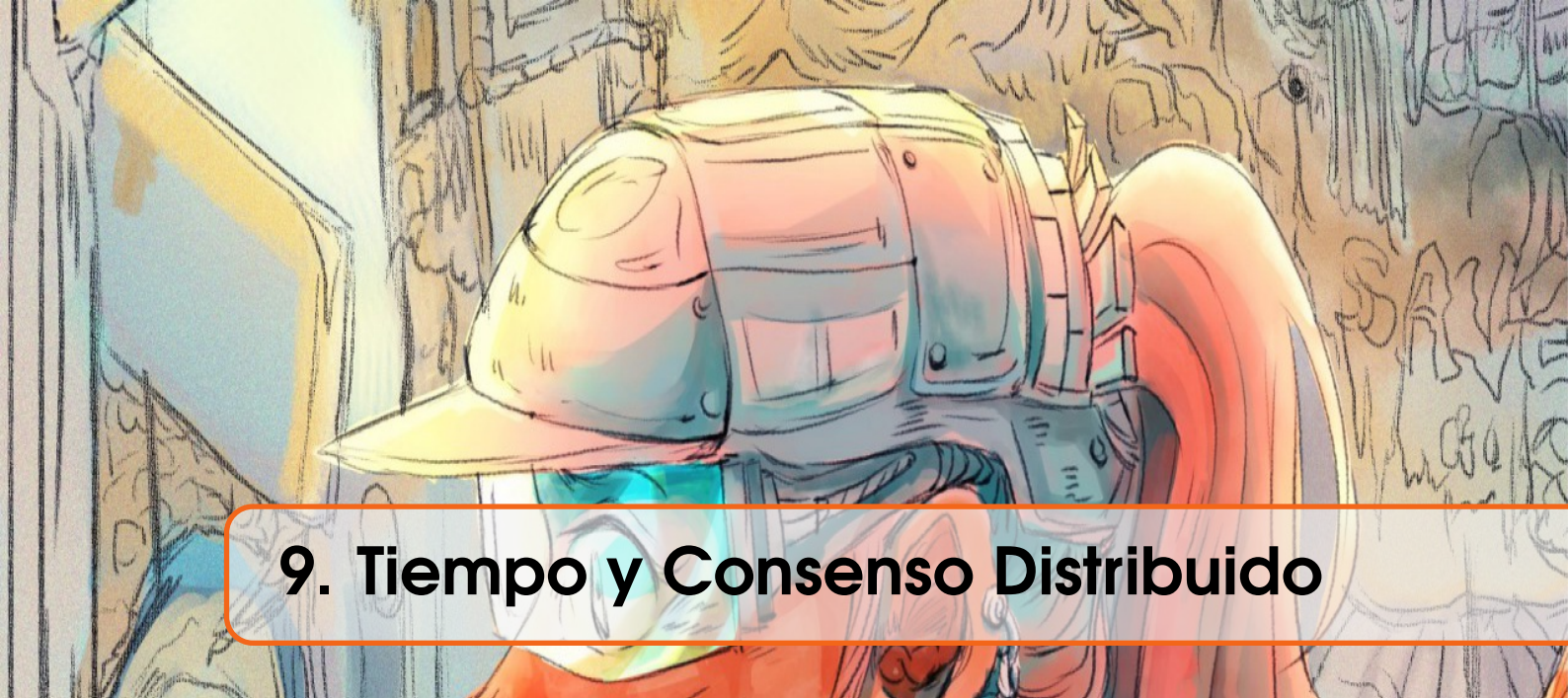
Dado un mensaje, el nodo primero verifica si la clave se encuentra dentro del rango de nodeIds cubiertos por su conjunto de hojas (línea 1). Si es así, el mensaje se reenvía directamente al nodo de destino, es decir, el nodo en el conjunto de hojas cuyo GUID está más cerca de la clave (posiblemente el nodo actual) (línea 3). Si la clave no está cubierta por el conjunto de hojas, entonces se utiliza la tabla de enrutamiento y el mensaje se reenvía a un nodo que comparte un prefijo común con la clave en al menos un dígito más (líneas 6 a 8). En ciertos casos, es posible que la entrada apropiada en la tabla de enrutamiento esté vacía o que no se pueda acceder al nodo asociado (líneas 11 a 14), en cuyo caso el mensaje se reenvía a un nodo que comparte un prefijo con la clave al menos siempre que el nodo local, y esté numéricamente más cerca de la clave que el nodo actual.

$p =$	GUID prefixes and corresponding nodehandles n															
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	n	n	n	n	n	n		n	n	n	n	n	n	n	n	n
1	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
	n	n	n	n	n		n	n	n	n	n	n	n	n	n	n
2	650	651	652	653	654	655	656	657	658	659	65A	65B	65C	65D	65E	65F
	n	n	n	n	n	n	n	n	n	n		n	n	n	n	n
3	65A0	65A1	65A2	65A3	65A4	65A5	65A6	65A7	65A8	65A9	65AA	65AB	65AC	65AD	65AE	65AF
	n		n	n	n	n	n	n	n	n	n	n	n	n	n	n

Figure 8.5: Tabla de enrutamiento Pastry. Tomado de [54]

IV Coordinación y Replicación

9	Tiempo y Consenso Distribuido	167
9.1	Tiempo	167
9.2	Consenso Distribuido	176
9.3	Caso de Estudio. Elección del Líder	185
10	Replicación	191
10.1	Replicación	191
10.2	Tolerancia a Fallas	193
10.3	Caso de Estudio. Fragmentación	198



9. Tiempo y Consenso Distribuido

9.1 Tiempo

Un sistema distribuido consta [54] de en una colección de N procesos que puede expresarse como: N procesos, $p_i \ i = 1, 2, 3, \dots, N$. Cada proceso se ejecuta en un único procesador, y los procesadores no comparten memoria. A su vez, cada proceso p_i en P tiene un estado que se transforma a medida que ejecuta. Incluye los valores de toda las variables internas y cualquier objeto en el entorno de sistema operativo local que afecta, como los archivos. A medida que se ejecuta cada proceso p_i , toma acciones, las cuales son operación de envío o recepción de mensajes, u operaciones que transforma el estado de p_i .

Por ejemplo, si los procesos participan en una aplicación de comercio electrónico, entonces las acciones pueden ser *Mensaje de pedido enviado por el cliente o transacción en el servidor del comerciante para iniciar sesión*.

Un evento es la ocurrencia de una sola acción que conlleva un proceso *fuera de su ejecución* a medida que se ejecuta: una acción de comunicación o una acción que transforma el estado del sistema como todo. La secuencia de eventos dentro de un solo proceso, se puede colocar en un único pedido total, que se denota por la relación i entre los eventos. Es decir, $e - - >_i e'$ si y solo si el evento e ocurre antes de e' en p_i .

Al la serie de eventos e_i que se colocan dentro de un procesos p_i se le llama historia del proceso [54], ordenado por la relación i :
 $history(p_i) = h_i = \langle e_i^0, \langle e_i^1, \langle e_i^2, \dots \rangle \rangle \rangle$.

La sincronización de relojes [56] [81] [93] en un sistema distribuido consiste en garantizar que los procesos p_i se ejecuten de forma cronológica y a la misma vez respetar el orden de los eventos e_i dentro del sistema, es decir la historia del proceso. La sincronización del reloj c_i es un método para sincronizar los valores del reloj de los nodos en un sistema distribuido con el uso de un reloj de referencia externo o un valor de reloj interno.

Tipos de sincronización de relojes físicos

Existen dos tipos de sincronización de relojes físicos:

Sincronización externa . Consiste en sincronizar los relojes c_i de los procesos p_i con una fuente de tiempo externa fiable, $S(t)$ para un límite de sincronización D , de manera que:
 $|S(t) - C_i(t)| < D$; siendo D un límite mayor que cero y S una fuente de tiempo UTC.

Sincronización interna : Si los relojes c_i están sincronizados entre ellos con un conocido grado de precisión y no necesariamente sincronizados con una fuente externa de tiempo:
 $|C_i(t) - C_j(t)| < D$; siendo D un límite mayor que cero.

El envío de mensajes entre los sistemas distribuidos es una parte fundamental para su funcionamiento. La sincronización de relojes ya sean físicos o lógicos aseguran que los procesos se realicen de manera secuencial y ordenada.

9.1.1 Relojes Físicos

Los relojes son dispositivos electrónicos que cuentan las oscilaciones que ocurren en un cristal a con una frecuencia definida, y generalmente dividen este recuento y almacenan el resultado en un contador de registro. Los dispositivos de reloj se pueden programar para generar interrupciones a intervalos regulares para que, por ejemplo, implementar la división de tiempo. En las computadoras el sistema operativo lee el valor del reloj de hardware del nodo, lo escala y agrega un desplazamiento para producir un reloj de software que aproximadamente mide el tiempo físico real t para el proceso p_i .

9.1.1.1 Sesgo y deriva del Reloj

Los relojes de computadora, cualquiera, tienden a no estar en perfecto acuerdo. La diferencia instantánea entre las lecturas de dos relojes se llama su **sesgo**. Los relojes a base de cristal utilizados en las computadoras están, cualquiera, sujetos a la deriva de reloj, lo que significa que cuentan el tiempo a diferentes velocidades muy divergente. esto es debido a los que un reloj no marcha exactamente a la misma velocidad que otro, lo que significa, que después de cierto tiempo la hora indicada por el reloj se irá separando de la indicada por el otro reloj, ver Figura 9.1 que ilustra las diferencias de hora que se podrían generar en consecuencia.

9.1.1.2 UTC

La hora universal coordinada, **UTC** es un estándar internacional para el cronometraje. Las señales UTC se sincronizan y transmiten regularmente desde estaciones terrenas de radio y satélites que cubren muchas partes del mundo. Las fuentes satelitales incluyen el Sistema de Posicionamiento Global (**GPS, Global Positioning System**). Dependiendo de la estación utilizada. Las señales recibidas de los satélites GPS tienen una precisión de aproximadamente 1 microsegundo. Las computadoras con receptores conectados pueden sincronizar sus relojes con estas señales de tiempo.

9.1.1.3 Algoritmo de Cristian

Un ejemplo de sincronización externa es el algoritmo de sincronización de relojes de Cristian.

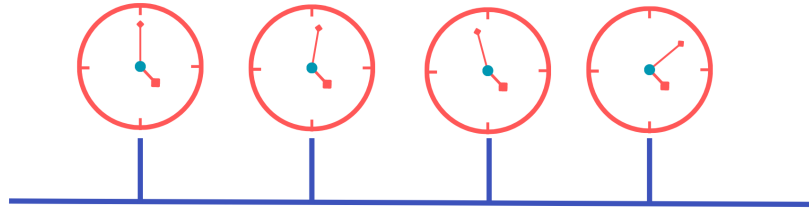


Figure 9.1: Sesgo y deriva del reloj

El Algoritmo de Cristian [10] propone sincronizar un conjunto de relojes de máquinas a partir de una que esté sincronizada externamente (tiempo u hora real), a través de la red de comunicaciones de datos entre computadoras.

Se parte de un sistema distribuido con varios nodos i , cada uno de ellos cuenta con un reloj local C_i . En cualquier instante t se cumple para todos los nodos $C_i(t) = t$, es decir, todos los relojes locales coinciden con la misma hora. Periódicamente cada máquina envía un mensaje para solicitar el tiempo actual a este servidor:

- Un proceso p hace una petición de tiempo al servidor en un mensaje m_r .
- El servidor responde con un mensaje m_t en el que incluye su tiempo t_{UTC} .
- El proceso que recibe el mensaje m_t actualiza su reloj con el tiempo t_{UTC} .
- Hay que considerar el error o demora ya que se ha requerido un tiempo para la transmisión del mensaje desde el servidor: se mide el tiempo que se tarda en recibir la respuesta desde que se envía el mensaje de petición, t_{trans} .
- El tiempo estimado de propagación será $t_{trans}/2$
- El cliente sincroniza su reloj a $t_{UTC} + t_{trans}/2$

Problemas con el Algoritmo de Cristian

1. Si tiempo del emisor es mayor al valor de $t + T_{trans}$ perjudicaría a los archivos compilados.
2. El tiempo de propagación T_{trans} , varía según la carga en la red.
3. Posibilidad de fallo debido a la existencia de un único servidor. Cristian sugiere múltiples servidores de tiempo sincronizados que suministren el tiempo. El cliente envía un mensaje de petición a todos los servidores y toma la primera respuesta recibida.

En la Figura 9.2 muestra el proceso del algoritmo y en la Figura 9.3 hay un esquema con el manejo del tiempo entre el cliente y el servidor para sincronizarse usando el algoritmo de Cristian.

Si se conoce el tiempo que tarda el servidor en manejar la interrupción t_1 , la estimación de tiempo puede mejorar con base en la siguiente expresión: $\frac{T_1 - T_0 - t_1}{2}$, que representa la estimación del tiempo de propagación realizado por el reloj de la máquina emisora.

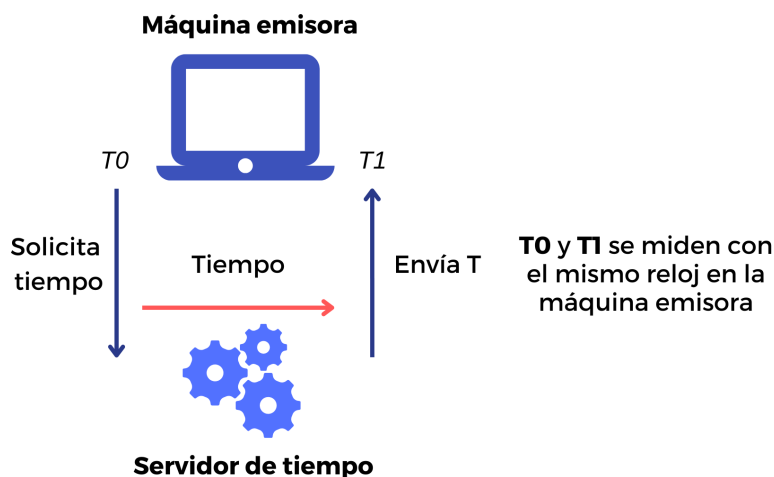


Figure 9.2: Algoritmo de Cristian

Cristian sugiere hacer varias mediciones para mejorar la precisión y descartar los valores límites de $T_1 - T_0$, ya que estos están en función de la operación de la red.

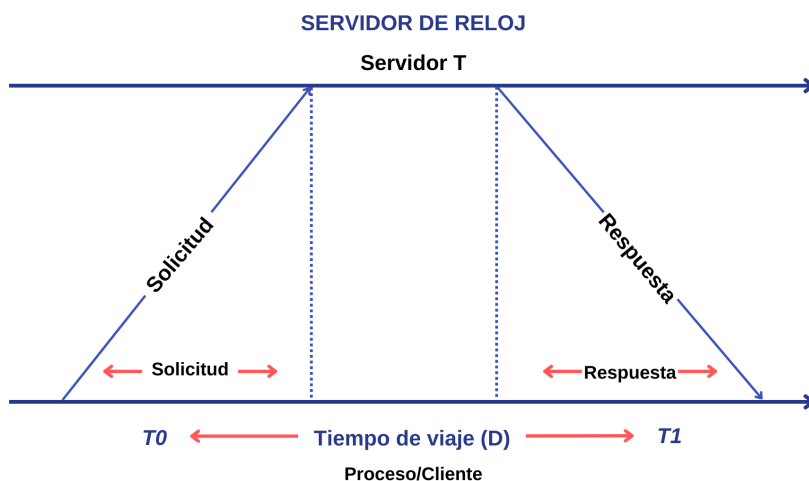


Figure 9.3: Tiempo en el Algoritmo de Cristian

El algoritmo de Cristian es probabilístico ya que no garantiza que un procesador pueda leer un reloj remoto con una precisión específica. Indica que, intentando una cantidad de veces suficiente, la hora recuperada puede ser leída con una precisión dada, con una probabilidad cerca de uno, según lo deseado.

9.1.1.4 Algoritmo de Bekerley

Algoritmo de Bekerley [18] es un algoritmo centralizado o de sincronización interna, en el que se elige un servidor entre todos los equipos que se encuentran conectados en el entorno. Este servidor toma el rol de maestro o servidor de tiempo y los servidores conectados son esclavos.

El algoritmo de Bekerley, opera de la siguiente manera:

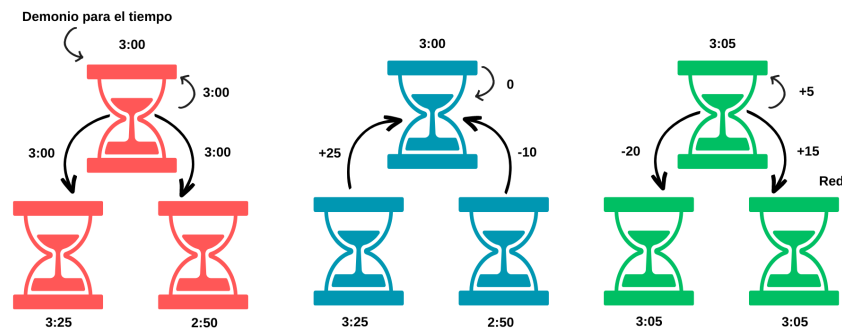


Figure 9.4: Algoritmo de Bekerley

- El servidor de tiempo le envía a cada máquina su tiempo
- Las máquinas cliente le indican el número de segundos que están adelantadas o retrasadas con respecto al tiempo del servidor, si es un número positivo entonces están adelantadas y si es negativo están atrasadas.
- El servidor suma todos estos datos y los divide entre el número de máquinas incluyendo él mismo.
- El resultado lo suma a su propio tiempo obteniendo t , y calcula el número de segundos que le falta o le sobra a una máquina para llegar a ese tiempo t , y se lo envía a la máquina. Lo mismo hace para cada máquina.
- Las máquinas esclavas reciben el resultado y sincronizan su tiempo. Cada equipo se limitará a actualizar su reloj adelantándolo en caso de ir con retraso o atrasándolo en caso de ir adelantado para aplicar la deriva recibida.

En la Figura 9.4 se ilustra como opera el algoritmo de Bekerley.

9.1.1.5 Protocolo NTP

Los algoritmos de Cristian y Berkeley son adecuados para redes rápidas y por lo tanto son usados principalmente para intranets. En [26] Mills define un protocolo llamado NTP (NTP, Network Time Protocol) que es una arquitectura y protocolo para distribuir «relojes» NTP permitir la sincronización de los relojes de clientes en todo internet usando UTC. La latencia/retraso es significativa y variable

Define una jerarquía dependiendo de la calidad del reloj

En la Figura 9.5 se muestra un esquema de los niveles que establece el protocolo basado en la calidad del Reloj. NTP usa los estratos 1 a 16 para definir la precisión del reloj. Un valor de estrato más bajo representa una mayor precisión. Los relojes en los estratos 1 a 15 están en estado sincronizado y los relojes en el estrato 16 no están sincronizados. Un servidor NTP de estrato 1 obtiene su hora de una fuente de tiempo autorizada, como un reloj atómico. Proporciona tiempo para otros dispositivos como servidor NTP principal. Un servidor de hora de estrato 2 recibe su hora de un servidor de hora de estrato 1, y así sucesivamente.

La sincronización entre cada par de elementos de la jerarquía se realiza mediante el

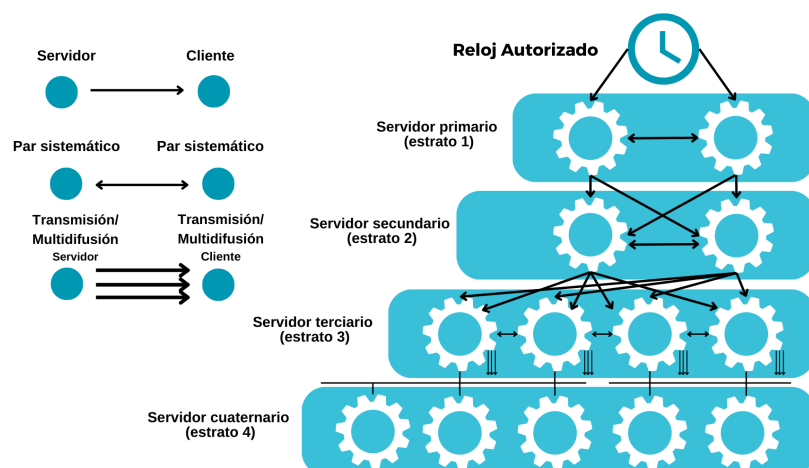


Figure 9.5: Protocolo NTP. Adaptado de h3c

siguiente proceso:

- Modo multicast: Para redes LAN. Se transmite por la red a todos los elementos de forma periódica. Tiene una precisión baja.
- Modo de llamada a procedimiento: Similar al algoritmo de Cristian. Se promedia el retardo de transmisión. Proporciona una mejor precisión.
- Modo simétrico: Los dos elementos intercambian mensajes de sincronización que ajustan los relojes. Tiene una mayor precisión.

