Opty: control concurrencia optimista*

Federico C. Repond
frepond@unq.edu.ar

Esteban Dimitroff Hódi
esteban.dimitroff@unq.edu.ar

15 de abril de 2016

Introducción

En este ejercicio vamos a implementar un servidor de transacciones utilizando control de concurrencia optimista. Vamos a aprender como utilizar una estructura de datos actualizable en Erlang que puede ser accedido por procesos concurrentes, posiblemente distribuidos. Antes de empezar demos conocer como funciona el control de concurrencia optimista con validación hacia atrás (backwards validation).

1. La Arquitectura

La arquitectura consiste en un servidor con acceso a un store y a un validator. El store está formado por un conjunto de entradas, cada una manteniendo una referencia única. La referencia es actualizada en cada operación de escritura así podemos validar que la entrada no se modificó desde la última vez que la leímos.

Un cliente empieza una transacción creando un handler de transacción que le da acceso al store y al proceso de validación. El servidor de transacciones no se involucra en la transacción, simplemente es un proceso por medio del cual obtenemos acceso al store y al proceso de validación.

1.1. El Handler de Transacciones

El handler de transacciones (`handler`) va a manejar los pedidos de lectura y escritura desde un cliente y cerrar la transacción. En cada operación de lectura el handler mantiene registro de la referencia única de la entrada leída. También mantiene la operación de escritura en un store local de forma tal que el store real no sea modificado hasta que cerremos la transacción.

Cuando la transacción se cierra el handler envía los conjuntos de lectura y escritura al validator.

*Adaptado al español del material original de Johan Montelius (`https://people.kth.se/~johanmon/dse.html`)

1.2. El Validator

El proceso de validación va a ser capaz de decir si un valor de una entrada fue cambiado desde que la transacción leyó la entrada. Si ninguna de las entradas cambió la transacción puede hacer commit y las operaciones de escritura asociadas llevadas a cabo.

Dado que hay un solo validator en el sistema, el mismo es l único proceso que escribe algo en el store.

2. La Implementación

Dado que las estructuras de datos en Erlang son inmutables (no podemos cambiar su valor), necesitamos hacer un truco. El store va a ser representado como una tupla de identificadores de procesos. Cada proceso es una entrada y podemos cambiar su valor enviando mensajes `write`.

Un handler de transacciones recibe una copia de la tupla cuando es creado pero los procesos que representan entradas por supuesto no son copiadas. El store entonces puede ser compartido por varios handlers de transacciones, todos los que envían un mensaje `read` a las entradas.

El validator no necesita acceder a todo el store dado que reciben los conjuntos de lecturas y escrituras desde los handlers de transacciones. Estos conjuntos contienen los identificadores de procesos de las entradas relevantes.

2.1. Una Entrada

Un proceso que implementa una entrada debe tener un valor y una referencia única como estado. Vamos a llamar a esta referencia más adelante "timestamp" pero no tiene nada que ver con tiempo, simplemente va a ser una referencia única. La referencia va a ser dada al que lee el valor de forma tal que el validator pueda determinar después si el valor fue modificado. Podríamos hacerlo sin la referencia pero tiene sus ventajas.

```
-module(entry).
-export([new/1]).

new(Value) ->
    spawn_link(fun() -> init(Value) end).

init(Value) ->
    entry(Value, make_ref()).
```

Notar que estamos usando la primitiva `spawn_link/1` para asegurarnos que si el creador de la entrada muere, entonces la entrada también muere.

La entrada debe manejar 3 mensajes:

- `{read, Ref, Handler}`: un pedido de un handler tageado con una referencia. Debemos devolver un mensaje tageado con una referencia de forma

tal que el handler pueda identificar el mensaje correcto. La respuesta va a contener el identificador de proceso de la entrada, el valor y el timestamp actual.

- `{write, Value}`: cambia el valor actual de la entrada, no requiere una respuesta. El timestamp de la entrada es actualizado.
- `{check, Ref, Read, Handler}`: verifica si el timestamp de la entrada desde que leímos el valor cambio. La respuesta es tageada con la referencia, va a indicar al handler si el timestamp es todavía el mismo o si tiene que abortar.
- `stop`: termina.

