Control de Versiones

Ejercicio N°3

Objetivos	● Diseño y construcción de sistemas con acceso distribuido ● Encapsulación de Threads y Sockets en Clases ● Definición de protocolos de comunicación ● Protección de los recursos compartidos ● Uso de buenas prácticas de programación en C++
Instancias de Entrega	**Entrega 1:** clase 8 (01/05/2018). **Entrega 2:** clase 10 (15/05/2018).
Temas de Repaso	● Definición de clases en C++ ● Contenedores de STL ● Excepciones / RAII ● Move Semantics ● Sockets ● Threads
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Eficiencia del protocolo de comunicaciones definido ● Control de paquetes completos en el envío y recepción por Sockets ● Atención de varios clientes de forma simultánea ● Eliminación de clientes desconectados de forma controlada

Índice

Introducción

A fin de reducir el tráfico de datos, y poder estudiarlo, una importante empresa multinacional nos solicita prototipar un sistema reducido de control de versiones .

Descripción

Nuestro cliente nos solicita un sistema centralizado de control de versiones. Este sistema consta de un único servidor, que acepta conexiones entrantes de clientes, y múltiples clientes que se conectarán y podrán interactuar con el servidor de forma concurrente.

El prototipo solicitado permitirá al cliente subir diferentes versiones de archivos locales, marcar (tag) algunas versiones en particular y descargar las versiones particulares de los archivos según un tag.

Formato de Línea de Comandos

Servidor

```
./server <puerto/servicio> <indice>
```

Donde `<puerto/servicio>` es el puerto TCP (o servicio) en donde estará escuchando la conexiones entrantes.
El parámetro `<indice>` es el nombre del archivo que contiene el índice. El servidor deberá leerlo al inicio y reescribirlo con el contenido actualizado al finalizar.

Cliente

El cliente tendrá diferentes modos de uso:

```
./client <ip/hostname> <puerto/servicio> push <archivo> <hash>
```

El cliente se conectará al servidor corriendo en la máquina con dirección IP `<ip>` (o `<hostname>`), en el puerto (o servicio) TCP `<puerto/servicio>`.
En este modo, el cliente **subirá (push)** el archivo local `<archivo>` al servidor.
El parámetro `<hash>` es el identificador del contenido del archivo y equivale a “la versión” del archivo. De esta manera un cliente podría ir subiendo distintas versiones de un mismo archivo bajo distintos hashes. A fines prácticos del TP, el hash es sólo un string.

```
./client <ip/hostname> <puerto/servicio> tag <hash> [<hash> ...]
```

En este modo, el cliente **marcará (tag)** un o una serie de hashes en el servidor. La cantidad de hashes está dada por la cantidad de argumentos.

```
./client <ip/hostname> <puerto/servicio> pull <tag>
```

En este modo, el cliente **descargará (pull)** el o los archivos desde el servidor y los guardará localmente.
El parámetro `<tag>` le indicará al servidor que hashes se están nombrando y cada hash a su vez indica que archivo se quiere descargar.

Códigos de Retorno

En cualquier caso, tanto cliente como servidor deben retornar 0.

Entrada y Salida

Servidor

El servidor no imprimirá nada por salida estándar.

En cambio, esperará el caracter `'q'` por entrada estándar. Cuando lo reciba, el servidor deberá finalizar todas las conexiones y cerrarse.

Cliente

El cliente imprimirá por salida estándar cualquier error que encuentre.

Imprimirá `"Error: argumentos invalidos.\n"` si:
- El modo a ejecutar es distinto de `push`, `tag` o `pull`
- Se reciben un número incorrecto de parámetros (nótese que la cantidad depende del modo del cliente ya que no son los mismos argumentos para un `push` que para un `pull`).

Imprimirá `"Error: archivo inexistente.\n"` si:
- El archivo a subir (`push`) no existe.

Imprimirá `"Error: tag/hash incorrecto.\n"` si:
- Durante la ejecución del modo `tag`, se encuentra que el tag ya está creado y guardado en el servidor. No pueden haber tags duplicados.
- Durante la ejecución del modo `tag`, se encuentra que alguno de los hashes no fue previamente creado (`push`) en el servidor. En otras palabras, todos los hashes a ser marcados deben existir previamente.
- El tag no existe cuando se quiere hacer un `pull`.

Nótese que si realizar un `push` de un hash existente no es un error. De hecho el cliente no realiza ninguna acción y no transfiere el archivo al servidor ya que asume que el archivo existe allí.

El cliente no recibirá nada de la entrada estándar.

