Trabajo sobre capa de aplicación

Carlos Urquiza

29 de diciembre de 2025

1. Introducción

La capa de Aplicaciones es aquella que justifica la existencia de internet, tenemos los protocolos para utilizar aplicaciones de redes. Vamos a ver que servicios y arquitecturas usan. En cierta medida también el protocolo HTTP. Una aplicación muy conocida en redes es, justamente web.

- Email.
- Web.
- Mensajeria instantanea.
- Login remoto.
- Compartición de archivos P2P.
- Juegos de red multiusuario.
- Reproducción de videos almacenados (Youtube, netflix).
- Computación paralela.
- Conferencias de video. (Meet)

¿Qué cosas necesito para hacer una aplicación de red?

En un principio, hay que crear un programa que sea capaz de correr en distintos dispositivos que querramos usar. Si queremos hacer un modelo web, como una pagina hay que desarrollar un cliente y un servidor. Si necesito una applicación peer to peer, hay que crear cada uno de los modulos y un programa que sea capaz de compactar todos estos.

Es importante notar que lo importante no es que se desarrolle en los dispositivos intermedios, si no en los dispositivos terminales. ¿Por qué? Porque esto se abstrae y se asume que ya funciona, es parte de todo el trabajo que hicimos con todas las capas anteriores.

1.1. Arquitecturas de aplicación

A la hora de pensar cómo sería una aplicación, conviene pensar cual sería la arquitectura de la aplicación.

¿Será cliente servidor? ¿Será peer to peer? ¿Será un hibrido?- Primero debemos saber qué es cada una de estas arquitecturas. Recordemos, la arquitectura de la aplicación $\neq$ arquitectura de la red.

Cliente servidor.

En la arquitectura cliente servidor tenemos un servicio muy claro, un cliente, junto a un servidor que está conectado 24/7 para asegurar la calidad del servicio. El cliente le manda una solicitud al servidor y el servidor le responde.

En el caso de la IP, en el servidor es permanente, ya que sirve para que el cliente siempre sepa a donde ir. En cambio, en el cliente no es necesariamente así, puede ser una IP dinamica.

Cuando se necesitan hacer operaciones complejas, como redes sociales, plataformas grandes, y otros se pueden conectar varios servidores para hacer un **cluster**. Esto permite a los servidores poder aceptar todas las solicitudes que se necesiten en tiempo y forma y no hacer esperar a los clientes. En otra jerga se dice que es **escalable**. Lo que significa que uno puede extender la operación (tener más clientes) sin perder calidad (tiempo de espera).

Una ultima caracteristica es que para conectarse ambos clientes, deben pasar por el servidor, y no entre sí.

Arquitectura peer to peer.

En este caso, los host se conectan entre sí. Acá no hay dependencia de infrasestructura, en cambio, se explota la comunicación directa entre pares (peers). Una caracteristica de la arquitectura peer to peer es su autoescalabilidad.

Agregar un nodo en la red hace que haya más carga, pero, a la vez añade servicio. Pero como contra añade falta de seguridad, rendimiento y fiabilidad.

Esto es una razón por la que no se usan tanto comercialmente. Ejemplos de peer to peer son BitTorrent, Xunlei.

Arquitectura hibrida

Dado los errores anteriormente mencionados, surje la idea de hacer hibridos que traten de asegurar mejor seguridad. Se usa un servidor central que crea sesiones entre ambos peers. De forma que podamos tratar de reducir costos y tratar de permitir más seguridad. Este modelo es el que tenía skype, pero luego de años terminó quedando en cliente servidor.

Por qué siempre cliente-servidor en la actualidad

Skype trató de hacer un servicio con peer to peer pero tuvo problemas:

- CGNAT
- Los firewalls restrictivos
- Redes moviles
- Dificultad a la hora de interceptar tráfico malicioso
- Complicaciones a la hora de cumplir leyes (retención de datos, abuso, spam)

En la epoca en la que se planteó Skype, estas cosas no eran problemas, se tenía ipv4, y routers simples. En cambio con el tiempo se volvió más y más inseguro.

Con el cambio a cliente servidor, se veía una inversión más grande pero también se veía:

- Suscripciones.
- Grabaciones.
- Integración empresarial.
- Publicidad.
- Analíticas.