Para el intercambio de mensajes entre los servidores se usa datagramas UDP.

9.1.2 Relojes Lógicos

A diferencia de los relojes físicos que miden el tiempo real, los relojes lógicos dan un marco de referencia de los eventos y del orden en el que ocurren. La idea de un reloj lógico consiste en crear un sistema de convergencia del tiempo mediante la medición de las derivas, de manera que la noción de tiempo universal se sustituye por la noción de un tiempo global auto-ajutable. Los relojes lógicos son útiles para ordenar eventos en ausencia de un reloj común.

A continuación se detallan dos algoritmos basados en relojes lógicos: algoritmo de Lamport y algoritmo de los relojes vectoriales.

9.1.2.1 Algoritmo de Lamport

Lamport, [81] [56] [21], señala que la sincronización de relojes no tiene que ser absoluta. Si dos procesos no interactúan no es necesario que sus relojes estén sincronizados. Indica que es importante que los procesos coincidan en el orden en el cual ocurren los eventos. No interesa que los procesos concuerden de manera exacta en la hora. En ciertos algoritmos lo que importa es la consistencia interna de los relojes, no su particular cercanía al tiempo.

El Algoritmo de los relojes lógicos de Lamport define una relación llamada *a ocurre antes de b* representada por $a \rightarrow b$. Esta relación se puede observar de manera directa en dos situaciones:

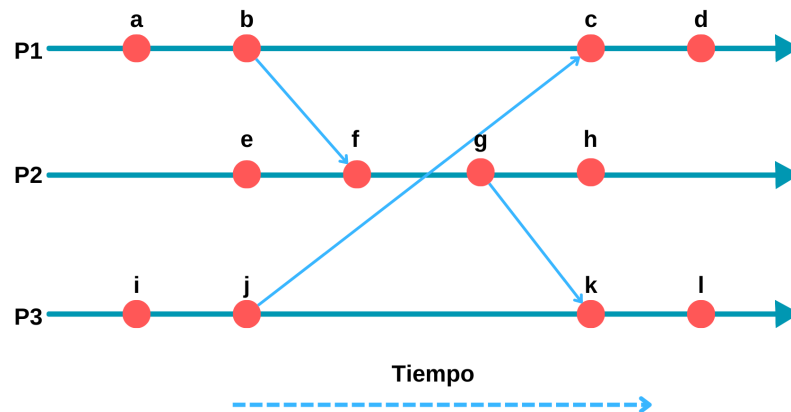


Figure 9.6: Eventos concurrentes

Algoritmo de los relojes lógicos de Lamport

1. Si a, b son eventos en el mismo proceso y a *ocurre antes de* b , entonces $a \rightarrow b$ es verdadero.
2. Si a es el evento de envío de un mensaje por un proceso y b es el evento de la recepción del mensaje por otro proceso, entonces $a \rightarrow b$ también es verdadero. Un mensaje no puede recibirse antes de ser enviado, o incluso al mismo tiempo en que es enviado, ya que necesita cierta cantidad de tiempo finita, diferente de cero, para llegar.

La ocurrencia anterior es una relación transitiva, por lo que si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$. Si dos eventos, x, y , ocurren en diferentes procesos que **no** intercambian mensajes, entonces $x \rightarrow y$ no es verdadera, pero tampoco $y \rightarrow x$. Se dice que estos eventos son eventos concurrentes.

Por ejemplo, en la Figura 9.6 los eventos a, b y c son eventos con relaciones transitivas al igual que los eventos a, b y f . Por otra parte entre los eventos a y e no se puede establecer algún tipo de relación por tanto son eventos concurrentes

Implementación de relojes lógicos de Lamport Para implementar los relojes lógicos de Lamport, cada proceso P_i mantiene un contador local C_i

1. Antes de ejecutar un evento, P_i ejecuta $C_i \leftarrow C_i + 1$
2. Cuando el proceso P_i envía un mensaje m a P_j , éste ajusta el registro de tiempo de $m, ts(m)$, igual a C_i después de haber ejecutado el paso anterior.
3. Al recibir mensaje m , el proceso P_j ajusta su contador local a $C_j \leftarrow \max(C_j, ts(m)) + 1$, después de lo cual ejecuta el primer paso y entrega el mensaje a la aplicación.

Un ejemplo de la implementación del algoritmo de Lamport se muestra en la Figura 9.7. Cuando un proceso p genera un evento, $C_i \leftarrow C_i + 1$ Cuando un proceso envía un mensaje incluye el valor de su reloj Y cuando un proceso q recibe un mensaje m con un valor t , q ajusta su reloj en $C_j \leftarrow \max(C_j, ts(m)) + 1$

Sincronización en Algoritmo de Lamport La sincronización en el algoritmo de los relojes lógicos de Lamport entre procesos puede visualizarse en el ejemplo de la Figura 9.8,

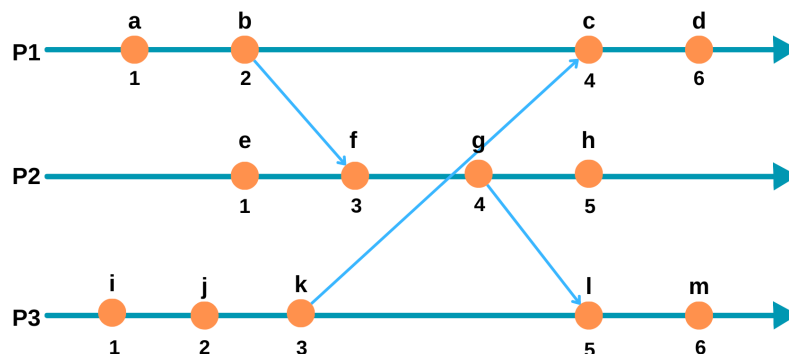


Figure 9.7: Ejemplo del Algoritmo de Lamport.

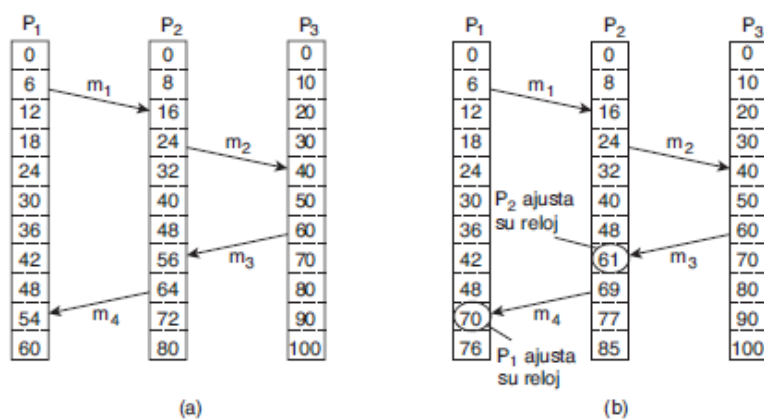


Figure 9.8: Sincronización en el Algoritmo de Lamport. Tomado de [91]

parte a:

- En el tiempo 6, el proceso P_1 envía mensaje m_1 al proceso P_2 . El reloj del proceso P_2 ajusta a 16 al llegar el mensaje.
- Mensaje m_2 desde P_2 hasta P_3 se lleva 16 marcas de tiempo
- Mensaje m_3 de proceso P_3 en 60 y llega a P_2 en 56. Y mensaje m_4 hasta P_1 llega en 54.

Estos valores son imposibles. Indica que esos procesos no están sincronizados. En la parte b de la Figura 9.8 se muestra como debería ser la marca de tiempo en procesos sincronizados.

Problemas con el algoritmo de Lamport

- Con los relojes de Lamport, nada puede decirse sobre la relación entre dos eventos a y b si se compara los valores de tiempo $C(a)$ y $C(b)$. En otras palabras, si $C(a) < C(b)$, entonces esto no necesariamente implica que a realmente ocurrió antes que b .
- Relojes lógicos de Lamport representan una relación de orden parcial. El orden

parcial en la Figura 9.7 es a, e, i, b, j, f, k, c, g, h, l, d, m

9.1.2.2 Algoritmo de Relojes Vectoriales

En distintos documentos, [24] y [14] proponen una solución para superar las dificultades que presenta el Algoritmo de relojes lógicos de Lamport.

- Un reloj vectorial, $VC(a)$, asignado a un evento a , tiene la propiedad de que si $VC(a) < VC(b)$ para algún evento b , entonces se sabe que el evento a precede en causalidad al evento b
- Los relojes vectoriales se construyen de manera que cada proceso P_i mantenga un vector VC_i con las dos siguientes propiedades:
 1. $VC_i[i]$ es el número de eventos que han ocurrido hasta el momento en P_i . En otras palabras, $VC_i[i]$ es el **reloj lógico** del proceso P_i .
 2. Si $VC_i[j] = k$, entonces P_i sabe que han ocurrido k eventos en P_j . Así, éste es el conocimiento de P_i del tiempo local en P_j .

Implementación de Relojes Vectoriales Se realizan los siguientes pasos:

1. Antes de ejecutar un evento, P_i ejecuta $VC_i[i] \leftarrow VC_i[i] + 1$.
2. Cuando el proceso P_i envía un mensaje m a P_j , éste establece el registro de tiempo de m , $ts(m)$, igual a VC_i después de haber ejecutado el paso anterior.
3. Una vez que se recibe el mensaje m , el proceso P_j ajusta su propio vector configurando $VC_j[k] \leftarrow \max(VC_j[k], ts(m)[k])$ para cada k , después de lo cual ejecuta el primer paso y libera el mensaje a la aplicación.

Relojes Vectoriales. Ejemplo Por medio de este mecanismo de Relojes Vectoriales siempre es posible evaluar si dos marcas de tiempo tienen o no relación de precedencia. Basado en la Figura 9.9 se puede decir que:

- $a \rightarrow b \Leftrightarrow V_a < V_b$
- $V_a < V_b \Leftrightarrow V_a \leq V_b \wedge V_a \neq V_b$
- $V_a \leq V_b \Leftrightarrow V_a[i] \leq V_b[i], \forall i \in [1, \dots, N]$
- $V_a = V_b \Leftrightarrow V_a[i] = V_b[i], \forall i \in [1, \dots, N]$

Por tanto mediante los vectores de relojes se puede establecer la precedencia o concurrencia de dos eventos

- $V_a \leq V_b \Leftrightarrow a \rightarrow b$
- $V_b \leq V_a \Leftrightarrow b \rightarrow a$
- $\overline{(V_a \leq V_b)} \wedge \overline{(V_b \leq V_a)} \Leftrightarrow a \parallel b$

Por ejemplo, en la Figura 9.9 ¿qué se podría decir acerca de los procesos a y m ?

El proceso $a \rightarrow (1, 0, 0)$ y el proceso $m \rightarrow (2, 3, 5)$ de allí: $1 \leq 2 \wedge 0 \leq 3 \wedge 0 \leq 5$
 $\wedge$ NOT $(1 = 2 \wedge 0 = 3 \wedge 0 = 5)$. Por lo tanto a ocurre antes que m . $a \rightarrow m$

¿Y con respecto a los procesos c y m ?

El proceso $c \rightarrow (3, 0, 3)$ y el proceso $m \rightarrow (2, 3, 5)$, de allí: $3 \leq 2 \wedge 0 \leq 3 \wedge 3 \leq 5$
 $\wedge$ NOT $(3 = 2 \wedge 0 = 3 \wedge 3 = 5)$ no se cumple, entonces es falso que $V_c \leq V_m$.

De la misma manera, con respecto a los procesos m y c :

$2 \leq 3 \wedge 3 \leq 0 \wedge 5 \leq 3 \wedge$ NOT $(2 = 3 \wedge 3 = 0 \wedge 5 = 3)$ no se cumple, entonces es falso que $V_m \leq V_c$. Por lo tanto c es concurrente con m . $a \parallel m$.

Uso de los Relojes Lógicos Los relojes lógicos de Lamport o los Vectoriales, se aplican a:

- Mensajes periódicos de sincronización.
- Campo adicional en los mensajes intercambiados

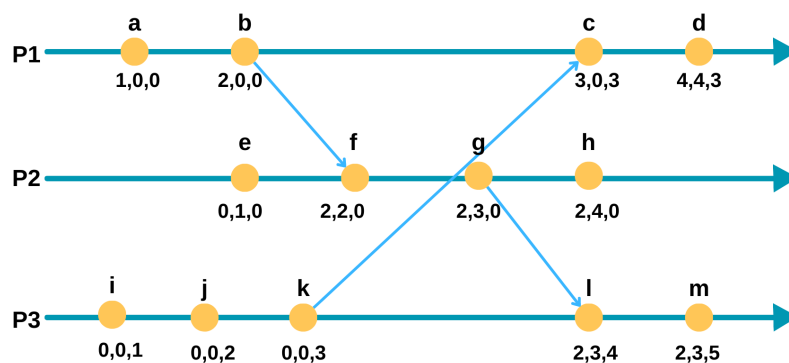


Figure 9.9: Ejemplo del Algoritmo Vectorial.

Por medio de relojes lógicos se pueden resolver el orden de los eventos considerando factores como prioridad o equitatividad. También, se puede detectar violaciones de causalidad.

9.2 Consenso Distribuido

En los sistemas distribuidos resultan fundamentales la concurrencia y la colaboración entre diversos procesos. La concurrencia es la capacidad de ejecutar varios procesos simultáneamente, es decir, la existencia de más de un proceso en períodos de tiempo superpuestos.

Los procesos necesitan el acceso simultáneo a los mismos recursos. Para evitar que tales accesos concurrentes corrompan los recursos, o los vuelvan inconsistentes, se necesita encontrar soluciones que garanticen que los procesos tengan acceso mutuamente exclusivo.

La exclusión mutua garantiza que los procesos concurrentes puedan compartir recursos. A continuación se presentan las soluciones relacionados con el problema de la exclusión mutua [94] [56].

9.2.1 Algoritmos basados en Token

Estos algoritmos logran la exclusión mutua pasando un mensaje especial entre los procesos llamado token [91]. Solo hay un token disponible y solo aquel proceso que lo tenga podrá entrar en la región crítica. Las soluciones basadas en Token usan la siguiente estrategia para garantizar el acceso exclusivo a recursos compartidos:

- La exclusión mutua se logra pasando entre los procesos un mensaje especial conocido como *token*.
- Sólo hay un *token* disponible, y quien lo tenga puede acceder al recurso compartido.
- Cuando termina, el *token* pasa al siguiente proceso.
- Si un proceso tiene el *token* pero no está interesado en acceder al recurso, simplemente lo pasa el *token* al siguiente proceso.

La inanición y el interbloqueo son las situaciones que se deben evitar cuando se usa las soluciones basadas en *token*:

- De acuerdo con la organización de los procesos, éstos pueden garantizar fácilmente que todos tendrán la oportunidad de acceder a los recursos. En otras palabras, evitan la inanición.
- Interbloqueo mediante los cuales diversos procesos se esperan unos a otros para continuar pueden evitarse fácilmente, contribuyendo a su simplicidad.

La desventaja de las soluciones basadas en *token* suceden en el caso de cuando el *token* se pierde (por ejemplo, debido a que falla del proceso que lo tiene), entonces es necesario iniciar un intrincado proceso distribuido para garantizar la creación de un nuevo *token*, pero sobre todo, para que sea el único *token*.

9.2.1.1 Algoritmo de Servidor Central

Una solución simple para la exclusión mutua distribuida utiliza un coordinador [91] [54] [94]. Para cada proceso que solicita el acceso a la sección crítica, se envía su solicitud al coordinador que pone en cola todas las solicitudes y les otorga permiso en función de una determinada regla, por ejemplo, en función de sus marcas de tiempo. En la Figura 9.10 se ilustra el comportamiento de algoritmo del servidor central.

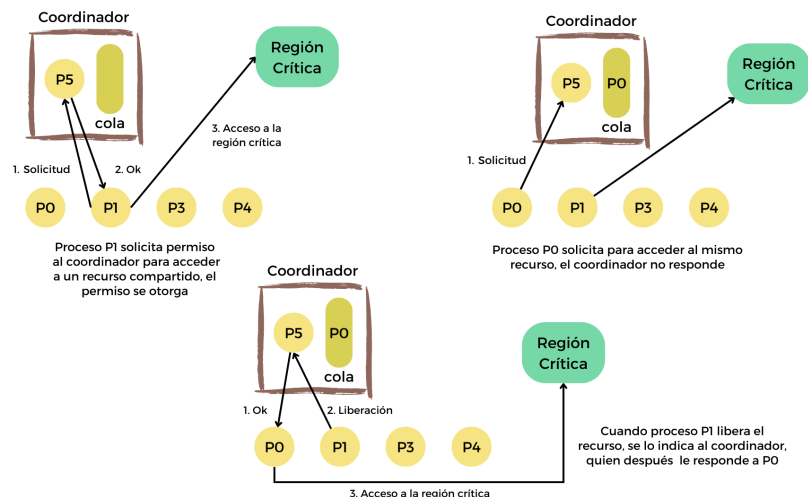


Figure 9.10: Algoritmo de Servidor Central.

Los problemas con este algoritmo se presentan cuando el coordinador falla o cuando falla el poseedor del token. También un solo coordinador en un gran sistema puede ocasionar un embotellamiento.

9.2.1.2 Algoritmo Anillo de Procesos

El algoritmo de Anillos de Procesos [54] [91], Figura 9.11 opera de la siguiente manera:

- Al arrancar el sistema, al proceso de posición 1 se le da un token, el cual irá circulando por el anillo.
- Cuando el proceso k tenga el token, debe transferirlo mediante un mensaje al proceso $k + 1$. Así, el token irá pasando por todos los nodos del anillo.
- Cuando un proceso recibe el token, si quiere entrar a la región crítica, retiene el token y entra en la región crítica.
- Cuando el proceso sale de la región crítica le pasa el token al siguiente nodo del anillo.

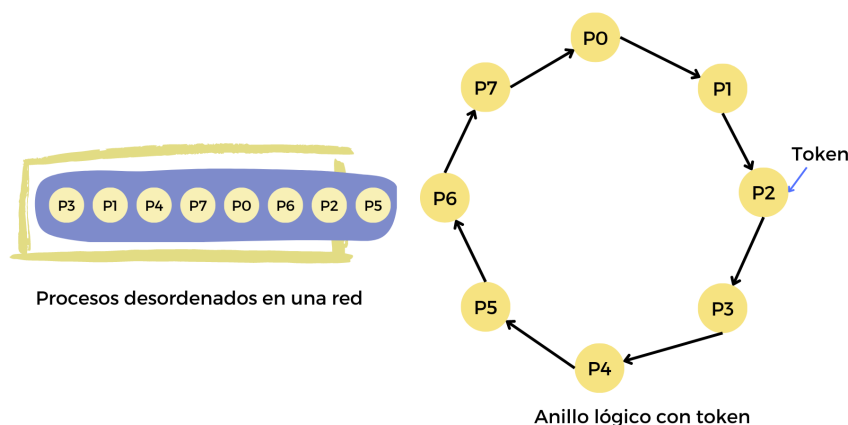


Figure 9.11: Algoritmo de Anillo de Procesos.

Los inconvenientes con este algoritmo es que si falla cualquiera de los nodos provocará el fallo del sistema completo, ya que romperá el anillo circular con el que se conectan todos los nodos. Otra falla es que al acceso a la sección crítica no se logra en el orden en el que los nodos se añaden al anillo, ya que depende del lugar donde se encuentre el token.

9.2.2 Algoritmos basados en Marcas de Tiempo

Los algoritmos basados en marcas de tiempo utiliza marcas de tiempo *timestamps* en lugar de números de secuencia para ordenar las solicitudes de acceso a la sección crítica. Este enfoque sigue este procedimiento:

- Un sitio se comunica con otros sitios para determinar qué sitios deben ejecutar la sección crítica. Esto requiere el intercambio de dos o más rondas sucesivas de mensajes.
- Cada vez que un sitio solicita una sección crítica, recibe una marca de tiempo. La marca de tiempo también se usa para resolver cualquier conflicto entre las solicitudes de sección crítica.
- Todo algoritmo que sigue un enfoque no basado en tokens mantiene un reloj lógico. Los relojes lógicos se actualizan según el esquema de Lamport.

