Programación IV
Profesora Cynthia Estrada

Fecha: 14/08

"HTTP vs WebSockets"

Integrantes:

- "Díaz Rossini Juan José",
- "Gallo Genaro"
- "Navarro Victor Leandro"

Cuadro comparativo entre HTTP y Websockets

Aspecto	HTTP	WebSockets
Modelo de comunicación	HTTP utiliza un modelo petición-respuesta, donde el cliente siempre inicia la comunicación y el servidor solo responde. Esto lo hace ideal para transacciones puntuales como cargar páginas web, enviar formularios o consumir APIs REST. Sin embargo, no permite que el servidor envíe información por iniciativa propia al cliente.	WebSockets implementa un modelo de comunicación full-duplex (bidireccional). Una vez establecida la conexión, tanto el cliente como el servidor pueden intercambiar mensajes en cualquier momento sin necesidad de que uno espere al otro. Esto es crucial para aplicaciones interactivas en tiempo real.
Gestión de la conexión	Cada vez que el cliente quiere algo, debe abrir una nueva conexión, enviar la petición y esperar la respuesta, para luego cerrar la conexión. Esto genera sobrecarga en aplicaciones que requieren muchas solicitudes pequeñas.	Se establece una única conexión persistente entre cliente y servidor, que se mantiene abierta mientras dure la sesión. Esto reduce significativamente la sobrecarga en contextos con mensajes frecuentes o transmisión continua de datos.
Latencia y rendimiento	La latencia tiende a ser más alta, ya que cada interacción conlleva apertura, transmisión y cierre de la conexión. Esto no afecta demasiado en páginas estáticas o APIs con pocas peticiones.	La latencia es mínima, porque no hay que renegociar conexiones cada vez. Esto lo hace más eficiente para aplicaciones que necesitan enviar/recibir información en milisegundos, como videojuegos, chats o cotizaciones en bolsa.

Eficiencia en el uso de recursos	**Es eficiente cuando las solicitudes son aisladas y poco frecuentes, porque el servidor solo trabaja cuando hay una petición. Además, aprovecha muy bien el caché HTTP, reduciendo aún más la carga.**	**Es mucho más eficiente en escenarios de alta frecuencia de mensajes, ya que evita abrir/cerrar conexiones de manera repetida. Sin embargo, mantener muchas conexiones abiertas simultáneamente puede requerir mayor consumo de memoria en el servidor.**
Compatibilidad y soporte	**Es un estándar universal, soportado por absolutamente todos los navegadores, servidores y dispositivos conectados a Internet. Funciona sin problemas con proxies, balanceadores de carga, firewalls y CDNs.**	**Tiene soporte amplio en navegadores modernos, pero algunos entornos antiguos, firewalls corporativos o proxies intermedios pueden bloquear o limitar su uso. Aun así, hoy en día está muy extendido y suele ser aceptado en la mayoría de infraestructuras.**
Escalabilidad	**HTTP es altamente escalable porque se integra fácilmente con CDNs, balanceadores de carga y sistemas de caché. Es la mejor opción para aplicaciones web que esperan millones de usuarios con peticiones dispersas.**	**La escalabilidad puede ser más compleja, porque el servidor debe mantener activas miles (o millones) de conexiones abiertas al mismo tiempo. Requiere arquitecturas optimizadas y servidores especializados (ejemplo: Node.js, Nginx con soporte WebSocket).**
Seguridad	**Utiliza el protocolo HTTPS, que cifra los datos con TLS/SSL. Es confiable y ampliamente auditado. Además, al ser tan estándar, se integra con firewalls, certificados y mecanismos de autenticación de manera nativa.**	**Utiliza WSS (WebSocket Secure), que también emplea TLS/SSL. Ofrece la misma seguridad en el transporte que HTTPS, pero en algunos casos puede requerir configuraciones extra para trabajar con proxies y certificados.**
Complejidad de implementación	**Es relativamente simple de implementar: la mayoría de frameworks (Express, Spring, Django, etc.) tienen soporte nativo para HTTP. La lógica cliente-servidor es sencilla y directa.**	**Su implementación es más compleja, ya que requiere configurar la conexión persistente y gestionar los eventos de apertura, mensaje, error y cierre. Además, los servidores deben estar preparados para manejar conexiones simultáneas de larga duración.**

Casos de uso típicos	**Cargar páginas web, enviar formularios, consumir APIs REST, sistemas de e-commerce, blogs, aplicaciones donde la interacción en tiempo real no es crucial.**	**Chats en vivo, aplicaciones de mensajería, videojuegos multijugador online, cotizaciones bursátiles, seguimiento en tiempo real (GPS), notificaciones instantáneas y transmisión de datos en directo (streaming).**