Hablamos del handler y no del cliente, esto va a ser claro más adelante. ¿Por qué queremos que el mensaje `read` incluya el identificador de proceso? No es claro en este momento pero vamos a ver que simplifica escribir un handler de transacciones asincrónico.

```
entry(Value, Time) ->
    receive
        {read, Ref, Handler} ->
            Handler ! {Ref, self(), Value, Time},
            entry(Value, Time);
        {check, Ref, Read, Handler} ->
            if
                Read == Time ->
                    Handler ! {Ref, ok};
                true ->
                    Handler ! {Ref, abort}
            end,
            entry(Value, Time);
        {write, New} ->
            entry(New, make_ref());
        stop ->
            ok
    end.
```

2.2. El Store

Vamos a ocultar la representación del store y proveer solo una API para crear un nuevo store y buscar una entrada. La creación de una tupla se hace simplemente creando una lista de los identificadores de procesos primero y convirtiéndolos en tuplas.

```
-module(store).
-export([new/1, stop/1, lookup/2]).
```

```
new(N) ->
    list_to_tuple(entries(N, [])).

stop(Store) ->
    lists:map(fun(E) -> E ! stop end, tuple_to_list(Store)).

lookup(I, Store) ->
    element(I, Store). % this is a builtin function

entries(N, Sofar) ->
    if
        N == 0 ->
            Sofar;
        true ->
            Entry = entry:new(0),
            entries(N - 1, [Entry | Sofar])
    end.
```

2.3. El Handler de Transacciones

Un cliente nunca debe acceder el store directamente. Va a realizar todas las operaciones por medio de un handler de transacciones. Un handler de transacciones se crea para un cliente específico y mantiene el store y el identificador de proceso del proceso validator.

Vamos a implementar el handler de forma tal que el cliente pueda hacer `read` asincrónicos al store. Si la latencia es alta no tiene sentido esperar que termine una operación `read` antes de empezar una segunda operación.

La tarea del handler de transacciones es registrar todas las operaciones de lectura (y cuando se hicieron) y hacer las transacciones de escritura solo visibles al store local. Para hacer esto el handler va a mantener 2 conjuntos: el conjunto de lecturas (`Read`) y el conjunto de escrituras (`Write`). El conjunto de lecturas es una lista de tuplas `{Entry, Time}` y el conjunto de escrituras es una lista de tuplas `{N, Entry, Value}`. Cuando es el momento de hacer commit, el handler envía los conjuntos de lectura y escritura al validator.

Cuando el handler se crea se linkea con su creador. Esto significa que si alguno muere entonces ambos mueren. Esto suena fuerte pero se va a explicar cuando implementemos el servidor.

```
-module(handler).
-export([start/3]).

start(Client, Validator, Store) ->
    spawn_link(fun() -> init(Client, Validator, Store) end).

init(Client, Validator, Store) ->
    handler(Client, Validator, Store, [], []).
```

La interfaz del handler es la siguiente:

- `{read, Ref, N}`: un pedido de lectura de un cliente contiene una referencia que debemos usar en el mensaje de respuesta. Un entero `N` que el índice de la entrada en el store. El handler debe primero mirar el conjunto de escrituras y ver si la entrada `N` ha sido escrita. Si no encontramos una operación de escritura se envía un mensaje al proceso de la N-ésima entrada del store. Esta entrada va a contestar al handler dado que debemos registrar el tiempo de lectura.
- `{Ref, Entry, Value, Time}`: una respuesta de una entrada debe ser reenviada al cliente. La entrada y el tiempo se guarda en el conjunto de lecturas del handler. La respuesta al cliente es `{Ref, Value}`.
- `{write, N, Value}`: un mensaje de escritura de un cliente. El entero `N` es el índice de la entrada en el store y `Value` el valor. La entrada con índice `N` y el valor son guardados en el conjunto de escrituras del handler.
- `{commit, Ref}`: un mensaje de commit del cliente. Este es el momento de contactar el validator y ver si hay algún conflicto en nuestro conjunto de lectura. Si no lo hay, el validator va a efectuar las operaciones de escritura en el conjunto de escrituras y responder al cliente.

Este es el esqueleto del handler.

```
handler(Client, Validator, Store, Reads, Writes) ->
    receive
        {read, Ref, N} ->
            case lists:keysearch(N, 1, Writes) of
                {value, {N, _, Value}} ->
                      :
                    handler(Client, Validator, Store, Reads, Writes);
                false ->
                      :
                      :
                    handler(Client, Validator, Store, Reads, Writes)
            end;
        {Ref, Entry, Value, Time} ->
              :
            handler(Client, Validator, Store, [...|Reads], Writes);
        {write, N, Value} ->
            Added = [{N, ..., ...}|...],
            handler(Client, Validator, Store, Reads, Added);
        {commit, Ref} ->
            Validator ! {validate, Ref, Reads, Writes, Client};
        abort -> ok
    end.
```

2.4. La Validación

El validator es responsable de hacer la validación final de las tansacciones. La tarea se hace bastante fácil dado que solo una transacción es válida al mismo tiempo. No hay operaciones concurrentes que pueden entrar en conflicto con el proceso de validación.

Cuando iniciamos el validator lo linkeamos también al proceso que lo crea. Esto es para asegurarnos de no dejar ningún proceso zombie.

```
-module(validator).
-export([start/0]).

start() ->
    spawn_link(fun() -> init() end).

init()->
    validator().
```

El validator recibe pedidos desde un cliente que tiene todo lo necesario para validar que la transacción puede realizar todas las operaciones de escritura que van a ser el resultado de la transacción. El pedido contiene:

- `Ref`: una referencia única que taggea el mensaje de respuesta.
- `Reads`: un lista de operaciones de lectura que fueron realizadas. El validator debe asegurarse que las entradas del conjunto de lecturas no fue cambiado.
- `Writes`: la operaciones pendientes de escritura, que si la transacción es válida, se aplicaran al store.
- `Client`: el identificador del cliente que quien debemos responder.