9.2.2.1 Algoritmo de Lamport de Exclusión Mutua Distribuida

Es un algoritmo basado en permisos propuesto por Lamport, [21] como una ilustración de su esquema de sincronización para sistemas distribuidos. Es un mecanismo basado en relojes lógicos para el orden total de las peticiones en el sistema. Los permisos basados en la marca de tiempo *timestamps* se utiliza para ordenar las solicitudes de sección crítica y para resolver cualquier conflicto entre las solicitudes.

Cada nodo mantiene una cola de peticiones que contiene las peticiones ordenadas por marcas de tiempo, y también tiene un conjunto de peticiones de los nodos que necesita permiso para entrar en su región crítica. El algoritmo requiere que los mensajes se entreguen en orden FIFO entre cada par de nodos.

El acceso a la sección crítica se ordena en orden creciente en marcas de tiempo, considerando el menor valor como prioritario. Es decir, una solicitud con una marca de tiempo

más pequeña tendrá permiso para ejecutar la sección crítica primero que una solicitud con una marca de tiempo más grande.

El algoritmo opera así:

- Se utilizan tres tipos de mensajes (SOLICITUD, RESPUESTA y LIBERACIÓN) y se supone que los canales de comunicación siguen el orden FIFO.
- Un sitio envía un mensaje de SOLICITUD a todos los demás sitios para obtener su permiso para ingresar a la sección crítica.
- Un sitio envía un mensaje de RESPUESTA al sitio que solicita el permiso para ingresar a la sección crítica.
- Un sitio envía un mensaje de LIBERACIÓN a todos los demás sitios al salir de la sección crítica.
- Cada sitio S_i , mantiene una cola para almacenar solicitudes de secciones críticas ordenadas por sus marcas de tiempo $\text{Request}(\text{queue}_i)$ denota la cola del sitio S_i .
- Se proporciona una marca de tiempo a cada solicitud de sección crítica utilizando el reloj lógico de Lamport. La marca de tiempo se utiliza para determinar la prioridad de las solicitudes de sección crítica.
- La marca de tiempo más pequeña tiene mayor prioridad sobre la marca de tiempo más grande. La ejecución de la solicitud de sección crítica siempre está en el orden de su marca de tiempo.

Acceso a la sección crítica

- Si sitio S_j desea ingresar a la sección crítica, envía un mensaje de solicitud $\text{Request}(t_{si}, i)$ a todos los demás sitios y coloca la solicitud en **Request**(queue_i). t_{si} es la marca de tiempo de S_{ji} .
- Cuando S_j recibe el mensaje **REQUEST**(t_{si}, i) del sitio S_i , devuelve un mensaje **REPLY** con marca de tiempo al sitio S_i y coloca la solicitud del sitio S_j en **Request**(queue_j).

Ejecutar en la sección crítica

- S_i puede ingresar a la sección crítica si ha recibido el mensaje con una marca de tiempo mayor que (t_{si}, i) de todos los demás sitios y su propia solicitud está en la parte superior de **Request**(queue_i)

Librerar la sección crítica

- S_i sale de la sección crítica, elimina su propia solicitud de la parte superior de su cola de solicitudes y envía un mensaje **RELEASE** con marca de tiempo a todos los demás sitios.
- Cuando un sitio S_i recibe el mensaje **RELEASE** con marca de tiempo de sitio S_i , elimina la solicitud de S_i de su cola de solicitudes.

El Algoritmo de Lamport de Exclusión Mutua requiere la invocación de $3(N - 1)$ mensajes por ejecución de sección crítica: $(N - 1)$ mensajes de solicitud, $(N - 1)$ mensajes de respuesta y $(N - 1)$ mensajes de liberación

Inconveniente Algoritmo de Lamport de Exclusión Mutua

- Enfoque no confiable: la falla de cualquiera de los procesos detendrá el progreso de todo el sistema.

En cuanto a su rendimiento, el retardo de sincronización es igual al tiempo máximo de transmisión de mensajes. Requiere $3(N - 1)$ mensajes por ciclo de ejecución, y se puede optimizar a $2(N - 1)$ mensajes omitiendo el mensaje **REPLY** en algunas situaciones

9.2.2.2 Algoritmo de Ricart y Agrawala

El algoritmo Ricart-Agrawala es un algoritmo de exclusión mutua en un sistema distribuido propuesto por *Glenn Ricart* y *Ashok Agrawala* [30]. Este algoritmo es una extensión y optimización del Algoritmo de Exclusión Mutua Distribuida de Lamport. Al igual que el algoritmo de Lamport, también sigue un enfoque basado en permisos para garantizar la exclusión mutua.

En la Figura 9.12 se muestra un esquema de la operación del algoritmo.

Operación

- Se utilizan dos tipos de mensajes (REQUEST y REPLY) y se supone que los canales de comunicación siguen el orden FIFO.
- Un sitio envía un mensaje de SOLICITUD a los demás sitios para obtener su permiso para ingresar a la sección crítica.
- Un sitio envía un mensaje de RESPUESTA a otro sitio para dar su permiso para ingresar a la sección crítica.
- Se proporciona una marca de tiempo a cada solicitud de sección crítica utilizando el reloj lógico de Lamport.
- La marca de tiempo se utiliza para determinar la prioridad de las solicitudes de sección crítica.
- La marca de tiempo más pequeña tiene mayor prioridad sobre la marca de tiempo más grande.

Acceso a la sección crítica

- Cuando un sitio S_i desea ingresar a la sección crítica, envía un mensaje de **SOLICITUD** con marca de tiempo a todos los demás sitios.
- Cuando un sitio S_j recibe un mensaje de **SOLICITUD** del sitio S_i , envía un mensaje de **RESPUESTA** al sitio S_i **si y solo si** Site S_j no solicita ni ejecuta actualmente la sección crítica.
- En caso de que Site S_j lo solicite, la marca de tiempo de la solicitud del Site S_i es **más pequeña** que la de su propia solicitud. De lo contrario, el sitio S_j aplaza la solicitud.

Operación en la sección crítica El sitio S_i ingresa a la sección crítica si ha recibido el mensaje **RESPUESTA** de todos los demás sitios.

Liberar la sección crítica Al salir del sitio, S_i envía un mensaje de **RESPUESTA** a todas las solicitudes diferidas.

El algoritmo Ricart-Agrawala requiere la invocación de $2(N - 1)$ mensajes por ejecución de sección crítica.

Su desventaja es su enfoque no confiable: la falla de cualquiera de los nodos del sistema puede detener el progreso del sistema. En esta situación, el proceso morirá de hambre.

Este problema de falla del nodo se puede resolver detectando la falla después de un tiempo de espera.

El Rendimiento:

1. El retardo de sincronización es igual al tiempo máximo de transmisión de mensajes
2. Requiere $2(N - 1)$ mensajes por ejecución de sección crítica.

9.2.2.3 Algoritmo de Suzuki-Kazami

El algoritmo de Suzuki-Kazami [39] es una modificación del algoritmo Ricart-Agrawala, un algoritmo basado en permisos que utiliza mensajes de SOLICITUD y RESPUESTA para garantizar la exclusión mutua.

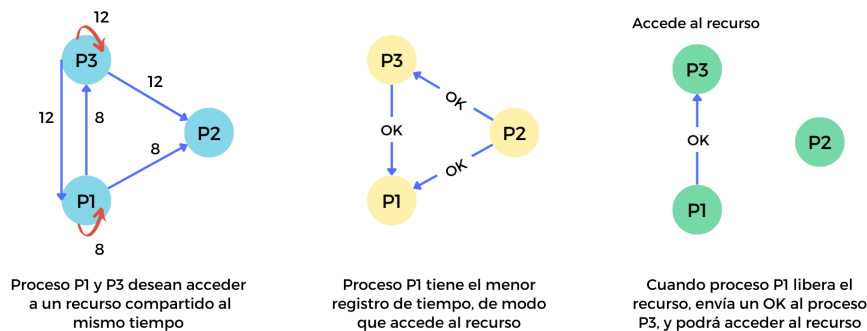


Figure 9.12: Algoritmo de Ricart-Agrawala.

En este algoritmo se presenta un método en el que se modifica la antigüedad y también se entrega la sección crítica a otro nodo enviando un solo mensaje de PRIVILEGIO. Entonces, el nodo que tiene el PRIVILEGIO puede usar la sección crítica. Si un proceso quiere ingresar a su sección crítica y no tiene el token, transmite un mensaje de solicitud a todos los demás procesos del sistema. El proceso que tiene el token, si no se encuentra actualmente en una sección crítica, enviará el token al proceso solicitante. Cada solicitud de sección crítica contiene un número de secuencia. Este número de secuencia se utiliza para distinguir las solicitudes antiguas de las actuales.

Estructura de Datos y notaciones

- Una matriz de enteros $RN[1 \dots N]$ Un sitio S_i mantiene $RN_i[1 \dots N]$, donde $RN_i[j]$ es el mayor número de secuencia recibido hasta el momento a través del mensaje SOLICITUD del sitio S_j .
- Una matriz de enteros $LN[1 \dots N]$ El token guarda en esta matriz. $LN[j]$ el número de secuencia de la solicitud que el sitio S_j ejecutó recientemente.
- Una cola q El token utiliza esta estructura de datos para mantener un registro de la identificación de los sitios que esperan el token.

Ingreso a la sección crítica

- Cuando un sitio S_i desea ingresar a la sección crítica y no tiene el token, incrementa su número de secuencia $RN_i[i]$ y envía un mensaje de solicitud $SOLICITUD_{(i,sn)}$ a todos los demás sitios para solicitar el token. sn es el valor de actualización de $RN_i[i]$.
- Cuando un sitio S_j recibe el mensaje de solicitud $SOLICITUD_{(i,sn)}$ del sitio S_i , establece $RN_j[i]$ al máximo de $RN_j[i]$ y sn , es decir, $RN_j[i] = \max(RN_j[i], sn)$.
- Después de actualizar $RN_j[i]$, el sitio S_j envía el token al sitio S_i si tiene token y $RN_j[i] = LN[j] + 1$.

Ejecutar la sección crítica El nodo S_i ejecuta la sección crítica si ha adquirido el token.

Liberar la sección crítica Después de terminar la ejecución, el nodo S_i sale de la sección crítica y hace lo siguiente:

- Establece $LN[i] = RN_i[i]$ para indicar que su solicitud de sección crítica $RN_i[i]$ ha sido ejecutada.

- Para cada nodo S_i , cuyo ID no está presente en la cola de tokens Q , agrega su ID a Q si $RN_{i[j]} = LN[j] + 1$ para indicar que el sitio S_j tiene una solicitud pendiente.
- Después de la actualización anterior, si la cola Q no está vacía, extrae una ID de sitio de la Q y envía el token al sitio indicado por la ID extraída.
- Si la cola Q está vacía, conserva el token

Rendimiento El algoritmo requiere la invocación de 0 mensajes si el sitio ya tiene el token inactivo en el momento de la solicitud de la sección crítica o un máximo de N mensajes por ejecución de la sección crítica. Este N mensajes implica:

- $(N - 1)$ mensajes de solicitud
- 1 mensaje de respuesta

9.2.3 Algoritmos basados en Elecciones

El objetivo de los algoritmos basados en elecciones es el de elegir un proceso único para que tome un determinado rol o para decidir una determinada acción n

Entre sus aplicaciones estan las de:

- Elegir un nuevo servidor si se cae el actual
- Elegir un nuevo proceso para entrar en una sección crítica
- Elegir el proceso menos activo (balanceo de carga)
- Elegir el proceso con la copia más reciente (réplicas)

Los algoritmos basados en Elecciones operan así:

- Un proceso convoca elecciones cuando lleva a cabo una acción que inicia el algoritmo de elección
- Puede haber N elecciones concurrentes
- Un proceso siempre tiene uno de estos dos roles:
 1. Participante: comprometido en una ejecución del algoritmo
 2. No participante: no comprometido en ninguna ejecución
- El proceso elegido debe ser único, incluso en elecciones concurrentes)
- Todos los procesos tienen un identificador Único para el conjunto
- El proceso elegido es aquél de mayor identificador Variable elegido
- Cada proceso p_i mantiene una variable que contiene el identificador del proceso elegido
- Cuando el proceso se convierta en participante, fija la variable al valor especial $\pm$, indicando que no hay consenso todavía

9.2.3.1 Algoritmos de Anillo basado en Elecciones

En el algoritmo de Anillo basado en Elecciones [6] se sigue el siguiente proceso:

1. Inicialmente todos los procesos son **no participantes**.
2. Cualquier proceso P decide arrancar una elección en cualquier momento.
3. . Proceso P se pone en estado **participante** y envía un mensaje de elección M a su vecino.
4. El mensaje M contiene el **ID** del proceso que ha iniciado la elección.
5. Cuando el vecino recibe el mensaje de elección M , establece su estado como participante y comprueba el **ID** del mensaje.
6. Si es mayor que su propio **ID**, entonces se lo envía directamente a su vecino.
7. Si su **ID** es mayor al **ID** recibido, entonces lo coloca en el mensaje M y lo envía a su vecino.
8. Así se circula el mensaje M sucesivamente hasta que llega a un proceso P_n que

comprueba que el **ID** recibido es el propio. Eso indica que ha sobrevivido el mayor **ID**, que es la del proceso P_n .

9. Entonces, este proceso P_n es el coordinador y lo notifica a su vecino. Cuando un proceso recibe un mensaje de coordinador debe de poner su estado como **no participante** y enviar el mensaje a su vecino.
10. Cuando el mensaje de coordinador retorna al proceso que lo emitió (coordinador), entonces todos los procesos saben quién es el coordinador y todos quedan en estado **no participantes**.

En la Figura 9.13 se muestra paso a paso como opera el algoritmo.

Los algoritmos de Anillo basado en Elecciones se usan cuando:

1. Los procesos están física o lógicamente ordenados en anillo.
2. No se conoce el número total de procesos (n).
3. Cada proceso se comunica con su vecino (izquierda o derecha).

Extinción del Proceso

1. En caso de que dos procesos inicien al mismo tiempo una elección y se envíen mensajes de elección, un proceso en estado de **participante** debe de verificar el **ID** del proceso que envía el mensaje de elección.
2. Si es menor al propio, el mensaje se descarta. Así, todos los mensajes de elección se extinguirán, excepto el que lleva el **ID** más alto.

9.2.3.2 Algoritmos de *Bully*

Las premisas del algoritmos de *Bully* [16]

1. El sistema es síncrono y utiliza tiempo de espera para la identificación de fallas en los procesos.
2. Se permite que los procesos se bloqueen durante la ejecución del algoritmo.
3. La entrega de mensajes entre procesos se supone fiable y dentro de un periodo máximo.
4. Los procesos están ordenados, tienen un único identificador (**IDs**) conocido y se sabe cuántos procesos existen.

Los tipos de mensajes entre los procesos:

1. Mensaje de Elección: seleccionar un nuevo coordinador.
2. Mensaje de Respuesta: respuesta al mensaje de elección.
3. Mensaje de Coordinador: comunica el ID del proceso seleccionado como coordinador.

El algoritmos de *Bully* opera de la siguiente manera:

1. Un proceso x manda un mensaje de elección a todos aquellos procesos que tengan un identificador más grande cuando detecta que el coordinador ha fallado.
2. El proceso x espera los votos y, si ningún voto (ok) llega después de un cierto tiempo, el proceso x se declara coordinador, y envía el mensaje de coordinador a los procesos con identificador más pequeño que el suyo.
3. Si un voto llega (aunque pueden llegar varios votos), puede ser que otro coordinador sea declarado ganador.
4. Si un proceso recibe un mensaje de elección, envía un voto y otra elección empieza.
5. Cuando un coordinador que ha fallado regresa, empieza una elección y puede volver a readquirir el control, aunque exista un coordinador actual.

En la Figura 9.14 se ilustra el proceso de elección del coordinador que propone el algoritmo del Bully.

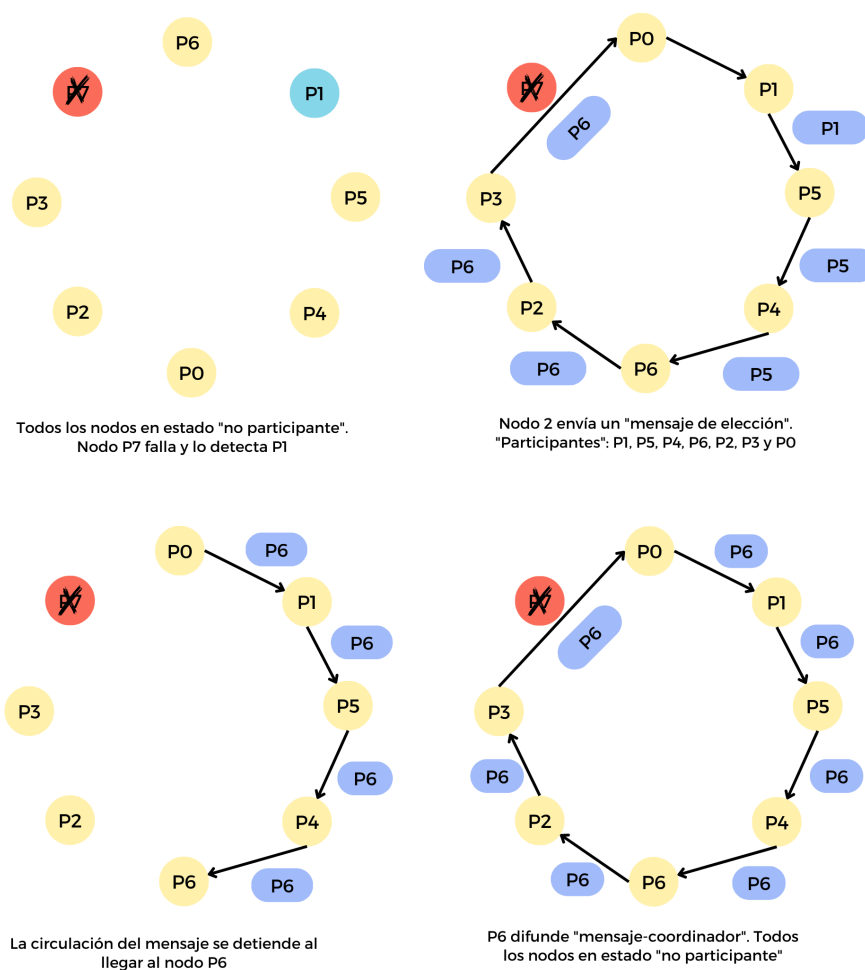


Figure 9.13: Algoritmo de Anillo basado en Elección.

9.2.3.3 Algoritmo Generales Bizantinos

El problema de los generales bizantinos [22] plantea que un grupo de generales sitia una ciudad y deben ponerse de acuerdo a través de un plan de ataque para atacar o retirarse. Los generales solo se comunican a través de mensajes a los otros generales. Uno de ellos, el comandante, da las órdenes. Los otros, tenientes, deben decidir si atacar o retirarse. Sin embargo, uno o más de los generales puede ser un traidor o pueden fallar.

Esta traición puede verse de dos formas:

1. Los mensajes pueden no llegar o, dicho de otra manera, las comunicaciones no son confiables.
2. Un general traidor puede mentir, es decir, un nodo puede fallar de manera impredecible

Algoritmo Generales Bizantinos. Caso 1: tres generales Escenario: un comandante y dos tenientes, y uno de ellos es traidor. ¿Pueden llegar a un consenso?, es decir, ¿acordar **atacar** o **retirarse**? Se asume que el comandante es el traidor

1. El comandante indicará al teniente 1 **atacar** y al teniente 2 **retirarse**. Tenientes verifican la orden recibida comunicándose entre ellos.

2. Teniente 1 al teniente 2 que recibió **atacar**
3. Teniente 2 al teniente 1 que recibió **retirarse**.
4. Tenientes deducen que el comandante es traidor y no pueden tomar una decisión consensuada.

Ver la Figura 9.15 con la ilustración de la operación del algoritmo.

Algoritmos Generales Bizantino. Caso 2- Cuatro Generales Se presentan dos escenarios:

Escenario 1: un comandante y dos tenientes, y uno de ellos es traidor. ¿Pueden los generales leales llegar a un consenso?, es decir, ¿acordar **atacar** o **retirarse**?

1. Asume que el **comandante** es el traidor
2. Independiente de las órdenes que reciban los tenientes del comandante, ellos intercambian información entre sí, de tal manera que los tres recibirán tres mensajes.
3. Al decidir por la mayoría de las órdenes recibidas, los tres tenientes tomarán la misma decisión. En este caso, **atacar**

Escenario 2: un general y dos tenientes, y uno de ellos es traidor. ¿Pueden los generales leales llegar a un consenso?, es decir, ¿acordar **atacar** o **retirarse**?