Formato de archivos

El servidor tendrá su índice guardado en un archivo. Este será leído por el servidor al momento de inicializarse y será guardado (actualizado) al momento de cerrarse.
El índice contiene qué archivos están bajo el control del servidor y cuales son sus hashes asi como tambien que tags existen y cuales son los hashes bajo dicho tag.

El formato del archivo consta de cero o más líneas de la forma:

```
<type> <name> hash <hash ...> ;
```

Nótese que cada línea termina en un punto y coma y que todos los componentes de la línea están separados por 1 o más espacios en blanco.

El campo `<type>` indica el tipo de entrada en el índice. Puede ser el valor **`t`** indicando que `<name>` es el nombre se un tag o **`f`** indicando que **<name>** es el nombre de un archivo.

Esta garantizado que en el índice aparecerán todos los campos de tipo **`f`** (archivos) antes que cualquier campo de tipo **`t`** (tags).
También se garantiza que todos los índices estarán bien formados.

El servidor no debe asumir ningún orden en las líneas ni tampoco en los hashes, asá como tampoco debe asumir la cantidad de espacios en blanco hay entre cada campo o entre cada línea.

Sin embargo, a la hora de guardar el índice, el servidor debe generar un archivo más estricto:

- Las líneas se separan con un único salto de línea.
- Los campos se separar con un único espacio.
- Los hashes en una línea se guardaran en orden alfabético.
- Las líneas de archivos se guardaran en orden alfabético por nombre de archivo. (1)
- Las líneas de tags se guardaran en orden alfabético por nombre de tag. (1)

(1) Primero se guardaran todas las líneas de archivos en orden y luego todas las líneas de tags en orden.

Protocolo de comunicación

Formato del mensaje

Estos son los delineamientos generales:

- Todas las cantidades y longitudes se enviarán en enteros de 4 bytes sin signo en big endian.
- Todos los strings, incluidos los archivos, se enviarán ***sin*** el delimitador `'\0'`, enviando primero (prefijo) la longitud del string.
- Los comandos o acciones a realizar así como también los códigos de retorno se enviarán en enteros de 1 bytes sin signo.

Push

El cliente quiere subir (`push`) un archivo y su hash.
Para eso le envía al servidor el siguiente mensaje:

- El comando o acción **`push`** codificado con el valor numérico **`1`**. En otras palabras, envía el número **`1`**.
- Luego envía el nombre del archivo. Como se indicó en la sección **Formato del mensaje**, para enviar un string se debe enviar primero su longitud (4 bytes, sin signo, big endian) y luego el contenido del string sin el `'\0'`.
- Luego se envía el hash. (otro string).

El servidor deberá responder con un código de retorno (1 byte, sin signo) con un valor numérico igual a **`0`** si el hash ya existe o **`1`** sino.

Si el hash ya existe, el cliente finaliza sin realizar ninguna otra acción.
En cambio, si no existe, el cliente continúa con la transferencia del archivo enviandole al servidor:

- El contenido del archivo. Como se indicó en la sección **Formato del mensaje**, un archivo no es más que un string y por lo tanto se debe enviar primero su longitud (4 bytes, sin signo, big endian) y luego el contenido del mismo.

Tag

El cliente quiere marcar (`tag`) una serie de hashes.
Para eso le envía al servidor el siguiente mensaje:

- El comando o acción **`tag`** codificado con el valor numérico **`2`**.
- La cantidad de hashes. Como se indicó en la sección **Formato del mensaje**, una cantidad se codifica con 4 bytes, sin signo y en big endian.
- El nombre del tag (un string).
- Los hashes, enviando un hash a la vez (o sea, un string a la vez).

El servidor deberá responder con un código de retorno con un valor numérico igual a **`1`** si todo salió bien o **`0`** sino (el nombre del tag está duplicado, o algunos de los hashes no existe).

Pull

El cliente quiere descargar (`pull`) una serie de archivos.
Para eso le envía al servidor el siguiente mensaje:

- El comando o acción **`pull`** codificado con el valor numérico **`3`**.
- El nombre del tag.

El servidor deberá responder con un código de retorno con un valor numérico igual a **`1`** si todo salió bien o **`0`** sino (el nombre del tag no existe).

Si todo salio bien, el servidor continúa enviandole al cliente:

- La cantidad de archivos que están taggeados.
- Para cada archivo se envía:
    - El nombre del archivo
    - El contenido del archivo.