Por esto, skype vió una oportunidad en el cliente servidor, y se decidió hacer un cambio de arquitectura.

1.2. Procesos de la capa de aplicación.

Cuando hablamos de programas que se comunican entre sí, en realidad hablamos de comunciación entre procesos. La comunicación de procesos puede surgir en un host o entre host, los primeros son arbitrados por un sistema operativo, y los segundos por la red.

Procesos entre cliente y servidor.

Una aplicación de red consta de parejas de procesos que se envían mensajes entre sí a través de una red. Si tenemos una red peer to peer, podemos designarle a un nodo el cliente, y al otro servidor, o a veces ambos a la vez como en skype. A veces va a ser dificil saber quien es el cliente y el servidor, para esto podemos hacer una regla especifica:

En el contexto de una sesión de comunicación entre una pareja de procesos, el proceso que inicia la comunicación (es decir, que inicialmente se pone en contacto con el otro proceso al principio de la sesión) se designa como el cliente. El proceso que espera a ser contactado para comenzar la sesión es el servidor.

Comunicacion de procesos.

Los procesos se envian mensajes a través de sockets. Los sockets son una interfaz que transforma la información de capa de aplicación a la capa de transporte.

Protocolos de la capa de aplicación.

Definen el tipo de mensaje a tener:

- El formato de los request como la respuesta de los procesos que corre en los host.

- Cómo se delimita los campos
- La semantica de los campos
- Las reglas o el flujo de los mensajes entre los procesos

Tipos de protocolo

Los protocolos publicos y los protocolos privados:
Protocolos de dominio publico:

- Definidos en RFC's
- Permiten tener inter operatividad.
- Ejemplos son : HTTP, SMTP

Protocolos privados:

- Secretos industriales
- Skype

Habíamos hablado hace poco sobre sockets, acá cada protocolo debe decidir qué tipo de protocolo de la capa de transporte quiere usar, ya que es una decisión que involucra confiabilidad de espera, retardo y tasa de datos. [1]

1.3. La Web y HTTP

La Web

Se debe diferenciar qué es una aplicación de red y cual es el protocolo de la capa que utiliza esa aplicación.

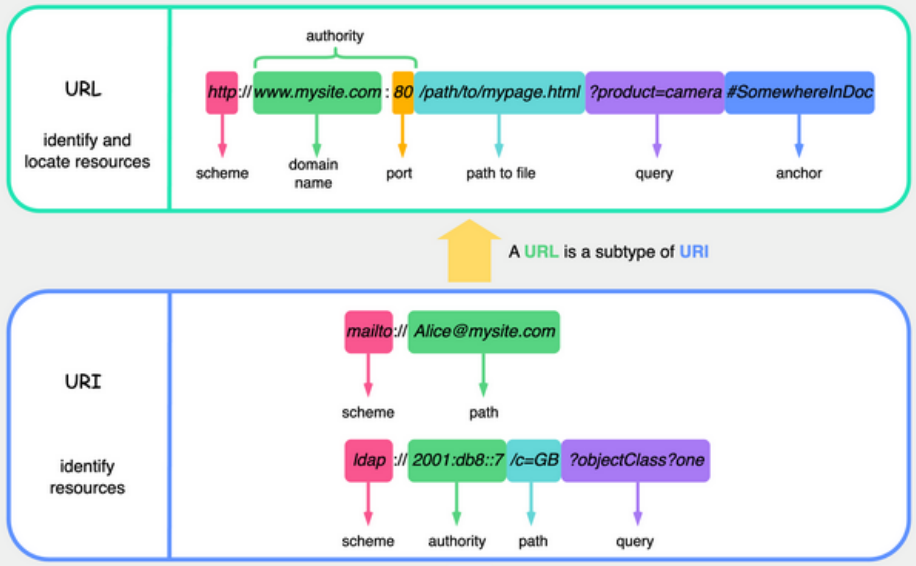
La web es una aplicación cliente servidor que permite al usuario obtener documentos bajo demanda, y esta contiene varios .ºbjetos"que lo convierten en algo más que un protocolo.

- Un estandar para los documentos, llamado html.
- Un interpretador de estos documentos, el navegador.
- El protocolo HTTP
- Servidor web, apache, microsoft, etc.
- URI: un tipo de identificador que identifica recursos. Solo promete que identifica un recurso de forma única dentro de algún sistema.
- URL: un subtipo de URI, este indica dónde está el recurso y cómo acceder a el.
- Tanto URI como URL se crearon independientemente ya que no todos los recursos son "navegables".