1. Asume que el **teniente 3** es el traidor
2. El comandante envía a todos los tenientes la orden de **atacar**.
3. Esta orden es retransmitida por los tenientes 1 y 2 a los demás
4. Teniente 3 cambia la orden recibida y envía a los tenientes generales 1 y 2 la orden de **retirarse**.
5. Esto no cambia la decisión consensuada de los tenientes 1 y 2: por mayoría deciden atacar.

En la Figura 9.16 se muestra el intercambio de mensaje entre los generales para el caso de cuatro generales.

9.3 Caso de Estudio. Elección del Líder

Un problema muy común es la elección de líder, donde los nodos que forman parte de un sistema distribuido necesitan elegir un nodo entre ellos para que actúe como su líder, coordinando el funcionamiento de todo el sistema. Un ejemplo de esto es el esquema de replicación de un solo maestro que se presentó anteriormente en el libro. Este esquema se basa en que un nodo, designado como primario, será el responsable de realizar operaciones de actualización de datos y los otros nodos, designados como secundarios, realizarán el seguimiento con las mismas operaciones. Sin embargo, para poder hacerlo, el sistema primero necesita seleccionar el nodo primario, lo cual es un proceso llamado elección de líder. Dado que todos los nodos prácticamente están de acuerdo en un único valor, la identidad del líder, este problema puede modelarse fácilmente como un problema de consenso [93]

El algoritmo de elección de líder de Raft [81] se implementa con una máquina de estados en la que un proceso se encuentra en uno de tres estados (ver Figura 9.17):

- el estado seguidor, en el que el proceso reconoce a otro como líder;
- el Estado candidato, en el que se inicia el proceso de una nueva elección proponiéndose como líder;
- o el estado líder, en el que el proceso es el líder.

En Raft, el tiempo se divide en períodos electorales de duración arbitraria. Un período electoral se representa con un reloj lógico, un contador numérico que sólo puede aumentar

con el tiempo. Un mandato comienza con una nueva elección, durante la cual uno o más candidatos intentan convertirse en líder. El algoritmo garantiza que para cualquier término haya como máximo un líder. Pero, en primer lugar, ¿qué desencadena una elección?

Cuando el sistema se inicia, todos los procesos comienzan su recorrido como seguidores. Un seguidor espera recibir un mensaje periódico del líder que contiene el período electoral en el que fue elegido el líder. Si el seguidor no recibe ningún mensaje dentro de un período de tiempo determinado, se activa un tiempo de espera y se presume que el líder está muerto. En ese punto, el seguidor comienza una nueva elección incrementando el período electoral actual y haciendo la transición al estado candidato. Luego vota por sí mismo y envía una solicitud a todos los procesos del sistema para que voten por él, sellando la solicitud con el período electoral actual.

El proceso permanece en el estado candidato hasta que sucede una de tres cosas: gana las elecciones, otro proceso gana las elecciones o pasa un tiempo sin ganador.

- El candidato gana las elecciones: el candidato gana las elecciones si la mayoría de los procesos del sistema votan a favor. Cada proceso puede votar por como máximo un candidato en un mandato por orden de llegada. Esta regla de la mayoría impone que como máximo un candidato pueda ganar un mandato. Si el candidato gana las elecciones, pasa al estado líder y comienza a enviar mensajes a los demás procesos.
- Otro proceso gana las elecciones: si el candidato recibe un mensaje de un proceso que afirma ser el líder con un mandato mayor o igual al mandato del candidato, acepta al nuevo líder y regresa al estado seguidor. Si no, continúa en el estado candidato. Quizás se pregunte cómo pudo suceder eso; por ejemplo, si el proceso de candidatura se detuviera por cualquier motivo, como por una pausa prolongada de la Asamblea General, para cuando se reanude, otro proceso podría haber ganado las elecciones.
- Pasa un período de tiempo sin un ganador: es poco probable, pero posible, que varios seguidores se conviertan en candidatos simultáneamente y ninguno logre recibir la mayoría de los votos; esto se conoce como votación dividida. Cuando eso suceda, el candidato eventualmente expirará y comenzará una nueva elección. El tiempo de espera de las elecciones se elige aleatoriamente entre un intervalo fijo para reducir la probabilidad de otra votación dividida en las próximas elecciones.

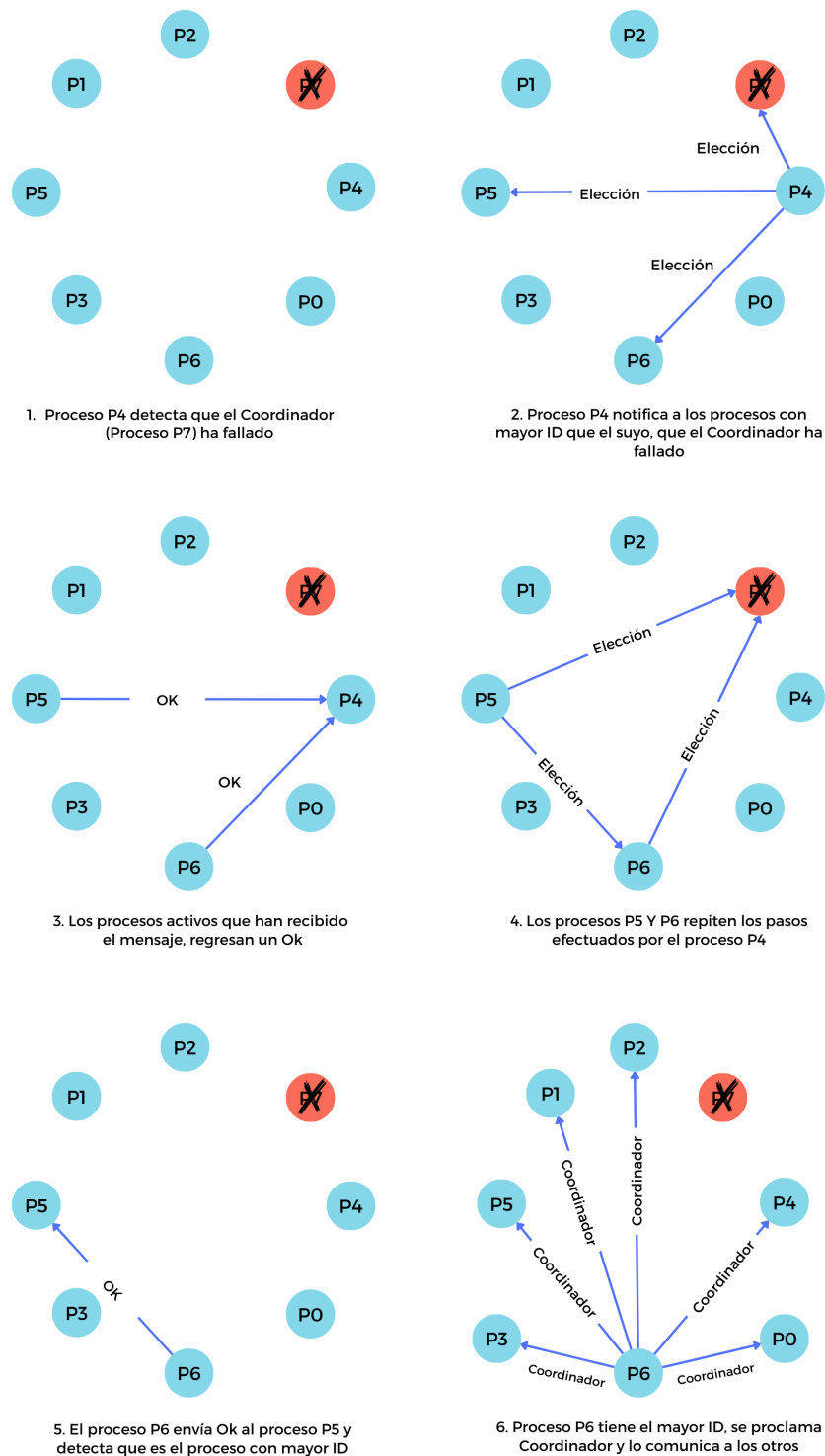


Figure 9.14: Algoritmo de Bully

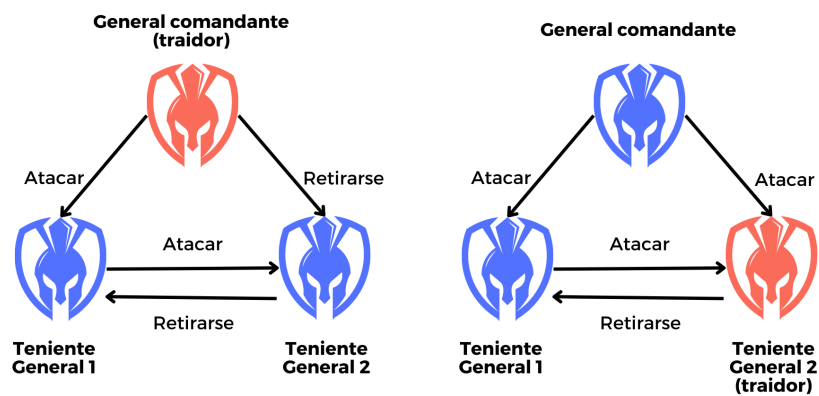


Figure 9.15: Algoritmo de Generales Bizantinos. Caso 1.

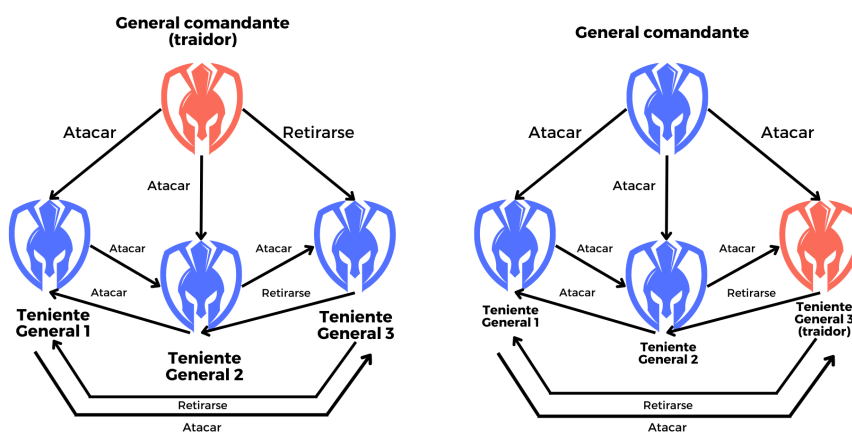


Figure 9.16: Algoritmo de Generales Bizantinos. Caso 2.

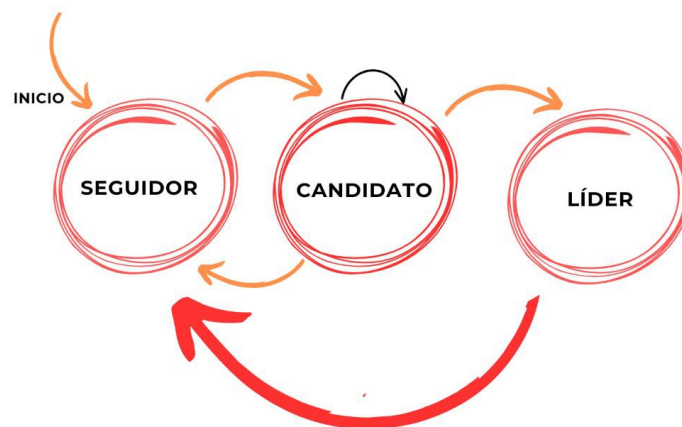
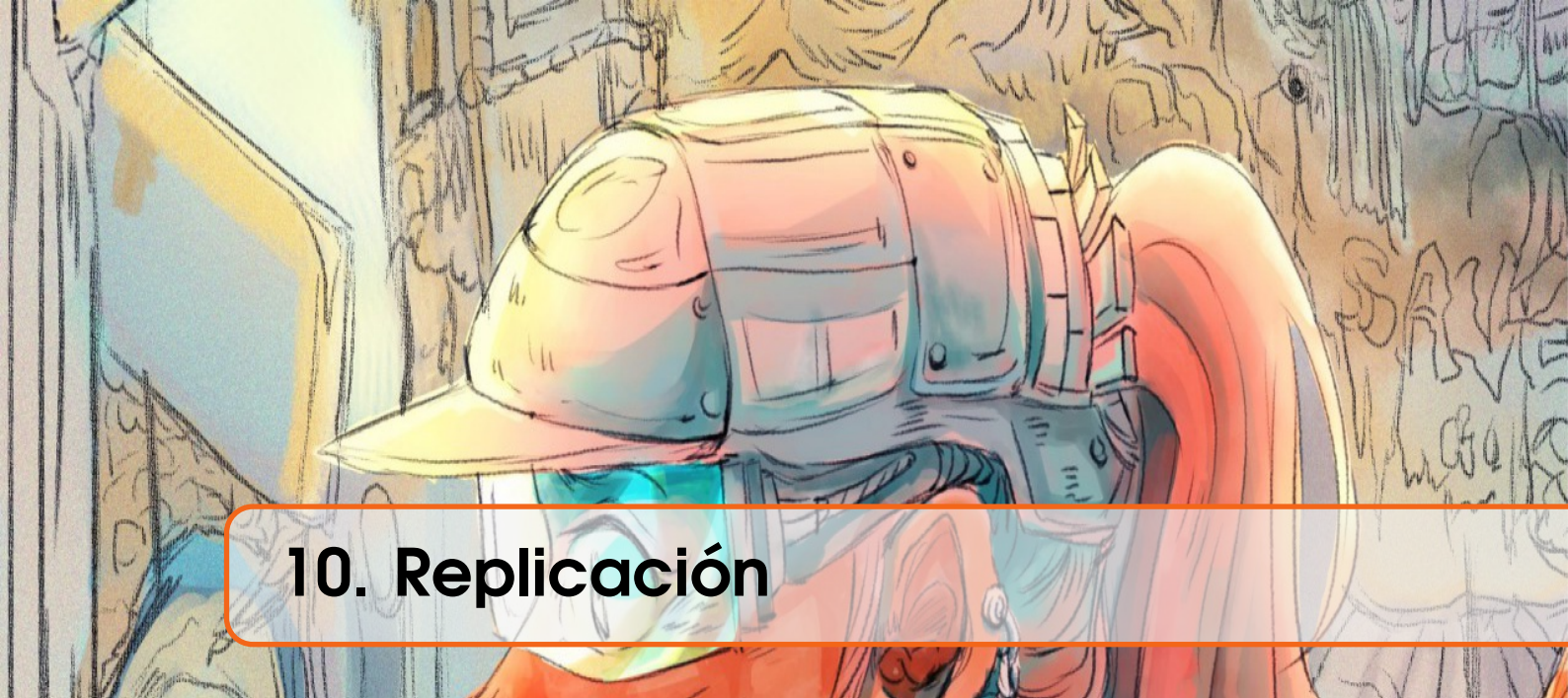


Figure 9.17: Algoritmo de elección del lider Raft.



10. Replicación

10.1 Replicación

Una de las razones para usar la técnica de replicación es aumentar la disponibilidad de servicios. Si algunos datos se almacenan exclusivamente en un solo nodo y ese nodo deja de funcionar, ya no se podrá acceder a los datos. Pero si los datos se replican en su lugar, los clientes pueden cambiar sin problemas a una réplica.

Factores que son relevantes para la alta disponibilidad son la ocurrencia de fallas en los servidores y las particiones de red con desconexión de operaciones (no planificadas y pueden ser un efecto secundario de la movilidad del usuario).

Las particiones de red dificultan la construcción de detectores de fallas, que se utilizan para lograr una multidifusión confiable y totalmente ordenada.

Otra razón para la replicación es aumentar la escalabilidad y el rendimiento; Cuantas más réplicas haya, más clientes podrán acceder a los datos simultáneamente sin sufrir degradaciones de rendimiento. [69] [64] [44]

No obstante, la replicación de datos debe asegurar que los clientes no deben tener en cuenta que existen múltiples copias físicas de datos, quiere decir esto que debe haber transparencia en la replicación. Los datos se organizan como objetos lógicos individuales e identifican solo un elemento en cada caso cuando solicitan que se realice una operación. Se espera que las operaciones devuelvan solo un conjunto de valores.

Las operaciones se realizan sobre una colección de objetos replicados produce resultados que cumplen con la especificación de corrección para esos objetos, es decir, se asegura la consistencia de datos.

Sin embargo, en una operación que ha sufrido de desconexión se puede permitir que los datos se vuelvan inconsistentes, temporalmente. Pero cuando los clientes permanecen conectado, no es aceptable que diferentes clientes (con diferentes copias físicas de datos) obtengan resultados inconsistentes cuando las solicitudes afectan el mismos objetos lógicos.

10.1.1 Administrador de Réplicas

Los Administradores de Réplicas son servidores cuya funcionalidad es la gestión de réplicas de objetos

En los modelos de administradores de réplicas los componentes arquitectónicos se

describen por sus roles y no se especifica la implementación (hardware). Puede aplicarse en un entorno cliente-servidor, (administrador de réplica es un servidor) o se puede aplicar a una aplicación y los procesos de la aplicación (servidores multiprocesos). Ejemplo: computadora portátil del usuario puede contener una aplicación que actúa como un administrador de réplicas para su sitio web. Se asume que cada administrador de réplicas mantiene una réplica de cada objeto

Los modelos administradores de réplicas:

- aplica operaciones a sus réplicas de forma recuperable (una operación en un administrador de réplicas no deja resultados inconsistentes si ocurren fallas)
- Un administrador de réplicas puede ser una máquina de estado [21] [34] Las operaciones en sus réplicas equivale a realizar operaciones en una secuencia estricta. Es decir, el estado de sus réplicas es una función determinista de sus estados iniciales y la secuencia de operaciones que les aplica)
- Se asume que el administrador de réplicas mantiene una réplica de cada objeto. Sin embargo, las réplicas de diferentes objetos pueden ser mantenidas por diferentes conjuntos de administradores de réplica.
- El conjunto de administradores de réplica puede ser estático o dinámico. En un sistema dinámico, pueden aparecer nuevos administradores de réplica (por ejemplo, si una segunda persona copia un sitio web en su computadora portátil).

En la Figura 10.1 se muestra un modelo de arquitectura para el administrador de datos replicados: una colección de administradores de réplicas ofrece un servicio a los clientes. Los clientes solicitan una serie de operaciones o invocaciones sobre los objetos. Las operaciones solicitadas son de solo lectura, o actualización. Las solicitudes de cliente son manejadas primero por el front-end (FE). El rol del FE es comunicarse mediante un mensaje que pasa a los administradores de réplicas. FE es el vehículo para hacer el modelo transparente a la replicación. Se puede implementar un FE en la dirección del espacio del cliente, o puede ser un proceso separado

Un protocolo de replicación [28], ver Figura 10.2, es un modelo abstracto que sirve como base para describir los modelos de replicación. Este protocolo se describe en una secuencia de cinco pasos. :

1. Solicitud: FE emite la solicitud a uno o más administradores de réplica:
 - FE se comunica con un solo administrador de réplica.
 - o FE difunde usando multidifusión, la solicitud al resto de administradores de réplica.
2. Coordinación: administradores de réplica se coordinan para ejecutar la solicitud de manera consistente. Acuerda, si la solicitud se aplica o decide sobre la misma. El orden de los mensajes pueden ser:
 - FIFO: si un FE emite la solicitud r antes que la solicitud r' , los administradores de réplica maneja r' después de resolver r .
 - Orden causal: si $r \rightarrow r'$, los administradores de réplica resuelven r antes de que r' .
 - Orden total: administrador de réplica maneja r antes de la solicitud r' , entonces otro administrador de réplica maneja r' después de r .
3. Ejecución: administrador de réplica ejecutan la solicitud, de manera que puedan deshacer sus efectos más tarde.
 - Acuerdo: los administradores de réplica llegan a consenso sobre el efecto de la solicitud, si la hay, se compromete.