La validación entonces es simplemente chequear que las operaciones de lectura todavía son válidas y si es así actualizar el store con las operaciones de escritura pendientes.

```
validator() ->
    receive
        {validate, Ref, Reads, Writes, Client} ->
            case  validate(Reads) of
                ok ->
                    update(Writes),
                    Client ! {Ref, ok};
                abort ->
                    Client ! {Ref, abort}
            end,
            validator();
        _Old ->
            validator()
    end.
```

Dado que cada operación de lectura es representado como una tupla `{Entry, Time}` el validator solo necesita enviar un mensaje a la entrada para asegurarse que el timestamp actual de la entrada es el mismo.

```
validate(Reads) ->
    {N, Tag} = send_checks(Reads),
    check_reads(N, tag).
```

Para una mejor performance el validator puede enviar primero los mensajes para chequear todas las entradas y luego recolectar las respuestas. Ni bien una de las entradas responde con un mensaje de `abort`, terminamos, Notar sin embargo, que debemos ser cuidadosos de que no estamos recibiendo mensajes que pertenecen a una validación previa. Cuando enviamos un el pedido de chequeo debemos taggearlos con una referencia única así sabemos que estamos analizando las respuestas correctas.

```
send_checks(Reads) ->
    Tag = make_ref(),
    Self = self(),
    N = length(Reads),
    lists:map(fun({Entry, Time}) ->
                      Entry ! {check, Tag, Time, Self}
              end,
              Reads),
    {N, Tag}.
```

Recolectar las respuestas es simple, solo debemos contemplar qye no recolectamos una respuesta vieja.

```
check_reads(N, Tag) ->
    if
        N == 0 ->
            ok;
        true ->
            receive
                {Tag, ok} ->
                    check_reads(N - 1, Tag);
                {Tag, abort} ->
                    abort
            end
    end.
```

Los mensajes viejos que queden encolados deben ser removidos de alguna forma, por eso es muy importante, que el loop principal del validator incluya una cláusula para capturar todo.

2.5. El Servidor

Ahora que tenemos todas las piezas para construir el servidor de transacciones. El servidor va a construir el store y un proceso validator. Los clientes pueden entonces abrir una transacción y cada transacción va a recibir un nuevo `handler` que realizará la tarea.

```
-module(server).
-export([start/1, open/1, stop/1]).

start(N) ->
    spawn(fun() -> init(N) end).

init(N) ->
    Store = store:new(N),
    Validator = validator:start(),
    server(Validator, Store).
```

El servidor va a esperar pedidos de los clientes y hacer `spqwn` de un nuevo handler y validator de transacciones. Sin embargo hay una trampa aquí en la que no queremos caer. Si el servidor crea el handler de transacciones debemos hacer que el handler sea independiente del servidor (no linkeado). De esta forma si el handler muere no queremos que el servidor muera. Por otro lado, si el cliente muere queremos que el handler muera. La solución es dejar que el cliente cree el handler. Es conveniente para la prueba implementar un cliente que tenga las operaciones: `read`, `write`, `commit`, `abort` y quizás una `state` extendiendo el handler para que nos devuelva el estado para analizar.

```
open(Server) ->
    Server ! {open, self()},
    receive
        {transaction, Validator, Store} ->
            handler:start(self(), Validator, Store)
    end.
```

Esto tiene implicancias también en donde corre el handler de transacciones- Algo que vamos a discutir cuando terminemos con el servidor.

El servidor ahora es casi trivial.

```
server(Validator, Store) ->
    receive
        {open, Client} ->
             :
            server(Validator, Store);
        stop ->
             store:stop(Store)
    end.
```

Listo, implementamos un servidor de transacciones con control de concurrencia optimista!

3. Performance

¿Performa? ¿Cuántas transacciones podemos hacer por segundo? ¿Cuáles son las limitaciones en el número de transacciones concurrentes y la tasa de éxito? Esto por supuesto depende del tamaño del store, cuántas operaciones de write cada transacción hace, cuanto tiempo tenemos entre las instrucciones de read y el commit final. ¿Algo más?

Algunas preguntas sobre Erlang también son interesantes plantear. ¿Es realista la implementación del store que tenemos? Independientemente del handler de transacciones, ¿qué rápido podemos operar sobre el store? ¿Qué sucede si hacemos esto en una red de Erlang distribuida, qué es lo que se copia cuando el handler de transacciones arranca? ¿Dónde corre el handler? ¿Cuáles son los pros y contras de la estrategia de implementación?