[1] A partir de acá se habla un poco sobre qué protocolo conviene, si TCP o udp. No me parece necesario ya que se vió en redes 1.

URI y URL

Para hablar de HTTP debemos entender que ambos usan URI's y URL's. Por lo tanto vamos a hacer un breve estudio de estos.



[2]

Una URL tiene 3 componenetes:

- Esquema: Le dice al cliente cómo acceder al recurso, HTTP, HTTPs, file, etc.
- Host : Normalmente, es un nombre de dominio que se traduce luego a una IP mediante un servidor DNS. El puerto se omite ya que se sabe el esquema y el nombre de dominio
- Camino : Identifica recursos en el servidor, se puede ver como un camino en un disco. También puede haber una query o una consulta a una base de datos.

[2] Fuente: HTTPs://bytebytego.com/guides/url-uri-urn-do-you-know-the-differences/

HTTP

HTTP funciona de una forma relativamente simple, la idea es manejarse con mensajes simples: request y response, pero dado que funciona mediante TCP, requiere que se haga una conexión primero.

1. Cliente inicia conexión TCP (crea socket) al servidor, puerto 80 (puede ser otro!)
2. Servidor acepta conexión TCP del cliente
3. Mensajes HTTP (mensajes del protocolo de capa aplicación) son intercambiados entre browser (cliente HTTP) y servidor Web (servidor HTTP)
4. Se cierra la conexión TCP

HTTP no mantiene estado de sesión por decisión de diseño, lo que simplifica el protocolo y mejora su escalabilidad. Esta característica permite que el manejo de estado se implemente de forma flexible a nivel de aplicación.

En muchas instancias, el cliente y el servidor pueden estar mucho tiempo en comunicación. El cliente puede hacer un conjunto de solicitudes, y el servidor debe responder a este tipo de solicitudes. Dependiendo de la naturaleza de las solicitudes pueden ser unas tras otras, en intervalos regulares o esporadicamente.

Como ingeniero, debo ser capaz de elegir una forma de usar el protocolo TCP para usarlo inteligentemente, ¿Debo hacer una conexión TCP para todos los mensajes? ¿o genero una conexión TCP para cada uno de los objetos?

En el primer caso, estamos hablando de una conexión HTTP persistente. En cambio con la segunda, estamos hablando de una conexón HTTP no persistente.[3]

HTTP persistente vs No persistente

El HTTP no persistente debe iniciar una sesión de TCP por cáda comando GET de HTTP. En el ejeplo planteado en la filmina:

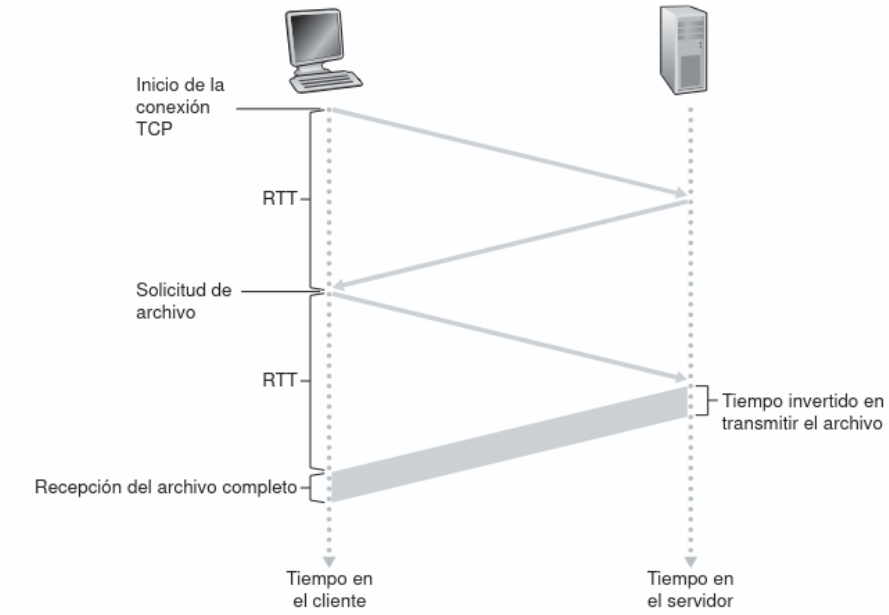
1. El cliente HTTP inicia una sesión TCP con el servidor HTTP.
2. El servidor HTTP responde ese mensaje TCP aceptando.
3. El cliente manda un mensaje de GET al servidor HTTP junto a la dirección del objeto que desea.
4. El servidor devuelve una respuesta, que contiene el objeto deseado (si todo sale bien)
5. El servidor cierra sesión