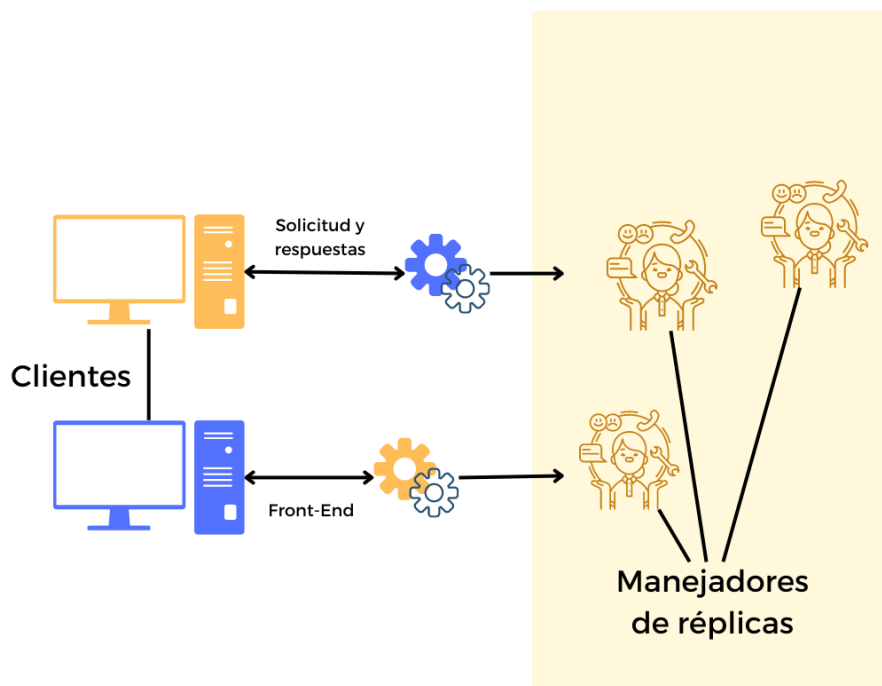


Figure 10.1: Arquitectura del administrador de réplicas de datos

- Respuesta: Uno o más administrador de réplica responden al FE. Un administrador de réplica envía la respuesta. O FE recibe respuestas de una colección de administrador de réplica y selecciona o una respuesta única a pasar al cliente.

10.2 Tolerancia a Fallas

Los servicios tolerantes a fallos son aquellos servicios que ante la presencia de fallos cumplen la especificación o responde correctamente. Obtener respuesta suele ser parte de la especificación (disponibilidad). Por su repetición, los fallos pueden ser: transitorios, intermitentes o permanentes. En cuanto a los fallos, aquellos relacionados con procesos (ver en tabla 1.3 en capítulo 1, estos se pueden clasificar de acuerdo a su tipo en: fallos de bloqueo, fallos de parada o fallas bizantinas

Para construir servicios tolerantes a fallos se basa en ofrecer redundancia en:

- De información (datos replicados, códigos redundantes, CRC).
- Temporal (retransmisiones, repetición de transacciones abortadas).
- Física (componentes, máquinas, conexiones replicados).
- De información y física combinadas (discos espejo, RAID).

La redundancia puede introducir a su vez fallos (respuestas incorrectas) en los servicios. Entonces, ¿Cómo proporcionar un servicio que sea correcto a pesar de la ocurrencia de fallas en el proceso?

Existen varios criterios de corrección para objetos replicados. Algunos de estos criterios de consistencia son la Linealizabilidad, Consistencia secuencial y Consistencias débiles.

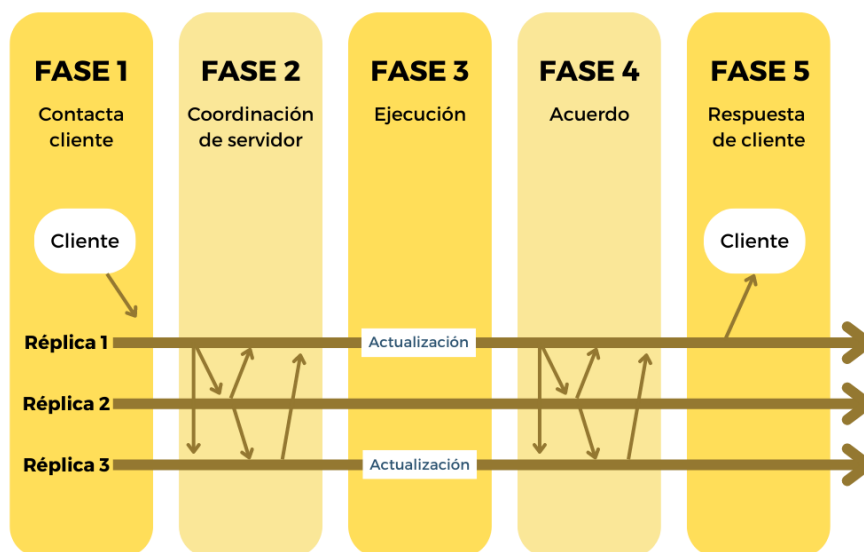


Figure 10.2: Protocolo de una solicitud. Tomado de [28]

10.2.1 Linealizabilidad y Consistencia Secuencial

Linealizabilidad Los sistemas más estrictamente correctos son linealizables y esta propiedad se denomina linealizabilidad (*linearizability*).

La linealizabilidad es la forma más fuerte de consistencia. Esto significa que todas las operaciones se ejecutan como si se ejecutaran en una sola máquina, a pesar de que los datos se distribuyen en múltiples réplicas. Como resultado, cada operación devuelve un valor actualizado.

Un servicio de objetos compartidos replicados es lineizable si para cualquier ejecución existe algún entrelazamiento de las series de operaciones emprendidas por cada cliente que satisfaga:

- La secuencia entrelazada de operaciones cumple la especificación de una (única) copia correcta de los objetos.
- El orden de las operaciones del entrelazamiento es consistente con los tiempos reales en los cuales ocurrieron las operaciones en la ejecución real.

Ejemplo de Linealizabilidad En el siguiente ejemplo de la Figura 10.3, es el esquema de procesos en un sistema distribuido con tres procesos diferentes: P_1 , P_2 y P_3 . El cliente A escribe un valor 1 en un objeto A. Dado que el sistema es lineizable, una vez que se completa la operación de escritura, todas las lecturas posteriores deben devolver el valor de esa escritura o el valor de la operación de escritura posterior.

Entonces, cuando el cliente B lee el valor de A, el resultado es 1. Una vez que una operación de lectura devuelve un valor particular, todas las lecturas posteriores deben devolver ese valor o el valor de la operación de escritura posterior.

Consistencia secuencial La consistencia secuencial es una fuerte propiedad de seguridad para los sistemas concurrentes. Se define [20] como el resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesadores se ejecutaran en

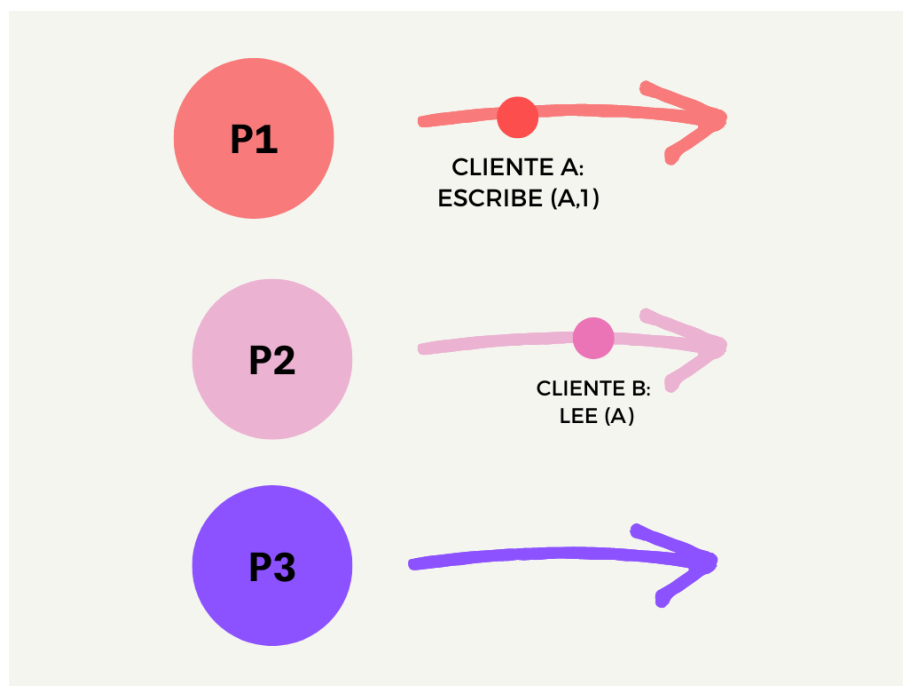


Figure 10.3: Ejemplo de Linealizabilidad

algún orden secuencial, y las operaciones de cada procesador individual aparecen en esta secuencia en el orden especificado por su programa.

Un servicio es consistente secuencialmente si para alguna de las posibles secuencias de operaciones intercaladas:

- El orden de las operaciones en el intercalado es consistente con el orden de programa que ejecuta cada cliente individual
- Dicho orden en serie ‘real’ es consistente con las copias u operaciones realizadas
- Cualquier servicio linealizable es consistente secuencialmente, pues el orden real de ejecución refleja el orden del programa. Lo contrario no es cierto

El resultado de cualquier ejecución es el mismo que si las operaciones (lectura y escritura) de todos los procesos en el almacén de datos fueron ejecutados en algún orden secuencial y las operaciones de cada proceso individual aparecen en esta secuencia en el orden especificado por su programa.

Existen varios modelos o técnicas de replicación tolerantes a fallas. A continuación se presentan dos de ellas:

10.2.2 Modelo de Replicación Pasiva

En el modelo de replicación pasiva [28], también modelo de replicación de Respaldo Primario con tolerancia a fallas, ver Figura 10.4, el modelo tiene un único administrador de réplica primarias y uno o más administrador de réplica secundarios, también nominados como copias de seguridad o esclavos. FE se comunican solo con el administrador de réplica principal para obtener el servicio. El administrador de réplica principal ejecuta las operaciones y envía copias de los datos actualizados a copias de seguridad. Si el primario falla, una de las copias de seguridad se modifica para actuar como el primario.

La secuencia de eventos cuando un cliente solicita que se realice una operación:

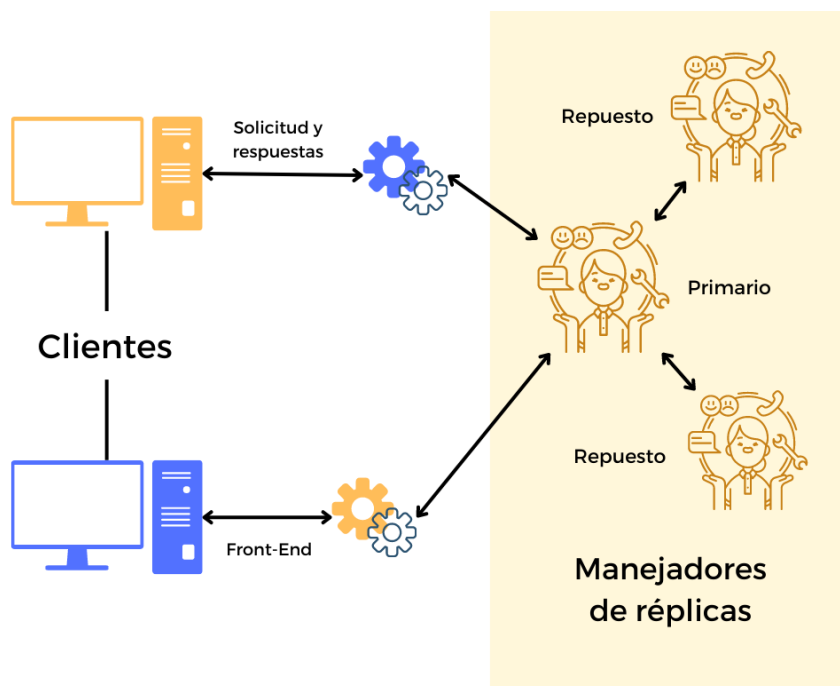


Figure 10.4: Modelo de Replicación Pasiva. Adaptado de [54]

- **Solicitud:** FE emite la solicitud, que contiene un ID único, al administrador de réplica principal.
- **Coordinación:** el servidor primario toma cada solicitud atómicamente, en el orden de su recepción. Comprueba el ID único, en caso de que ya haya ejecutado la solicitud, y si es así, reenvía la respuesta.
- **Ejecución:** el servidor primario ejecuta la solicitud y almacena la respuesta.
- **Acuerdo:** si la solicitud es una actualización, el servidor primario envía el estado actualizado, la respuesta y el ID único a todas las copias de seguridad. Las copias de seguridad envían un acuse de recibo.
- **Respuesta:** El servidor primario responde al FE, que devuelve la respuesta al cliente.

El sistema implementa la linealización si el servidor primario es correcto, el servidor primario secuencía las operaciones sobre los objetos compartidos. Si el servidor primario falla, entonces el sistema conserva la capacidad de linealización si una única copia de seguridad se convierte en el nuevo servidor primario y si la nueva configuración del sistema toma el control exactamente donde quedó el último. El servidor primario se reemplaza por una copia de seguridad. Los administradores de réplica sobrevivientes acuerdan qué operaciones se realizaron en el momento en que se hace cargo el reemplazo del servidor primario.

10.2.3 Modelo de Replicación Activa

En los modelos de replicación activa [54], los administradores de réplica son máquinas de estado que juegan roles equivalentes y están organizados como un grupo. La Figura 10.5 se plasma la arquitectura del modelo de replicación activa. Los servidores FE distribuyen sus solicitudes al grupo de administrador de réplica y todos procesan la solicitud de forma independiente pero idéntica y responden a cada una de ellas. Si un administrador de réplica

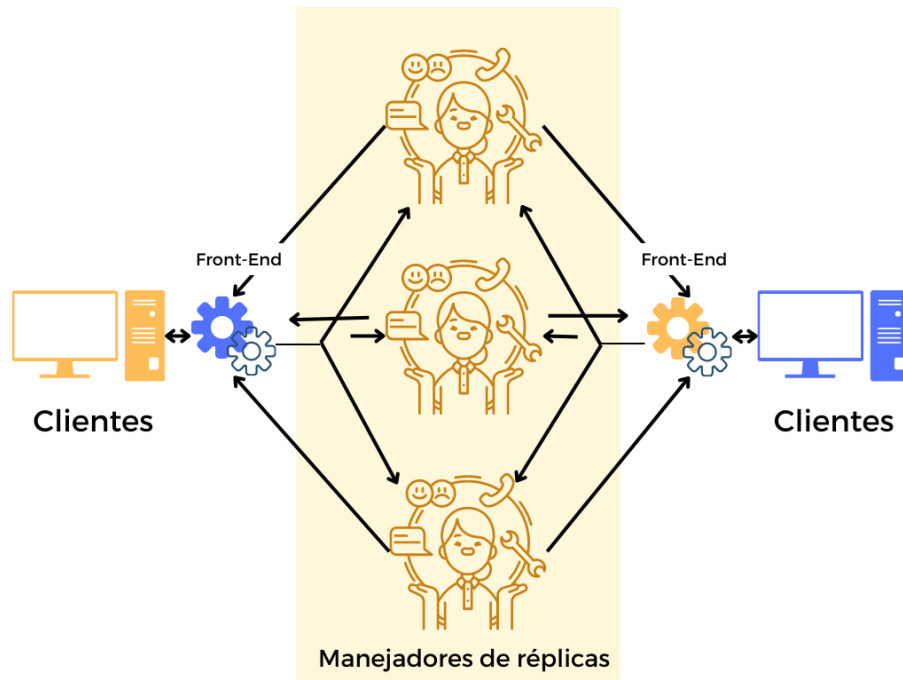


Figure 10.5: Modelo de Arquitectura de Replicación Activa. Adaptado de [54]

falla, esto no afecta el rendimiento del servicio, ya que los administradores de réplica restantes continúan respondiendo de la manera normal. Este modelo es tolerante a las fallas bizantinas, porque el FE puede recopilar y comparar las respuestas que recibe.

La secuencia de operación en el modelo:

- **Solicitud:** el FE adjunta un ID único a la solicitud y lo transmite al grupo de administrador de réplica, utilizando una primitiva de multidifusión confiable y totalmente ordenada. Se supone que si el FE falla al estrellarse en el peor de los casos. No emite la siguiente solicitud hasta que recibe una respuesta.
- **Coordinación:** el sistema de comunicación grupal entrega la solicitud a cada administrador de réplica correcto en el mismo orden (total)
- **Ejecución:** cada administrador de réplica ejecuta la solicitud. Como son máquinas de estado y las solicitudes se entregan en el mismo orden total, los AR correctos procesan la solicitud de manera idéntica. La respuesta contiene el ID de solicitud del cliente.
- **Acuerdo:** no se necesita una fase de acuerdo, debido a la semántica de entrega de multidifusión.
- **Respuesta:** Cada administrador de réplica envía su respuesta al FE. El número de respuestas que recopila el FE depende de los supuestos de falla y el algoritmo de multidifusión

El modelo de replicación activa logra consistencia secuencial:

- Todos los administrador de réplica procesan la misma secuencia de solicitudes. La confiabilidad de la multidifusión asegura que cada administrador de réplica procesa el mismo conjunto de solicitudes y el orden total asegura que las procesen en el mismo orden.
- Como son máquinas de estado, todas terminan con el mismo estado que las demás

después de cada solicitud. Las solicitudes de cada FE se atienden en Orden FIFO (FE espera una respuesta antes de hacer la siguiente solicitud).

- Si los clientes no se comunican con otros clientes mientras esperan respuestas a sus solicitudes, entonces sus solicitudes se procesan en el orden anterior.
- Por otra parte, si los clientes son multiproceso y pueden comunicarse entre sí mientras esperan respuestas del servicio. Para garantizar el procesamiento de la solicitud en el orden anterior, tendríamos que reemplazar la multidifusión por una que esté causal y totalmente ordenada.
- El sistema de replicación activa no logra linealización. Esto se debe a que el orden total en el que los administradores de réplicas procesan las solicitudes no es necesariamente el mismo que el orden en tiempo real en el que los clientes hicieron sus solicitudes.

10.3 Caso de Estudio. Fragmentación

La fragmentación [32] es una técnica ampliamente utilizada para escalar horizontalmente sistemas de almacenamiento y caché, y abordar los cuellos de botella en la capacidad de procesamiento y almacenamiento. Según esta técnica, un gran conjunto de elementos se divide en un conjunto de segmentos, denominados fragmentos, de acuerdo al resultado de una función hash calculada en el identificador del objeto. Luego, cada fragmento se asigna a un dispositivo de almacenamiento físico o de caché.

La fragmentación es un patrón de arquitectura de base de datos [76] que divide una única base de datos en tablas más pequeñas conocidas como fragmentos, cada una almacenada en un nodo independiente. Cada partición de base de datos se conoce como fragmento lógico y su almacenamiento dentro de un nodo se conoce como fragmento físico. Esta técnica permite dividir datos entre los miembros de un clúster e identificar el miembro del clúster responsable de un elemento determinado simplemente calculando una función hash. La fragmentación se usa ampliamente en una variedad de aplicaciones como en sistemas de bases de datos, cachés web en redes empresariales y almacenes clave-valor.

10.3.0.1 ¿Fragmentación, partición o replicación?

¿Cuál es la diferencia entre fragmentación y partición? Fragmentación, partición y replicación son conceptos similares, pero con diferencias importantes entre ellos. De hecho, la fragmentación puede considerarse una clase especial de partición.

El particionamiento se define como cualquier división de una base de datos en partes distintas, generalmente por motivos como un mejor rendimiento y facilidad de gestión [76] [71]. Dependiendo de la situación, se puede utilizar particiones horizontales o verticales:

- Partición horizontal: cada partición utiliza el mismo esquema de base de datos y tiene las mismas columnas, pero contiene filas diferentes.
- Partición vertical: cada partición es un subconjunto adecuado del esquema de base de datos original, es decir, contiene todas las filas, pero solo un subconjunto de las columnas originales.