Ejemplo

Inicialmente el archivo `indice.txt` contiene lo siguiente:

```
f B 1bb421 1bb433 ;
```

Y en el servidor hay un archivo llamado `1bb421` que contiene:

```
46 69 6c 65 20 42 2c 20  63 6f 6e 74 65 6e 74 20  |File B, content |
32 0a                                             |2.              |
```

Y hay otro archivo llamado `1bb433` que contiene:

```
46 69 6c 65 20 42 2c 20  63 6f 6e 74 65 6e 74 20  |File B, content |
33 0a                                             |3.              |
```

Nótese como en el servidor los archivos están guardados con nombre igual al hash.
Y es en el archivo `indice.txt` en donde se relacionan los hashes con el nombre del archivo original.
En este caso hay un único archivo llamado `B` que tiene 2 versiones `1bb421` y `1bb433.`

Luego, el servidor se inicializa con:

```
./server 8081 indice.txt
```

Durante la inicialización, el servidor cargará el índice. Se garantiza que todos los índices estarán bien formados.

Luego, un cliente quiere hacer un `push` y ejecuta:

```
./client 127.0.0.1 8081 push A a14242
```

Donde el archivo A existe y tiene como contenido:

```
46 69 6c 65 20 41 2c 20  63 6f 6e 74 65 6e 74 20  |File A, content |
31 0a                                             |1.              |
```

Entonces el cliente envía el comando `push` (1 byte), el nombre de archivo
(4 bytes para la longitud más el texto en sí)

```
01 00 00 00 01 41 00 00  00 06 61 31 34 32 34 32  |.....A....a14242|
```

El servidor responde que no hubo ningún problema (1 byte)

```
01                                                |.               |
```

El cliente entonces termina la operación enviandole el archivo

```
00 00 00 12 46 69 6c 65  20 41 2c 20 63 6f 6e 74  |....File A, cont|
65 6e 74 20 31 0a                                 |ent 1.          |
```

El servidor entonces recibe el archivo y lo guarda localmente bajo el nombre `a14242` y actualiza su índice.

Luego, un cliente decide marcar ciertos hashes bajo un mismo nombre (`tag`) y se ejecuta:

```
./client 127.0.0.1 8081 tag v1.0 a14242 1bb421
```

El cliente envía el comando `tag` (1 byte), la cantidad de hashes que van a ser taggeados (4 bytes) y el nombre del tag.
Luego, envía cada uno de los hashes:

```
02 00 00 00 02 00 00 00  04 76 31 2e 30 00 00 00  |.........v1.0...|
06 61 31 34 32 34 32 00  00 00 06 31 62 62 34 32  |.a14242....1bb42|
31                                                |1               |
```

El servidor verifica que el tag `v1.0` no existe previamente y que todos los hashes (`a14242 1bb421`) si existen.

El servidor responde que todo estuvo en orden (1 byte)

```
01                                                |.               |
```

El cliente recibe este código de retorno y finaliza.

Luego, un cliente decide descargarse todos los archivos asociados a un tag (`pull`) y ejecuta:

```
./client 127.0.0.1 8081 pull v1.0
```

El cliente envía el comando `pull` (1 byte) seguido por el tag

```
03 00 00 00 04 76 31 2e  30                      |.....v1.0       |
```

El servidor responde que todo está en orden (1 byte) ya que existe dicho tag.
Luego envía la cantidad de archivos que están taggeados (4 bytes), y por cada archivo envía su nombre y el contenido del mismo.

```
01 00 00 00 02 00 00 00  01 42 00 00 00 12 46 69  |.........B....Fi|
6c 65 20 42 2c 20 63 6f  6e 74 65 6e 74 20 32 0a  |le B, content 2.|
00 00 00 01 41 00 00 00  12 46 69 6c 65 20 41 2c  |....A....File A,|
20 63 6f 6e 74 65 6e 74  20 31 0a                  | content 1.     |
```

El cliente recibirá cada archivo y lo guardará localmente bajo el nombre **`<nombre>.<tag>`**. En este caso el cliente recibirá 2 archivos y los guardará bajo los nombres `B.v1.0` y `A.v1.0`.

Luego, si se le envía por entrada estándar al servidor el caracter `'q'`, este guardará su estado en el archivo `indice.txt` y finalizará.

El archivo `indice.txt` final tendrá el siguiente contenido (nótese el orden):

```
f A a14242 ;
f B 1bb421 1bb433 ;
t v1.0 1bb421 a14242 ;
```

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C++11.
2. Se debe aplicar herencia y polimorfismo en alguna parte del TP.
3. Se debe implementar la sobrecarga del operador << en alguna parte del TP.
4. La aplicación debe soportar clientes en simultáneo.

Referencias

[1] std::map: http://www.cplusplus.com/reference/map/map/