[3] En HTTP 1.0, por defecto usamos HTTP no persistente, en HTTP 1.1, por defecto usamos persistente

6. El cliente HTTP recibe el mensaje de respuesta que contiene el archivo HTML y despliega el HTML. Luego ve las referencias que contiene el archivo (imagenes, videos, gif's, etc).

7. Repetir los pasos anteriores hasta que no haya referencias para obtener.

Ahora, en el caso de que solo sea una pagina plana, tendríamos 2 interacciones de cliente y servidor, junto con el tiempo de transmisión de la pagina. En la imagen 2RTT + Tiempo de transmisión de la pagina.



Los modelos persistentes

Dado que requiere 2RTT + Tiempo de transmisión de pagina / objeto, no es tan eficiente para paginas grandes, además, implica que el sistema operativo debe usar recursos para cada conexión TCP.

Para estos casos, sirve hacer servidores HTTP persistentes. Se deja abierta la conexión TCP y se hacen sucesivas solicitudes y respuestas.

Hay dos tipos de conexiones persistentes, las que tienen pipelining y las que no. Las que no tienen pipelining mandan un GET y esperan a recibir el elemento para poder mandar otro GET. En cambio, con pipelining se mandan todas las solicitudes seguidas una vez encuentro una referencia. Esto permite acortar los tiempos, y es estandar en HTTP 1.1.

Ambos en el peor caso ocupan 1 RTT. Mientras que en el mejor caso, la conexión con pipelining puede llegar a costar menos.

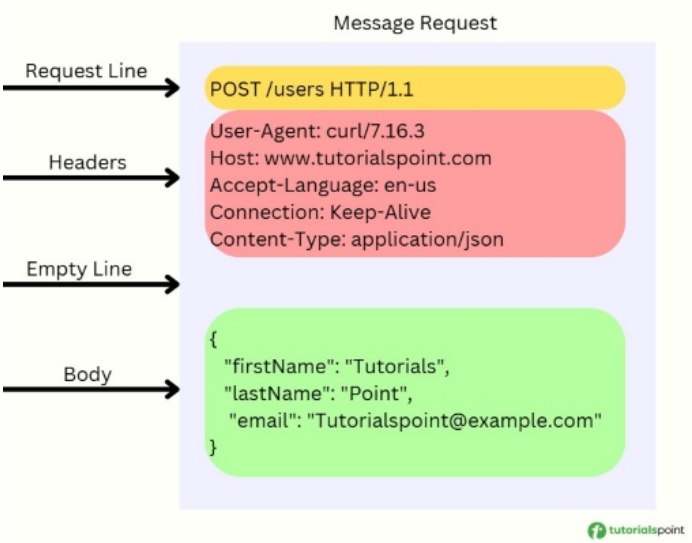
Se puede llegar a elegir la conexión HTTP sin pipelining para no sobrecargar el flujo si hay poca conexión a internet -preguntar, no estoy seguro pero parece plausible.

Como dato en color, hoy en día, en HTTP 2.0, se puede entrelazar múltiples solicitudes y respuestas en la misma conexión, y proporcionar también un mecanismo para priorizar los mensajes de solicitud y respuesta HTTP dentro de dicha conexión.

1.4. Formato de mensajes HTTP

Hay dos tipos de mensajes HTTP. Request y response. Todos funcionan en ASCII.