La fragmentación suele ser un caso de partición horizontal. Existen varios esquemas de fragmentación posibles para determinar cómo particionar los datos en una base de datos:

- Fragmentación basada en rango: la base de datos se fragmenta en función de un valor determinado, como el nombre o el número de identificación. Por ejemplo, una base de datos de estudiantes universitarios puede fragmentarse según la primera letra

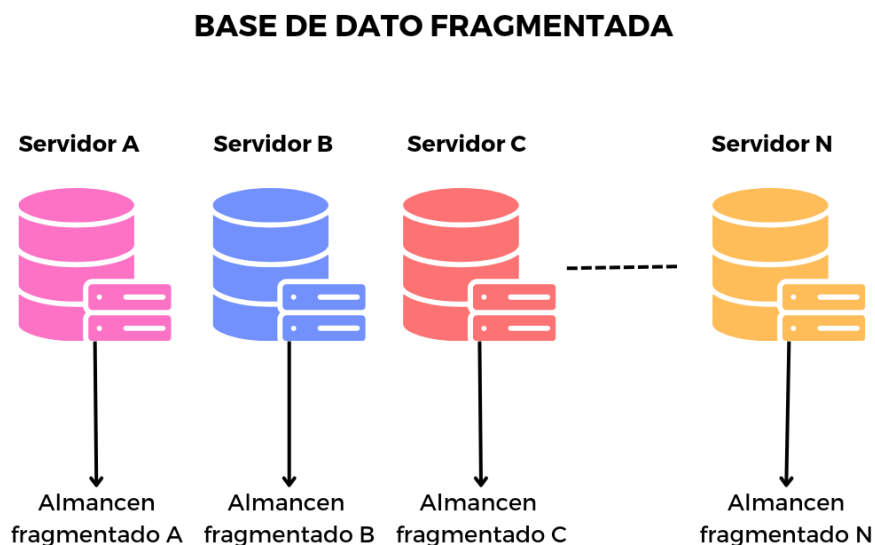


Figure 10.6: Fragmentación de base de datos

de su apellido.

- Fragmentación basada en hash: esta técnica se utiliza para bases de datos clave-valor. La clave de cada entrada de la base de datos se pasa a través de una función hash, que genera un resultado que determina a qué fragmento se asignará la entrada.

Un ejemplo de una base de datos fragmentada se muestra en la Figura 10.6.

Mientras la replicación es el término de hacer una copia de la información de una base de datos en otra ubicación. La fragmentación y la replicación son estrategias separadas, pero complementarias, para mejorar la disponibilidad de la base de datos.

10.3.0.2 Almacenamiento en caché fragmentado

Como ejemplo de un sistema fragmentado, esta sección proporciona una presentación diseño de un sistema de almacenamiento en caché fragmentado. Un caché fragmentado [50] es un almacenamiento caché que se encuentra entre las solicitudes del usuario y la implementación real del frontend. En la Figura 10.7 se muestra la arquitectura de un cache fragmentado.

¿ Por qué se necesita fragmentar el caché ? La razón principal para fragmentar cualquier servicio es aumentar el tamaño de los datos que se almacenan en el servicio. Para comprender cómo ayuda esto a un sistema de almacenamiento en caché, imagine el siguiente sistema [50]: cada caché tiene 10 GB de RAM disponibles para almacenar resultados y puede atender 100 solicitudes por segundo (RPS). Supongamos que nuestro servicio tiene un total de 200 GB de resultados posibles que podrían devolverse y un RPS esperado de 1000. Esto puede implementarse mediante replicación con 10 réplicas del caché para satisfacer 1000 RPS (10 réplicas × 100 solicitudes por segundo por réplica). Esta implementación contendría un máximo del 5% (10 GB/200 GB) del conjunto de datos total que estamos sirviendo. Usando un caché fragmentado de 10 vías, aún podemos servir la cantidad adecuada de RPS (10 × 100 sigue siendo 1000), pero debido a que cada caché

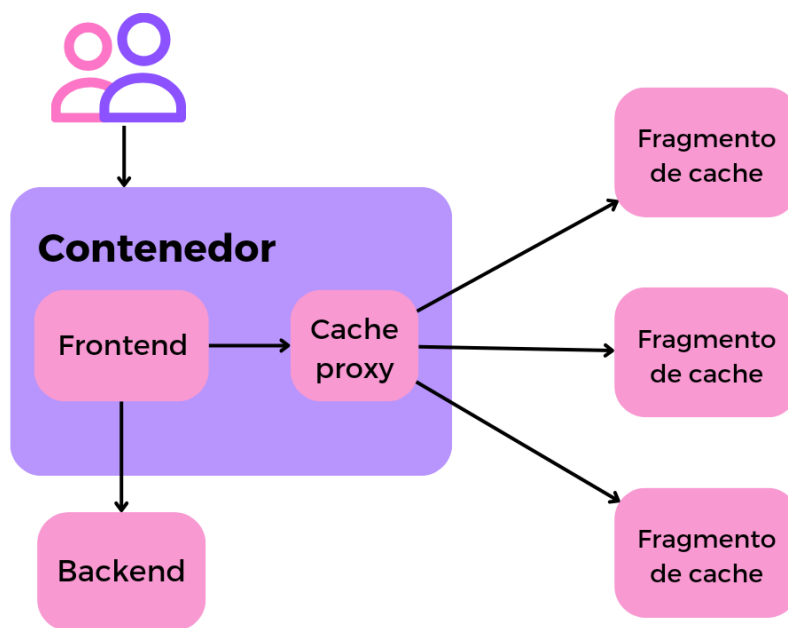


Figure 10.7: Fragmentación de datos

sirve un conjunto de datos completamente único, podemos aumentar el almacenamiento al 50% ($10 \times 10 \text{ GB}/200 \text{ GB}$) del conjunto de datos total.

Beneficios de utilizar caché fragmentado A continuación se detallan algunos de sus beneficios:

- Uso de caché más eficiente. Sólo se necesita una copia de un archivo para todos los servidores de los centros de datos.
- Reduce la carga en el servidor de origen. Los servidores de una red de distribución de contenido utilizan la clave hash para determinar si hay contenido en la caché del grupo. El origen solo se solicita si el grupo no tiene ningún contenido almacenado en caché. Esto reduce los costos en los que se incurre al pagar por el tráfico desde los servidores de la red hasta el origen.
- Velocidad de entrega de contenido mejorada. Verificar las cachés de los servidores cercanos en un grupo en busca de un archivo es más rápido que enrutar una solicitud al origen.

Caché fragmentado y replicado Un servicio fragmentado y replicado combina el patrón de servicio replicado con el patrón fragmentado. En lugar de que un único servidor implemente cada fragmento de la caché, se utiliza un servicio replicado para implementar cada fragmento de la caché. En la Figura 10.8 se muestra un esquema de una base de datos de clientes fragmentada por la inicial de su nombre. Cada fragmento tiene dos réplicas. Note que esta distribución puede cambiar de acuerdo a las necesidades del usuario.

Este diseño es más complicado de implementar, pero tiene varias ventajas sobre un servicio fragmentado simple. Lo más importante es que, al reemplazar un único servidor con un servicio replicado, cada fragmento de caché es resistente a las fallas y siempre está presente durante la ocurrencia de fallas. Además se espera una mejora del rendimiento que proporciona la memoria caché.

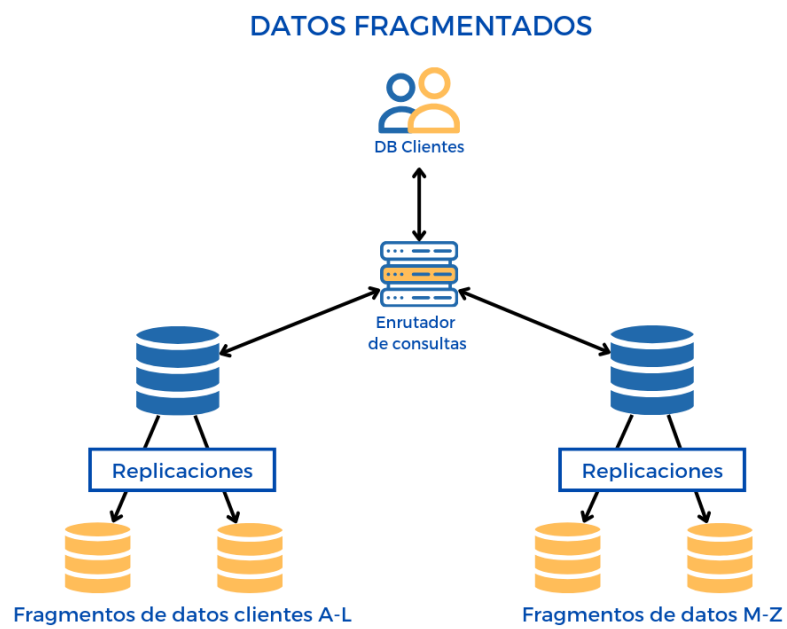


Figure 10.8: Caché fragmentado y replicado

Bibliografía

Artículos

- [1] Portal Azure. “Publisher-Subscriber pattern”. In: (2023). Documento Electrónico. URL: <https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber> (cited on page 123).
- [2] S. A. Baset and H. G. Schulzrinne. “An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol”. In: (Apr. 2006), pages 1–11. ISSN: 0743-166X. DOI: [10.1109/INFOCOM.2006.312](https://doi.org/10.1109/INFOCOM.2006.312) (cited on page 85).
- [3] Elizabeth A. Basha, Sai Ravela, and Daniela Rus. “Model-Based Monitoring for Early Warning Flood Detection”. In: SenSys ’08 (2008). <https://doi.org/10.1145/1460412.1460442>, pages 295–308. DOI: [10.1145/1460412.1460442](https://doi.org/10.1145/1460412.1460442). URL: <https://doi.org/10.1145/1460412.1460442> (cited on page 49).
- [4] Philip A. Bernstein and Nathan Goodman. “Serializability Theory for Replicated Databases”. In: *J. Comput. Syst. Sci.* 31 (1985), pages 355–374. URL: <https://api.semanticscholar.org/CorpusID:29626206> (cited on page 154).
- [5] Eric A. Brewer. “Towards Robust Distributed Systems”. In: PODC ’00 (2000), pages 7–. DOI: [10.1145/343477.343502](https://doi.org/10.1145/343477.343502). URL: <http://doi.acm.org/10.1145/343477.343502> (cited on page 24).
- [6] Ernest Chang and Rosemary Roberts. “An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes”. In: *Commun. ACM* 22.5 (May 1979), pages 281–283. ISSN: 0001-0782. DOI: [10.1145/359104.359108](https://doi.org/10.1145/359104.359108). URL: <https://doi.org/10.1145/359104.359108> (cited on page 182).
- [7] Ian Clarke et al. “Freenet: A Distributed Anonymous Information Storage and Retrieval System”. In: (2001). Edited by Hannes Federrath, pages 46–66. DOI: [10.1007/3-540-44702-4_4](https://doi.org/10.1007/3-540-44702-4_4). URL: https://doi.org/10.1007/3-540-44702-4_4 (cited on page 155).
- [8] B. Cohen. “Incentives to Build Robustness in BitTorrent”. In: *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley* (2003). Berkeley, pages 68–72. (Cited on page 155).
- [9] Corba. “Common Object Request Broker Architecture”. In: (Dec. 2022). Documento electrónico (cited on page 51).
- [10] Flaviu Cristian. “Probabilistic Clock Synchronization”. In: *Distrib. Comput.* 3.3 (Sept. 1989), pages 146–158. ISSN: 0178-2770. DOI: [10.1007/BF01784024](https://doi.org/10.1007/BF01784024). URL: <https://doi.org/10.1007/BF01784024> (cited on page 169).
- [11] Jörg Eberspächer and Rüdiger Schollmeier. “3. Past and Future”. In: (2005). Edited by Ralf Steinmetz and Klaus Wehrle, pages 17–23. DOI: [10.1007/11530657_3](https://doi.org/10.1007/11530657_3). URL: https://doi.org/10.1007/11530657_3 (cited on page 155).

- [12] Jörg Eberspächer and Rüdiger Schollmeier. “5. First and Second Generation of Peer-to-Peer Systems”. In: (2005). Edited by Ralf Steinmetz and Klaus Wehrle, pages 35–56. DOI: [10.1007/11530657_5](https://doi.org/10.1007/11530657_5). URL: https://doi.org/10.1007/11530657_5 (cited on pages 155, 156).
- [13] Mohamed Salem Elshebani. “Communication paradigms in distributed systems infrastructure”. In: (2009). DOI: [10.1109/TELSKS.2009.5339464](https://doi.org/10.1109/TELSKS.2009.5339464) (cited on page 54).
- [14] C. Fidge. “Logical Time in Distributed Computing Systems”. In: *Computer* 24.08 (Aug. 1991), pages 28–33. ISSN: 1558-0814. DOI: [10.1109/2.84874](https://doi.org/10.1109/2.84874) (cited on page 175).
- [15] Jacob Gabrielson. “Los desafíos de los sistemas distribuidos”. In: *Amazon Web Services, Inc.* (2019). URL: <https://aws.amazon.com/es/builders-library/challenges-with-distributed-systems/> (cited on page 30).
- [16] Garcia-Molina. “Elections in a Distributed Computing System”. In: *IEEE Transactions on Computers* C-31.1 (1982), pages 48–59. DOI: [10.1109/TC.1982.1675885](https://doi.org/10.1109/TC.1982.1675885) (cited on page 183).
- [17] Stefan Götz, Simon Rieche, and Klaus Wehrle. “8. Selected DHT Algorithms”. In: (2005). Edited by Ralf Steinmetz and Klaus Wehrle, pages 95–117. DOI: [10.1007/11530657_8](https://doi.org/10.1007/11530657_8). URL: https://doi.org/10.1007/11530657_8 (cited on pages 157, 159, 160).
- [18] Riccardo Gusella and Stefano Zatti. “The Berkeley UNIX 4.3BSD Time Synchronization Protocol”. In: (1985) (cited on page 170).
- [19] Ingrid Van Den Hoogen. “Deutsch’s Fallacies, 10 Years After”. In: (2004). Documento electrónico (cited on page 32).
- [20] Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (Sept. 1979), pages 690–691. ISSN: 1557-9956. DOI: [10.1109/TC.1979.1675439](https://doi.org/10.1109/TC.1979.1675439) (cited on page 194).
- [21] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pages 558–565. ISSN: 0001-0782. DOI: [10.1145/359545.359563](https://doi.org/10.1145/359545.359563). URL: <https://doi.org/10.1145/359545.359563> (cited on pages 172, 178, 192).
- [22] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems* (July 1982), pages 382–401. URL: <https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/> (cited on page 184).
- [23] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. “Deconstructing the Kazaa network”. In: (June 2003), pages 112–120. DOI: [10.1109/WIAPP.2003.1210295](https://doi.org/10.1109/WIAPP.2003.1210295) (cited on pages 85, 155).
- [24] Friedemann Mattern. “Virtual Time and Global States of Distributed Systems.” In: *Proceedings of the International Workshop on Parallel and Distributed Algorithms* (1989), pages 215–226 (cited on page 175).

- [25] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: (2002). Edited by Peter Druschel, Frans Kaashoek, and Antony Rowstron, pages 53–65 (cited on page 155).
- [26] D.L. Mills. “Improved algorithms for synchronizing computer network clocks”. In: *IEEE/ACM Transactions on Networking* 3.3 (1995), pages 245–254. DOI: [10.1109/90.392384](https://doi.org/10.1109/90.392384) (cited on page 171).
- [27] M.P. Papazoglou. “Service-oriented computing: concepts, characteristics and directions”. In: (2003), pages 3–12. DOI: [10.1109/WISE.2003.1254461](https://doi.org/10.1109/WISE.2003.1254461) (cited on page 125).
- [28] Fernando Pedone et al. “Understanding Replication in Databases and Distributed Systems”. In: *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS’2000)* (Jan. 2000), pages 464–474. DOI: [10.1109/ICDCS.2000.840959](https://doi.org/10.1109/ICDCS.2000.840959) (cited on pages 192, 194, 195).
- [29] Sylvia Ratnasamy et al. “A Scalable Content-Addressable Network”. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pages 161–172. ISSN: 0146-4833. DOI: [10.1145/964723.383072](https://doi.org/10.1145/964723.383072). URL: <https://doi.org/10.1145/964723.383072> (cited on page 155).
- [30] Glenn Ricart and Ashok K. Agrawala. “An Optimal Algorithm for Mutual Exclusion in Computer Networks”. In: *Commun. ACM* 24.1 (Jan. 1981), pages 9–17. ISSN: 0001-0782. DOI: [10.1145/358527.358537](https://doi.org/10.1145/358527.358537). URL: <https://doi.org/10.1145/358527.358537> (cited on page 180).
- [31] Antony Ian Taylor Rowstron and Peter Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: (2001). URL: <https://api.semanticscholar.org/CorpusID:2200101> (cited on pages 155, 160).
- [32] Lorenzo Saino, Ioannis Psaras, and George Pavlou. “Understanding sharded caching systems”. In: (Apr. 2016), pages 1–9. DOI: [10.1109/INFOCOM.2016.7524442](https://doi.org/10.1109/INFOCOM.2016.7524442) (cited on page 198).
- [33] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. “Earthquake shakes Twitter users: real-time event detection by social sensors”. In: (2010) (cited on page 49).
- [34] F.B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial.” In: *ACM Computing Surveys (CSUR)* 22 (1990), pages 299–319. DOI: <https://doi.org/10.1145/98163.98167> (cited on page 192).
- [35] Abhinav Singh. “System Design Basics: Pub/Sub Messaging”. In: (Feb. 2022). URL: <https://medium.com/geekculture/system-design-basics-pub-sub-messaging-88dfd98e67b7> (cited on page 122).
- [36] J.E. Smith and Ravi Nair. “The architecture of virtual machines”. In: *Computer* 38.5 (2005), pages 32–38. DOI: [10.1109/MC.2005.173](https://doi.org/10.1109/MC.2005.173) (cited on pages 81, 82).
- [37] Ralf Steinmetz and Klaus Wehrle. “What Is This “Peer-to-Peer” About?” In: (2005). Edited by Ralf Steinmetz and Klaus Wehrle, pages 9–16. DOI: [10.1007/11530657_2](https://doi.org/10.1007/11530657_2). URL: https://doi.org/10.1007/11530657_2 (cited on pages 153, 154).

- [38] Ion Stoica et al. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pages 149–160. ISSN: 0146-4833. DOI: [10.1145/964723.383071](https://doi.org/10.1145/964723.383071). URL: <https://doi.org/10.1145/964723.383071> (cited on pages 50, 155, 157).
- [39] Ichiro Suzuki and Tadao Kasami. “A Distributed Mutual Exclusion Algorithm”. In: *ACM Trans. Comput. Syst.* 3.4 (Nov. 1985), pages 344–349. ISSN: 0734-2071. DOI: [10.1145/6110.214406](https://doi.org/10.1145/6110.214406). URL: <https://doi.org/10.1145/6110.214406> (cited on page 180).
- [40] W3C. “World Wide Web Consortium.” In: (2022). Documento electrónico. URL: <http://www.w3.org> (cited on pages 34, 38, 54, 126, 129–131, 133, 135–138, 140, 141).
- [41] Klaus Wehrle, Stefan Götz, and Simon Rieche. “7. Distributed Hash Tables”. In: (2005). Edited by Ralf Steinmetz and Klaus Wehrle, pages 79–93. DOI: [10.1007/11530657_7](https://doi.org/10.1007/11530657_7). URL: https://doi.org/10.1007/11530657_7 (cited on pages 155, 157).
- [42] Alex Xu. “What are the most common misconceptions about distributed environments?, [@alexxubyte], Twitter”. In: (Dec. 2022). Twitter. Documento electrónico. URL: <https://twitter.com/alexxubyte/status/1603067499674681345> (cited on pages 32, 33, 95).
- [43] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. “Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and”. In: (2001). URL: <https://api.semanticscholar.org/CorpusID:221325581> (cited on page 155).

Libros

- [44] Bharat B. Bhargava Abdelsalam A. Helal Abdelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. 1st edition. Advances in Database Systems. Springer, 1996. ISBN: 0792398009,9780792398004,9780306477966. URL: <http://gen.lib.rus.ec/book/index.php?md5=6b44cd5d499c8066229dcfcb527c397b> (cited on page 191).
- [45] K. Scott Allen. *What Every Web Developer Should Know About HTTP*. 2nd edition. OdeToCode LLC, 2017. URL: <http://gen.lib.rus.ec/book/index.php?md5=52691194FAEE6930B9675E5ACA7F93F1> (cited on page 34).
- [46] Ryan Asleson and Nathaniel T. Schutta. *Foundations of Ajax*. 1st edition. Apress, 2005. ISBN: 9781590595824,1590595823. URL: <http://gen.lib.rus.ec/book/index.php?md5=3ce4d87c6feab07e8e784f8723017b05> (cited on pages 44, 46).
- [47] Eve Banks Alex;Porcello. *Learning GraphQL: declarative data fetching for modern web apps*. First edition. O’Reilly Media, Inc., 2018. ISBN: 9781492030713,9781492030683,1492030683. URL: <http://gen.lib.rus.ec/book/index.php?md5=A290D328453092A8A31F8E62DF6D38FB> (cited on page 108).
- [48] Douglas K. Barry. *Web services, service-oriented architectures, and cloud computing: the savvy manager’s guide*. 2nd edition. The Savvy Manager’s Guides. Morgan Kaufmann, 2013. ISBN: 0123983576,9780123983572,0-12-407200-3,978-0-12-407200-8. URL: <http://gen.lib.rus.ec/book/index.php?md5=1a78c893769a3591f91335f8f3d0ec2d> (cited on page 141).