Formato de HTTP request



El HTTP Request tiene primero la request line, la cual posee:

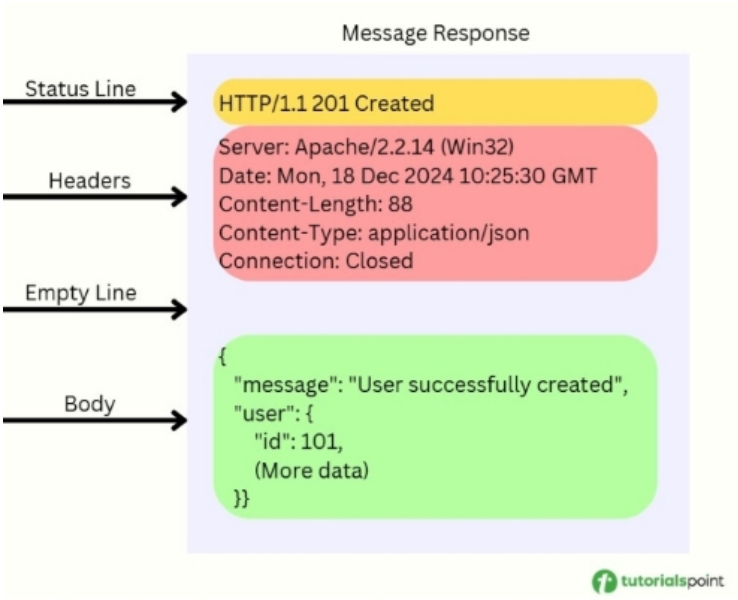
- La acción que se quiere hacer
- El Path de el archivo que necesitamos
- La versión de HTTP

Luego tenemos el header, este contiene:

- Host: Si bien parece que no hay necesidad de usar esto, ya que por TCP se conocen ambos, se va a usar en cache.
- User agent: Sirve para agregar información sobre qué navegador está usando el host. Esto para poder utilizar una visualización dependiendo que navegador se usa.

- Connection: Sirve para establecer si se usa una conexión persistente o no.
- Lenguage: Dice que lenguaje acepto, para que me den una pagina web con mi lenguaje de preferencia en lo posible.

Formato de HTTP response



Se divide en linea de estatus y esta contiene:

- Protocolo.
- Código de estado (vamos a ver más a detalle en la siguiente subsección).
- Resumen sobre el código de estado.

Encabezado:

- Conexíon: Habla sobre si la conexión TCP se cierra o no.
- Date: El momento en el que se hizo la solicitud al servidor.
- Server: Habla sobre el tipo de servidor que se está corriendo. Analogo a user agent.
- Last modified: Habla sobre la ultima vez que fue modificado el objeto. Sirve para usarse con cache. Si llego a necesitar la información y no cambió, puedo usar todos los archivos que ya tenía y no tengo por qué pedir más.
- Content-Lenght: La cantidad de bytes del objeto requerido.
- Content-Type: El tipo del objeto requerido.

Luego del header vienen todos los datos de lo solicitado.

Codigos HTTP de respuesta

Los codigos de respuesta se pueden ver de la siguiente forma:

- 1xx - Informativo: La solicitud está en curso o hay otro paso que dar.
- 2xx - Exito: La solicitud tuvo éxito. El servidor está enviando los datos que solicitó el cliente.
- 3xx - Redireccionamiento: El servidor le está diciendo al cliente un URI diferente al que debe redirigirse. Los encabezados generalmente contendrán un encabezado de ubicación con el URI actualizado. Diferentes códigos le dicen al cliente si un redireccionamiento es permanente o temporal.
- 4xx - Error del Cliente: El servidor no entendió la solicitud del cliente, o no puede o no quiere procesarla. Los diferentes códigos le dicen al cliente si fue un URI incorrecto, un problema de permisos u otro tipo de error.
- 5xx - Error del Servidor : Algo salió mal del lado del Servidor.

1.5. Las cookies: Una aplicación de HTTP.

Para hablar sobre cookies, primero debemos saber cual es la motivación de estas: HTTP, por diseño no recuerda nada entre un request y el siguiente. Imaginemos que necesitamos hacer un carrito de compras, pero no sabemos cómo guardar la información de lo que quiere comprar el cliente ¿Qué deberíamos hacer? Podemos usar cookies.

¿Qué es una cookie?

Una cookie es información generada por el servidor web que se guarda en la computadora del usuario y que el navegador envía automáticamente al servidor en futuras solicitudes

Nos puede servir para:

- Identificar a un usuario
- Mantener un carrito de compras
- Recordar una sesión iniciada

Una cookie esta compuesta por 4 cosas:

1. Encabezado con *Set-Cookie en la respuesta HTTP.*
2. *Encabezado con Cookie en la request HTTP.*
3. *Un archivo de cookies en la máquina del usuario*
4. *Una base de datos del sitio web*

1.6. Servidores proxy: Otra aplicación de HTTP.

Una web cache (también llamada proxy cache) es un servidor intermedio que se ubica entre el cliente (navegador) y el servidor web original Su principal función es: Responder solicitudes web usando copias locales de objetos previamente descargados, evitando contactar al servidor original cuando no es necesario. Bajo este punto de vista, el cliente no habla directamente con el servidor web si no que habla con el proxy el proxy decide si puede responder o si debe reenviar la solicitud.

El servidor proxy, es un servidor que actua como cliente y servidor a la vez. Y este:

- Reduce el tiempo de respuesta.
- Reduce el tráfico en el enlace de acceso.
- Beneficia a proveedores pequeños: Internet con caches permite entregar contenido eficientemente sin grandes infraestructuras.

1.7. Cache

La idea del cache es almacenar "localmente" datos ya solicitados y así poder acceder a éstos más rápidamente en el futuro.

- Un problema que debe atender el cache es la obsolescencia que puede tener los datos locales.
- El cache puede usar tiempos de expiración, o consultar a la fuente por vigencia del dato local.

1.8. Get condicional

El GET condicional aparece para resolver un problema propio de las cachés: La caché puede tener una versión vieja de un objeto. Por lo tanto, se creó un tipo de GET especial, un GET con condición, en el que preguntamos: ¿Este objeto está actualizado?

1. If-Modified-Since: fecha
2. Si esto está actualizado: HTTP/1.0 304 Not Modified
3. Si esto no fue actualizado: HTTP/1.0 200 OK (Vuelve con el archivo).

Qué gana el sistema con esto
Según la lógica de las filminas:

- Menos tráfico de red
- Menor tiempo de respuesta
- Menor carga para el servidor web
- La caché sigue siendo consistente

2. Una pequeña introduccion de API's

API viene de Interfaz de programación de aplicaciones. Y son un conjunto de reglas o protocolos que permiten a una aplicación informatica interactuar con otra. Un ejemplo, imaginemos que quiero hacer un inicio de sesión en un navegador, ya que ingresé mi usuario

1. Asumimos que ya tenemos una conexión TCP en este ejemplo.
2. Hago click en el navegador.
3. Se manda un mensaje de request por internet.
4. Llega a la API. Esta API le pregunta a la BDD si el usuario es correcto.
5. Si el mensaje es correcto, se genera un mensaje de respuesta diciendo que es correcto, si no es correcto se genera otro.

Las API's pueden ser:

- Privadas.
- Compartidas con un socio.
- Publicas.

Pero ¿Cual es su necesidad?

- Reducir esfuerzo manual: Imaginemos que hacemos una aplicación que muestra todos los tweets de un usuario en el pasado. Debería tener que buscar todos los tweets nuevos cada cierto tiempo, y es mucho gasto. Para esto nos ayuda la API, esta nos dara información cada vez que la persona twittee.
- Se automatiza todo: El sistema complejo se convierte en un conjunto de sistemas más chicos, los cuales operan todo el tiempo.

Las APIs HTTP (como las APIs REST) utilizan los métodos que conocemos de HTTP (GET, POST, PUT, DELETE) para operar sobre recursos.

La intención de las APIs es permitir y automatizar la comunicación entre distintos sistemas a través de la red. En este contexto, se prefiere usar el protocolo HTTP para interactuar con recursos que se encuentran en otra computadora (o incluso en la misma), en lugar de utilizar llamadas directas a métodos como en el paradigma orientado a objetos.

Esto se debe a que HTTP está diseñado para comunicación distribuida, desacoplada y estandarizada, mientras que los objetos y sus métodos están pensados para ejecutarse dentro del mismo proceso.

En las operaciones de HTTP la intención es la siguiente:

- GET: Se manda una consulta sobre el elemento que se ingresa en la URL. No debería cambiar nada en el servidor y deberia poder repetirse sin consecuencias.
- POST: Produce un nuevo recurso. Luego de cada llamada debería haber un nuevo recurso. Se envian datos en el body.
- PUT: Modifica el estado de un recurso. Apunta a un recurso en especifico.
- DELETE: Elimina un recurso en especifico.