- [49] Samer Buna. *GraphQL in Action*. 1st edition. Manning Publications, 2021. ISBN: 161729568X,9781617295683. URL: <http://gen.lib.rus.ec/book/index.php?md5=6F9B5661E34A6183CEDB2534D06DA9F9> (cited on page 108).
- [50] Brendan Burns. *Designing Distributed Systems*. O'Reilly Media, 2018. URL: <http://gen.lib.rus.ec/book/index.php?md5=58458334e614eda7835abd93beec2bb5> (cited on page 199).
- [51] Buyya. *Mastering Cloud Computing*. Mc Graw Hill India, 2013. ISBN: 9781259029950. URL: <http://gen.lib.rus.ec/book/index.php?md5=5B05F6B37AA93B2A0709774ABBD9971> (cited on page 125).
- [52] Szyperski C. and Murer S. *Component Software: Beyond Object-Oriented Programming*. 2nd edition. Addison-Wesley Professional, 2002. ISBN: 0-201-67520-X. URL: <http://gen.lib.rus.ec/book/index.php?md5=5e0ed0adba7de052286a916afb2b8fa> (cited on page 51).
- [53] Douglas Comer. *Internetworking with TCP/IP Volume 1*. Sixth edition, Pearson New International Edition. Always learning. Pearson Education Limited, 2014. ISBN: 1292040815,1269374508,9781292040813,9781269374507,9781292056234,1292056231. URL: <http://gen.lib.rus.ec/book/index.php?md5=8F450B533F4862B1C6C4090563E66671> (cited on pages 87, 90).
- [54] George Coulouris et al. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011 (cited on pages 19, 21, 22, 30, 32, 34, 47, 50, 53, 57, 68, 73, 81, 84, 92, 97, 101, 102, 105, 111–114, 116, 118–121, 129, 131, 155, 161, 163, 167, 177, 196, 197).
- [55] Dave Crane, Eric Pascarello, and Darren James. *Ajax in Action*. 1st edition. Manning Publications, 2005. ISBN: 9781932394610,1-932394-61-3. URL: <http://gen.lib.rus.ec/book/index.php?md5=b745e2a62837bf1ec6779dc47ce6c862> (cited on pages 44, 45).
- [56] Ludwik Czaja. *Introduction to Distributed Computer Systems. Principles and Features*. Springer, 2018. ISBN: 978-3-319-72023-4. URL: <http://gen.lib.rus.ec/book/index.php?md5=69aac00014fb447523f232026c0e2df2> (cited on pages 22, 167, 172, 176).
- [57] Ganesh Chandra Deka. *NoSQL: Database for Storage and Retrieval of Data in Cloud*. CRC Press, 2017. URL: <http://gen.lib.rus.ec/book/index.php?md5=4103AA6FCEF9A0CA4D2BD1D3EFB0A58E> (cited on pages 22, 28).
- [58] Mike P. Papazoglou Dimitrios Georgakopoulos. *Service-oriented computing*. Cooperative information systems. MIT Press, 2009. ISBN: 0262072963,9780262072960,9781435699861. URL: <http://gen.lib.rus.ec/book/index.php?md5=8ba1094821a1dbfce6752aa91075d75e> (cited on pages 125, 141).
- [59] David DuRocher. *HTML & CSS QuickStart Guide: The Simplified Beginners Guide to Developing a Strong Coding Foundation, Building Responsive Websites, and Mastering the Fundamentals of Modern Web Design*. ClydeBank Media LLC, 2021. ISBN: 9781636100005,9781636100012,9781636100029,9781636100036,2020952160,978163610023. URL: <http://gen.lib.rus.ec/book/index.php?md5=B9793B0B840688161C6B4192E3CBBAD> (cited on page 41).

- [60] Thomas Erl. *Service Oriented Architecture: Principles of Service Design (The Prentice Hall Service-Oriented Computing Series)* by Thomas Erl. 1st edition. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall, 2007. ISBN: 0132344823. URL: <http://gen.lib.rus.ec/book/index.php?md5=4638a31e209e582b69dbdb95e1cbbfde> (cited on pages 51, 125, 127, 128, 133, 144, 146, 148, 149).
- [61] Thomas Erl. *SOA Principles of Service Design*. 1st edition. Prentice Hall, 2007. ISBN: 0132344823,9780132344821. URL: <http://gen.lib.rus.ec/book/index.php?md5=64e129a161bd5e0be3d040a333e70cae> (cited on page 125).
- [62] Thomas Erl. *SOA with REST : principles, patterns & constraints for building enterprise solutions with REST*. The Prentice Hall service technology series. Prentice Hall, 2013. ISBN: 9780137012510,0137012519. URL: <http://gen.lib.rus.ec/book/index.php?md5=4186ccf8b17b1e98f2f73a267821e8b2> (cited on pages 108, 141).
- [63] Thomas Erl et al. *Web Service Contract Design and Versioning for SOA*. 1st edition. 2008. ISBN: 013613517X,9780136135173. URL: <http://gen.lib.rus.ec/book/index.php?md5=91f7c78149f3172941479121b81bdb54> (cited on pages 125, 135).
- [64] Alan D. Fekete et al. *Replication: Theory and Practice*. 1st Edition. Lecture Notes in Computer Science 5959 : Theoretical Computer Science and General Issues. Springer Berlin Heidelberg, 2010. ISBN: 3642112935,9783642112935. URL: <http://gen.lib.rus.ec/book/index.php?md5=e311d3e8017d70b084e5c6f1354fbc5> (cited on page 191).
- [65] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000 (cited on pages 107, 125, 141).
- [66] David Flanagan. *JavaScript: The Definitive Guide*. Fifth Edition. O'Reilly Media, 2006. ISBN: 9780596101992,0596101996. URL: <http://gen.lib.rus.ec/book/index.php?md5=c8ddd77ef565859ebee86c67e56420d2> (cited on page 43).
- [67] Behrouz A. Forouzan. *Data communications & networking with TCP/IP protocol suite*. McGraw-Hill Forouzan networking series. McGraw-Hill US Higher Ed ISE, 2021. ISBN: 9781264363353,1264363354. URL: <http://gen.lib.rus.ec/book/index.php?md5=225ABCA66669C697F4CF2A3B4EA988C6> (cited on pages 89, 90).
- [68] Justin Gehtland, Ben Galbraith, and Dion Almaer. *Pragmatic Ajax: a Web 2.0 primer*. 1st edition. The pragmatic programmers. Pragmatic Bookshelf, 2006. ISBN: 9780976694083,0-9766940-8-5. URL: <http://gen.lib.rus.ec/book/index.php?md5=80aa520071ee877c04f137bc367833ba> (cited on page 44).
- [69] Ian Gorton. *Foundations of Scalable Systems: Designing Distributed Architectures*. 1st edition. O'Reilly Media, 2022. ISBN: 1098106067,9781098106065. URL: <http://gen.lib.rus.ec/book/index.php?md5=6C948A413C67A924BB114003B3FE0E3A> (cited on page 191).
- [70] David Gourley et al. *HTTP: The Definitive Guide*. 1st edition. O'Reilly Media, 2002. ISBN: 1565925092,9781565925090. URL: <http://gen.lib.rus.ec/book/index.php?md5=ee1f2dd4bad6e04f242e2b5cc79faa47> (cited on page 34).

- [71] Guy Harrison. *Next Generation Databases: NoSQL, NewSQL, and Big Data: What every professional needs to know about the future of databases in a world of NoSQL and Big Data*. Apress, 2016. ISBN: 978-1-484213-30-8. URL: <http://gen.lib.rus.ec/book/index.php?md5=E442A104390D20CCB808AD972EBBDE21> (cited on page 198).
- [72] Jeff Hoffer, Ramesh Venkataraman, and Heikki Topi. *Modern Database Management*. 12 (Global Ed.) Pearson Education Limited, 2016. ISBN: 1292101857,9781292101859. URL: <http://gen.lib.rus.ec/book/index.php?md5=b9fc83edf1d49b03404a8d83d6738f7e> (cited on pages 42, 43).
- [73] Eng Keong Lua John Buford Heather Yu. *P2P networking and applications*. The Morgan Kaufmann series in networking. Elsevier/Morgan Kaufmann, 2009. ISBN: 0123742145,978-0-12-374214-8. URL: <http://gen.lib.rus.ec/book/index.php?md5=0c8a4fb0c386f3bab181b634344f9bde> (cited on page 156).
- [74] Libor Dostálek; Alena Kabelová. *Understanding TCP/IP : a clear and comprehensive guide to TCP/IP protocols = Velký průvodce protokoly TCP/IP a systémem DNS*. From technologies to solutions. Birmingham, U.K. Packt Pub., 2006. ISBN: 9781847190567,1847190561. URL: <http://gen.lib.rus.ec/book/index.php?md5=ff35432fa330fff251a6eb3109d65132> (cited on page 66).
- [75] Cricket Liu. *DNS and BIND on IPv6*. O'Reilly Media, 2011. ISBN: 1449305199,9781449305192. URL: <http://gen.lib.rus.ec/book/index.php?md5=62a0e77da1c2f16a22a806398169c148> (cited on page 66).
- [76] Ozsu M.T. and Valduriez P. *Principles of distributed database systems*. 4th edition. Springer, 2020. ISBN: 9783030262525,9783030262532. URL: <http://gen.lib.rus.ec/book/index.php?md5=219FE4B3DEEA2E7700341CA15F56C487> (cited on pages 83, 198).
- [77] Chuck Musciano and Bill Kennedy. *HTML & XHTML: The Definitive Guide, Fifth Edition*. 5th edition. O'Reilly Media, 2002. ISBN: 059600382X,9780596003821. URL: <http://gen.lib.rus.ec/book/index.php?md5=0e1387dfe2eec83247cbd4ab0070b1b6> (cited on page 41).
- [78] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. 6th edition. Morgan Kaufmann, 2021. ISBN: 9780128182000. URL: <http://gen.lib.rus.ec/book/index.php?md5=5F874E482329D0C5DABA6D691668BF1C> (cited on pages 84, 87, 90, 92).
- [79] B. Pfaffenberger, Steven M. Schafer, and C.White. *HTML-XHTML and CSS Bible*. Wiley, 2004. ISBN: 0-7645-7718-2. URL: <http://gen.lib.rus.ec/book/index.php?md5=8ffe29bc74e33a9989a1611b16e0e660> (cited on page 41).
- [80] Roger S. Pressman and Bruce R. Maxim. *Software Engineering: A Practitioner's Approach*. 9th edition. McGraw-Hill Education, 2019. ISBN: 1260548007. URL: <http://gen.lib.rus.ec/book/index.php?md5=101ADC6D80142C727B6DF5AC55173102> (cited on page 58).
- [81] Dimos Raptis. *Distributed Systems for practitioners*. 2020. URL: <http://gen.lib.rus.ec/book/index.php?md5=F1C068077A4FB6CA740CFFA3ECBD04D1> (cited on pages 167, 172, 185).

- [82] C. Richardson and F. Smith. *Microservices from Design to Deployment*. Edited by NG. NGINX, Inc, 2016. URL: <https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/> (cited on pages 148, 149).
- [83] A. Salas, A. Marquez, and V. Padilla. *Arquitectura orientada a Microservicios aplicada a un sistema de Comercio Electrónico*. Trabajo de Grado de Pregrado, UNEG. 2017. 2017 (cited on pages 149, 150).
- [84] Mithun Satheesh and Bruno Joseph. *Web Development with MongoDB and NodeJS, 2nd Edition: Build an interactive and full-featured web application from scratch using Node.js and MongoDB*. Packt Publishing, 2015. ISBN: 978-1-78528-752-7. URL: <http://gen.lib.rus.ec/book/index.php?md5=6471e18c72e213b6200e8e20800a63c3> (cited on page 108).
- [85] Peter Shaw. *CSS3 Succinctly*. Edited by Createspace Independent Publishing Platform. 2017. URL: <http://gen.lib.rus.ec/book/index.php?md5=d6b4d2d62fbe04b9c83d677f3498c239> (cited on page 41).
- [86] J. Silcock et al. *Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems*. Technical reports: Computing series. Deakin University, School of Computing and Mathematics, 1995. URL: <https://books.google.co.ve/books?id=EfEwNAAACAAJ> (cited on page 54).
- [87] Dan Sullivan. *NoSQL for Mere Mortals*. Pearson, 2015. URL: <http://gen.lib.rus.ec/book/index.php?md5=8b4ecadb83b3ae2c34285b12705bfe15> (cited on pages 25, 26).
- [88] Andrew Tanenbaum and Van Marteen. *Distributed Systems: Principles and Paradigms*. Edited by Pearson Prentice Hall. 2nd. 2007. ISBN: 978-15-302817-5-6 (cited on pages 50–52).
- [89] Sasu Tarkoma. *Publish/Subscribe Systems: Design and Principles*. Edited by Joe Sventek David Hutchison Serge Fdida. 2012. ISBN: 9781119951544,9781118354261. URL: <http://gen.lib.rus.ec/book/index.php?md5=f1b33bcfa1035bb87499e892a5407c2f> (cited on pages 115, 116, 119).
- [90] Christina J. Hogan Thomas A. Limoncelli Strata R. Chalup. *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*. Volume 2. Addison Wesley, 2014. ISBN: 032194318X. URL: <http://gen.lib.rus.ec/book/index.php?md5=90c5548855a2ca750b7c9505619ce772> (cited on pages 22, 27, 42, 57, 60, 63, 65, 66).
- [91] M van Steen and A.S. Tanenbaum. *Distributed Systems*. Edited by Pearson Prentice Hall. Third. distributed-systems.net, 2017 (cited on pages 19, 22, 24, 54, 73–75, 83, 97, 174, 176, 177).
- [92] Paulo Veríssimo and Luís Rodrigues. *Distributed Systems for System Architects*. Advances in Distributed Computing and Middleware Ser. 1. Springer, 2012. ISBN: 9781461356660. URL: <http://gen.lib.rus.ec/book/index.php?md5=12F75D05F048B18B4E02721F1BDCDAE4> (cited on pages 19, 20, 22, 105, 111).
- [93] Roberto Vitillo. *Understanding Distributed Systems*. 2021. ISBN: 1838430202,9781838430207. URL: <http://gen.lib.rus.ec/book/index.php?md5=0755E00AFD01DA498C7B05076D5C00F4> (cited on pages 97, 101, 105, 167, 185).

-
- [94] J. Wu. *Distributed System Design*. First Edition. CRC-Press, 1998. ISBN: 0849331781 9780849331787. URL: <http://gen.lib.rus.ec/book/index.php?md5=8fb92e32c392c7eb049a47cf00572060> (cited on pages 176, 177).
- [95] Edward Yourdon and Yourdon Press Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979 (cited on page 111).

Glosario

acoplamiento Se define como la fuerza de la interconexión entre dos módulos de software: cuanto mayor es la fuerza de la interconexión, mayor es el acoplamiento. Para que el software sea más fácil de entender, corregir y mantener, un sistema debe estar dividido de modo que el acoplamiento entre los módulos sea lo más flojo o débil posible. 23, 128

agente móvil es un programa en ejecución (que incluye tanto código como datos) que viaja de una computadora a otra en una red llevando a cabo una tarea en la computadora de alguien, como la recopilación de información y, finalmente, regresar con los resultados. 58

API Una interfaz de programación de aplicaciones (API) es un conjunto de herramientas que un sistema hace disponible para que los sistemas o software no relacionados tengan la capacidad de interactuar entre sí. . 107

applet es un componente de una aplicación que se ejecuta en el contexto de otro programa, por ejemplo, en un navegador web. 58

backend El *backend* se le llama servidor, motor o modo administrador. También, Los backends se denominan réplicas porque todos son clones o réplicas entre sí. En diseño de software, el *backend* es la parte que procesa la entrada desde el *frontend*. 63

código móvil Se utiliza para referirse al código del programa que se puede transferir de una computadora a otra y ejecutarse en el destino: los applets de Java son un ejemplo. 24

caché es un almacén de objetos de datos utilizados recientemente que está más cerca de un cliente o un conjunto particular de clientes. Cuando se recibe un nuevo objeto de un servidor se agrega al almacén de caché local, reemplazando objetos existentes si es necesario. Cuando un proceso de cliente necesita un objeto, el servicio de almacenamiento en caché primero comprueba el caché, y luego hace la búsqueda en la red. 57

cliente ligero se refiere a una capa de software que admite una ventana interfaz de usuario que es local para el usuario mientras ejecuta programas de aplicación o, accede a servicios en una computadora remota. 60, 74

cliente-servidor Es un enfoque donde el solicitante de procesos se conocen como cliente, y el que atiende el proceso es el servidor. Esta arquitectura usa reglas estándares para la interacción como el HTTP, protocolo de transferencia de hipertexto (HTTP) mediante el cual los navegadores y otros clientes obtienen documentos y otros recursos de servidores web. 20, 47

componentes Los componentes de software son como objetos distribuidos en el sentido de que están encapsulados en unidades de composición, pero un componente dado especifica sus interfaces al mundo exterior y sus dependencias de otros componentes del entorno distribuido.. 51

composición de servicios es un agregado de servicios compuestos colectivamente para

- automatizar un determinado tarea o proceso de negocio.. 127
- computación en la nube** La computación en la nube (*cloud computing*), conocida también cm servicios en la nube, informática en la nube o simplemente la nube, es un paradigma que permite ofrecer servicios de computación a través de la red, que usualmente es internet. 49
- computación ubicua** Es entendida como la integración de la informática en el entorno de la persona, de forma que los ordenadores no se perciben como objetos diferenciados, apareciendo en cualquier lugar y momento. 49
- comunicación de multidifusión** La comunicación en la que se envía un mensaje a todos los miembros del grupo mediante una única operación. 112
- comunicación grupal** Ofrece un servicio mediante el cual se envía un mensaje a un grupo y luego este mensaje se entrega a todos los miembros del grupo . 55, 111
- comunicación indirecta** Se define como la comunicación entre entidades en un sistema distribuido a través de un intermediario sin acoplamiento directo entre el emisor y el receptor. 111
- comunicación unidifusión** La comunicación en la que se envía un mensaje a un solo miembro del grupo mediante una única operación.. 112
- concurrency** es la capacidad de ejecutar varios procesos simultáneamente, es decir, la existencia de más de un proceso en períodos de tiempo superpuestos.. 176
- consistencia** La consistencia significa que cada usuario de la base de datos tiene una vista idéntica de los datos en cualquier instante dado. Si hay varias réplicas y se procesa una actualización, los usuarios verán que la actualización se activa al mismo tiempo. 24, 194
- contexto de hilos** Es una colección mínima de valores almacenados en registros y memoria, utilizada para la ejecución de una serie de instrucciones (es decir, contexto procesador, estado).. 73
- Corba** *Common Object Request Broker Architecture (CORBA)* es un estándar definido por *Object Management Group (OMG)* que permite que diversos componentes de software escritos en múltiples lenguajes de programación y que corren en diferentes computadoras, puedan trabajar juntos; es decir, facilita el desarrollo de aplicaciones distribuidas en entornos heterogéneos. 51
- CSS** CSS, *Cascading Style Sheets*, es el lenguaje que se emplea para dar estilo a las páginas HTML, que es el lenguaje en que se escriben las páginas Web. 39
- código por demanda** es la capacidad de envío de códigos ejecutables del servidor al cliente cuando se requiera, lo cual amplía las funciones del cliente.. 108
- datagrama** Un datagrama es un paquete de datos que constituye el mínimo bloque de información en una red de conmutación por datagramas, la cual es uno de los dos tipos de protocolo de comunicación por conmutación de paquetes usados para encaminar por rutas diversas dichas unidades de información entre nodos de una red, por lo que se dice que no está orientado a conexión.. 88
- debilmente acoplado** El sistema está débilmente acoplado si cada componente tiene poco o ningún conocimiento de las partes internas de los otros componentes. 23, 129
- degradación elegante** concepto que describe a sistemas informáticos y de red tolerantes a fallas que siguen funcionando incluso si algunos componentes fallan, y enfatiza la adaptación automática a las circunstancias para mantener el servicio. 21
- deriva de reloj** se refiere a varios fenómenos relacionados debido a los que un reloj no marcha exactamente a la misma velocidad que otro, lo que significa, que después de

cierto tiempo la hora indicada por el reloj se irá separando de la indicada por el otro.. 168

descubrimiento de servicios Se refiere a la necesidad de que los servicios estén disponible, publicados, accesible y documentados con una serie de meta-datos que permitan lanzar búsquedas ricas para identificar los servicios que podamos reutilizar. 49, 93

disponibilidad La disponibilidad es que la base de datos permanece operativa a pesar de la ocurrencia de falla. 24, 25

DNS El sistema de nombres de dominios *Domain Name Systems* (DNS), un servidor DNS es responsable de informar a todas las demás computadoras en Internet sobre su nombre de dominio y la dirección de su sitio. 23

DOM El DOM, Modelo de Objetos del Documento, presenta la estructura de las páginas web como un conjunto de objetos programables que se pueden manipular con JavaScript. 44

entorno de ejecución Un entorno de ejecución es la unidad de gestión de recursos: una colección del kernel gestionado recursos locales a los que sus subprocesos o hilos pueden acceder.. 73

evento es la ocurrencia de una acción que conlleva un proceso *fuera de la ejecución* a medida que se ejecuta: una acción de comunicación o una acción que transforma el estado del sistema como todo.. 167

eventos concurrentes No se puede establecer una relación de precedencia y por tanto nada se puede decir (o se necesita decir) sobre cuándo ocurrieron estos eventos, o cuál evento ocurrió primero.. 173

fragmentación es un patrón de arquitectura de base de datos que divide una única base de datos en tablas más pequeñas conocidas como fragmentos, cada una almacenada en un nodo independiente.. 198

fuertemente acoplado El sistema tiene acoplamiento fuerte si cada componente de un sistema tiene conocimiento de partes de los otros componentes, como memoria común, almacenamiento secundario, entrada y salida de datos. 23, 111

hilos un **hilo** o **hebra** (del inglés *thread*), **proceso ligero** o **subproceso** es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo. Es básicamente una tarea que puede ser ejecutada en paralelo con otra tarea.. 50, 73

historia del proceso es la serie de eventos que se colocan dentro del proceso p_i , ordenado como se ha descrito por la relación i :

$$history(p_i) = h_i = \langle e_i^0, \langle e_i^1, \langle e_i^2, \dots \rangle \rangle \rangle \dots 167$$

HTML Lenguaje de marca de hipertexto o *HyperText Markup Language* (HTML), un lenguaje para especificar los contenidos y diseño de las páginas tal como se muestran en los navegadores web. 38

HTTP El Protocolo de transferencia de hipertexto, *HyperText Transfer Protocol* (HTTP) define las formas en que los navegadores y otros tipos de clientes interactúan con los servidores web. 34, 90

inanición La inanición de un programa ocurre si y solo si la ejecución de algunas de sus acciones que deben ejecutarse nunca se ejecutarán en un tiempo finito durante la evolución del sistema, lo que resultará en la suspensión indefinida de la actividad del programa.. 176

- informática orientada a servicios** es un término general que representa una nueva generación de plataforma informática distribuida. Como tal, abarca muchas cosas, incluida su propio paradigma de diseño y principios de diseño, catálogos de patrones de diseño, lenguajes de patrones, un modelo arquitectónico distinto y conceptos, tecnologías y marcos relacionados.. 125
- interbloqueo** también Deadlock o abrazo mortal, es una situación en la que dos o más procesos no pueden continuar porque están esperando que el otro libere recursos.. 176
- interfaz de servicio web** es la colección de operaciones que pueden ser usadas por un cliente sobre la internet. 130, 132
- inventario de servicios** es una colección estandarizada y gobernada de forma independiente de servicios complementarios dentro de un límite que representa una empresa o un segmento significativo de una empresa.. 127
- JavaScript** JavaScript es un lenguaje de secuencias de comandos de propósito general diseñado para ser incrustado dentro de las aplicaciones. 43
- latencia** La demora entre el inicio de la transmisión de un mensaje de un proceso y el comienzo de su recepción por otro. 32
- máquina virtual** Proporciona una forma de hacer que el código sea ejecutable en una variedad de equipos host: el compilador de un lenguaje en particular genera código para una máquina virtual en lugar de un código de pedido de hardware en particular. Por ejemplo, el compilador Java produce código para una máquina virtual Java, que lo ejecuta por interpretación. 24
- middleware** una capa de software que proporciona abstracción a la programación, así como enmascarar la heterogeneidad de las subyacente redes, hardware, sistemas operativos y lenguajes de programación. 19, 24, 60, 65
- MIME** Multipurpose Internet Mail Extensions o MIME (en español "extensiones multi-propósito de correo de internet") son una serie de convenciones o especificaciones dirigidas al intercambio a través de Internet de todo tipo de archivos (texto, audio, vídeo, etc.) de forma transparente para el usuario. 35
- modelos arquitectónicos** presenta la arquitectura de un sistema así como su estructura en términos de sus componentes y de sus interrelaciones. 47
- modelos descriptivos** Captura las propiedades y los problemas de diseño de los sistemas, y proporciona una descripción abstracta, simplificada pero consistente de un aspecto relevante del diseño de sistemas distribuidos. 47
- modelos físicos** capturan la composición de hardware de un sistema en términos de las computadoras (y otros dispositivos, como teléfonos móviles) y sus redes de interconexión descripción. 47
- nodos** En informática y en telecomunicación, un nodo es un punto de intersección o conexión de varios elementos que confluyen en el mismo lugar. En informática la palabra nodo puede referirse a conceptos diferentes según el ámbito: computadora, servidor, enrutadores, entre otros. 50
- nodos autónomos** Nodos en gran medida independiente de otras computadoras en términos de su físico e infraestructura. 48
- nodos discretos** Nodos no integrados en otras entidades físicas. 48
- nodos estáticos** Nodos que permanecer en una ubicación física períodos extendidos. 48

- objetos** un objeto puede ser una variable, una estructura de datos, una función o un método y, como tal, es un valor en la memoria referenciado por un identificador.. 51
- orientación a servicios** es un paradigma de diseño destinado a la creación de unidades lógicas de solución que tienen forma individual para que puedan utilizarse colectiva y repetidamente en soporte de la realización de los objetivos y beneficios estratégicos específicos asociados con SOA y la computación orientada a servicios.. 126
- página dinámica** Las páginas dinámicas son páginas *HTML* generadas a partir de lenguajes de programación (*scripts*) que son ejecutados en el propio servidor web. 42
- página estática** Las páginas estáticas son páginas *HTML* que permanecen tal y como fueron diseñadas. 42
- paradigma de diseño** Un paradigma de diseño es un conjunto de reglas o principios complementarios que definen colectivamente el enfoque global representado por el paradigma. . 125
- patrón proxy** es un contenedor o un objeto de agente ubicado en el cliente, que representa al objeto remoto. El proxy ofrece exactamente la misma interfaz al objeto remoto, y el programador realiza llamadas a este objeto proxy sin necesidad de estar consciente de la naturaleza distribuida de la interacción. 60
- patrón de diseño** Describe un problema común y proporciona una solución correspondiente. Documenta la solución en un formato genérico para que pueda aplicarse de forma repetida. . 125
- peticiones idempotentes** ante una repetición de solicitud de información ya realizada, el servidor debe responder con la misma información dejando el sistema en el mismo estado que si hubiese llegado de nuevas peticiones. 32
- procesos** Se define como un programa en ejecución, es decir, un programa que se está ejecutando actualmente en uno de los procesadores virtuales del sistema operativo. 50, 73, 87
- protocolo solicitud-respuesta** El cliente envía un mensaje de solicitud al servidor que contiene la URL del recurso requerido. El servidor busca el nombre de la ruta y, si existe, envía el contenido del recurso en un mensaje de respuesta al cliente. De lo contrario, devuelve una respuesta de error como, por ejemplo: 404 no encontrado. 51, 54, 97
- publicación-suscripción** Los publicadores diseminan un evento a través de una operación *publish(e)* y los suscriptores expresan interés en el conjunto de eventos a los que se suscriben, con la operación *subscrib(e)*.. 55, 115
- recurso** En su forma más simple, un recurso en la Web es una página web o algún otro tipo de contenido que se puede presentar al usuario, como archivos multimedia y documentos en formato de documento portable. 20, 22, 34
- redes superpuesta** es una red virtual de nodos enlazados lógicamente, que está construida sobre una o más redes subyacentes *underlying network*. Los nodos de la red superpuesta están conectados por enlaces virtuales. Su objetivo es implementar servicios de red que no están disponibles en la red subyacente. Las redes superpuestas pueden apilarse de forma que tenga capas que proporcionen servicios a la capa superior. 81, 84
- relojes** son dispositivos electrónicos que cuentan las oscilaciones que ocurren en un cristal a con una frecuencia definida, y generalmente dividen este recuento y almacenan el resultado en un contador de registro.. 168

resolutores Se denomina Resolutores a los clientes del DNS.. 66

script Un script es un código de programa que no necesita procesamiento previo (por ejemplo, compilación) antes de ejecutarse. En el contexto de un navegador web, las secuencias de comandos generalmente se refieren al código de programa escrito en JavaScript que ejecuta el navegador cuando se descarga una página o en respuesta a un evento desencadenado por el usuario. 42

semántica mejor-esfuerzo un datagrama puede perderse, retrasarse, duplicarse, o distribuirse fuera de orden. . 94

servicio Un servicio es una unidad de lógica de solución a la que se ha aplicado la orientación al servicio en una medida significativa. Es la aplicación de principios de diseño orientados al servicio que distinguen una unidad de lógica como un servicio en comparación con unidades de lógica que pueden existir únicamente como objetos o componentes.. 20, 126

servicio web es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones.. 129

servicios web Es un sistema software diseñado para soportar la interacción máquina-a-máquina, a través de una red, de forma interoperable. Cuenta con una interfaz descrita en un formato procesable por un equipo informático (específicamente en WSDL), a través de la que es posible interactuar con el mismo mediante el intercambio de mensajes SOAP, típicamente transmitidos usando serialización XML sobre HTTP conjuntamente con otros estándares web.. 51

servidor proxy su propósito es aumentar la disponibilidad y el rendimiento del servicio al reducir la carga en la red y los servidores web. Los servidores proxy pueden asumir otros roles como acceder a servidores web remotos a través de un firewall. 58

servidor web Servidor web o servidor HTTP es un programa informático que procesa una aplicación del lado del servidor, realizando conexiones bidireccionales o unidireccionales y síncronas o asíncronas con el cliente y generando una respuesta en cualquier lenguaje o aplicación del lado del cliente. 34

servidores de nombres Son el conjunto de servidores que conforman el DNS.. 66

sin estado En arquitecturas cliente-servidor, implica que no se almacena la información del cliente entre las solicitudes de GET y que cada una de ellas es independiente y está desconectada del resto de las solicitudes.. 108, 129

sistemas abiertos Se basan en la provisión de un mecanismo de comunicación uniforme e interfaces publicadas para el acceso a recursos compartidos. Además, se pueden ampliar e implementar de nuevas formas sin alterar su funcionalidad existente. 22, 48

tablas hash distribuidas Las tablas hash distribuidas, conocidas como DHT (del inglés, Distributed Hash Tables), son un tipo de tablas hash que almacenan pares de clave-valor y permiten consultar el valor asociado a una clave, en las que los datos se almacenan de forma distribuida en una serie de nodos y proporcionan un servicio eficiente de búsqueda que permite encontrar el valor asociado a una clave. 50, 154

teorema CAP Teorema CAP (Consistency, Availability, Partition Tolerance) es un principio que indica que en un sistema de cómputo distribuido, se puede cumplir como máximo dos de las siguientes propiedades: consistencia, disponibilidad y tolerancia de partición. 24

tolerancia de partición La tolerancia de partición indica que la base de datos puede

mantener las operaciones en caso de que la red falle, entre dos segmentos del sistema distribuido. 24, 25

transparencia La transparencia en los sistemas distribuidos es el ocultar del usuario y del programador de la aplicación, la separación de componentes en un SD, de manera que el sistema se percibe como uno más que como la colección de componentes independientes. 22

URL Localizadores uniforme *Uniform Resource Locator* (URL), también conocidos como identificadores uniformes de recursos *Uniform Resource Identifiers* (URI), que identifican documentos y otros recursos almacenados como parte de la Web. 35

XMLHttpRequest El objeto XMLHttpRequest permite a los programadores web recuperar datos desde el servidor web como una actividad de fondo. El formato de datos suele ser XML, pero funciona bien con cualquier dato basado en texto.. 44

Índice

- algoritmo basado en elecciones, 182
- algoritmo de anillo basado en elecciones, 182
- algoritmo de bully, 183
- algoritmo de encuentros, 120
- algoritmo de filtrado, 119
- algoritmo de generales bizantinos, 184
- algoritmo de Lamport de exclusión mutua, 178
- algoritmo de Ricart y Agrawala, 180
- Algoritmo de Suzuki-Kazami, 180
- algoritmo servidor central, 177
- algoritmos basados en marcas de tiempo, 178
- algoritmos basados en token, 176
- autonomía de servicios, 129
- composición de servicios, 127
- criptografía de datos, 24
- elementos de informática orientada a servicios, 125
- exclusión mutua, 176
- implementación distribuida en pub-sub, 117
- interfaz en REST, 143
- orientación a servicios, 126
- publicación-suscripción, 55
- servicio sin estado, 129
- servicios, 126
- servidores sin estado, 142
- sistemas de capas, 144
- SOA, 126
- transparencia de concurrencia, 22
- transparencia de ubicación, 22
- vista física de servicios, 128

- abstracción, 23, 129
- acoplamiento, 23, 128
- acoplamiento en espacio y tiempo, 112
- acoplamiento fuerte, 23
- administradores de réplicas, 191
- agente, 142
- agente móvil, 58
- AJAX, 44

- algoritmo de relojes vectoriales, 175
- algoritmo de Bekerley, 170
- algoritmo de Cristian, 168
- algoritmo de elección de líder de Raft, 185
- algoritmo de inundación, 119
- algoritmo de relojes lógicos de Lamport, 172
- algoritmos DHT, 157
- alias, 67
- aplicaciones pub-sub , 115
- applet, 58
- arquitectura de capas, 60
- arquitectura de RMI, 105
- arquitectura de Skype, 85
- ausencia de reloj, 20

- bloqueo en UDP, 90

- código móvil, 58
- caché, 58
- cache, 142
- caché fragmentado, 199
- capas, 60
- características de colas, 121
- características de TCP, 91
- características del protocolo solicitud-respuesta, 100
- caso de estudio
 - ¿como funciona un aplicación de chat?, 94
 - API, 107
 - cache, 198
 - elección del lider, 185
 - fragmentación, 198
 - mensajería pub-sub, 122
 - microservicios, 148
 - monolitos, 148
 - Pastry, 160
 - patrón pub-sub, 122
 - Sistema de Nombres de Dominio, 66
 - tecnología web, 34
 - virtualización de redes, 84
- centralizado vs distribuido, 20

- Chord, 157
- cliente-servidor, 48
- clientes flacos, 60
- cola de mensaje, 55
- colas, 120
- colector de basura distribuida, 107
- comercio electrónico, 21
- componentes, 51
- computación en la nube, 49
- computación ubicua, 49
- comunicación grupal, 55
- comunicación entre procesos, 97
- comunicación indirecta, 111
- concurrency, 19, 22
- confiabilidad, 21, 24
- confiabilidad en grupos, 114
- consenso distribuido, 176
- consistencia, 24
- consistencia secuencial, 194
- contrato de servicios, 128
- contratos no Rest, 145
- contratos Rest, 147
- control de flujo en TCP, 91
- coordinación de servicios web, 141
- Corba, 51
- CSS, 39
- código por demanda, 144
- datagram, 88
- datos replicados, 94
- debilmente acoplado, 23
- degradación elegante, 21
- deriva del reloj, 168
- desacoplamiento en espacio, 55
- desacoplamiento en tiempo, 55
- descubrimiento de servicio, 129
- descubrimiento de servicios, 49, 93
- destino de mensajes en TCP, 91
- disponibilidad, 21, 25
- DNS, 24, 66
- dominio, 67
- e-learning, 21
- eCiencias, 21
- entidades, 50
- escalable, 23
- espacio de nombres, 67
- espacio de nombres de dominio, 67
- espacio de tuplas, 56
- estilos de protocolos, 100
- estructura del mensaje, 97
- fallas, 20, 30
- gestión ambiental, 22
- Gnutella, 156
- grupos, 111
- grupos abiertos, 113
- grupos cerrados, 113
- grupos no superpuestos, 113
- grupos superpuestos, 113
- Hcliente, 34
- heterogéneos, 20, 24
- hilos, 50
- HTML, 38
- HTTP, 34
- IaaS, 83
- implementación centralizada en pub-sub, 117
- implementación de RPC, 102
- informática móvil, 49
- informática de la salud, 21
- informática orientada a servicios, 125
- infraestructura como servicio, 83
- interfases de usuarios, 74
- interfases remotas, 105
- inventario de servicios, 127
- invocación remota, 54
- JavaScript, 43
- juegos en línea, 21
- linealizabilidad, 194
- llamadas a procedimientos remotos, 54
- membresía de grupos, 112
- memoria compartida distribuida, 56
- mensajes perdidos en TCP, 91
- metas de REST, 144
- middleware, 32
- MIME, 35
- modelo de programación de colas, 121
- modelo de programación de grupos, 112
- modelo de programación pub-sub, 116
- modelo de suscripción pub-sub, 116
- modelo fallas, 100

- modelos arquitectónicos, 49
- modelos descriptivos, 47
- modelos físicos, 47
- modular, 20
- multidifusión, 54, 93
- multihilo, 73
- métodos, 36

- Napster, 155
- niveles de transparencia, 22
- nodos, 50
- nombres de dominio, 68
- notificaciones de eventos, 94

- objeto, 51
- objetos en grupos, 113
- orden de mensajes en TCP, 91
- ordenamiento en grupos, 114

- P2P, 48
 - 1era generacion, 155
 - 2da generacion, 156
 - 3era generacion, 157
 - evolución, 155
- p2p, 153
- P2P estructurados, 154
- P2P no estructurados, 154
- PaaS, 84
- paradigmas de comunicación, 54
 - invocación a métodos remotos, 54
- pase de mensaje, 54
- Pastry, 160
- patrones
 - Balanceador de carga con múltiples réplicas de *backend*, 63
 - Servidor con varios backends, 63
 - Árbol de servidores, 64
- plataforma como servicio, 84
- potocolos capa de transporte, 87
- principios de diseño, 22
- procesos, 50, 54
- procesos e hilos, 73
- procesos en grupos, 113
- programación con interfases, 101
- protocolo de replicación, 192
- protocolo NTP, 171
- protocolo solicitud respuesta, 97
- protocolo solicitud-respuesta, 54
- protocolos, 90
- pub-sub, 55, 115
- página dinámica, 42
- página estáticas, 42

- recepción de mensajes en UDP, 90
- recursos, 22, 34
- red superpuesta, 85
- referencia a objetos remotos, 105
- reloj, 168
- reloj físico, 168
- reloj lógico, 172
- rendimiento, 24
- replicación, 191
- replicación activa, 196
- replicación pasiva, 195
- resolución, 68
- REST, 141
- Rest, 107
- RestFul, 107
- restricciones de REST, 142
- reutilización de servicios, 129
- RMI, 54, 104
- RPC, 54, 101

- SaaS, 84
- script, 42
- seguridad, 24
- seguridad XML, 140
- semántica de llamadas RPC, 101
- servicios web, 51, 129
- servidor multihilos, 76
- servidor web, 34
- servidores múltiples, 57
- sesgo del reloj, 168
- simplicidad, 23
- sincronización, 167
- sistemas abiertos, 22
- sistemas distribuidos, 19
- Skype, 85
- SOA, 125
- SOAP, 130
- socket, 54, 87
- software como servicio, 84
- solicitud-respuesta, 136

- tablas hash distribuidas, 154
- tamaño mensaje en TCP, 91

tamaño mensaje en UDP, 89
TCP, 90
teorema CAP, 24
tiempo, 167
tiempo de espera en UDP, 90
Tiempo y Consenso Distribuido, 167
tipos de transparencia, 22
tolerancia a fallas, 21, 93, 193
transacción, 36
transparencia, 22
transparencia de escalamiento, 22
transparencia de fallas, 22
transparencia de movilidad, 22
transparencia de red, 22
transparencia de rendimiento, 22
transparencia de replicación, 22
transparencia en el acceso, 22
transparencia en RPC, 102
transporte, 21
técnicas contra fallas, 30

UDDI, 139
UDP, 88
URL, 35
User Datagram Protocol, 88
uso de UDP, 90
UTC, 168

virtualización, 81, 83

WSDL, 132

XHTML, 41

ámbito geográfico, 20

