

Opty: control concurrencia optimista^{*}

Federico C. Repond
frepond@unq.edu.ar

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

30 de marzo 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 accedida 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 almacenamiento y a un validador. El almacenamiento 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 almacenamiento 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 almacenamiento y al proceso de validación.

1.1. El Handler

El handler de transacciones 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 almacenamiento local de forma tal que el almacenamiento real no sea modificado hasta que cerremos la transacción.

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

^{*} Adaptado al español del material original de Johan Morellius (<https://people.kth.se/~johanmon/dse.html>)

1.2. El Validador

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 validador en el sistema, el mismo es el único proceso que escribe algo en el almacenamiento.

2. La Implementación

Dado que las estructuras de datos en Erlang son inmutables (no podemos cambiar su valor), necesitamos hacer un truco. El almacenamiento 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 almacenamiento entonces puede ser compartido por varios handlers de transacciones, todos los que envían un mensaje `read` a las entradas.

El validador no necesita acceder a todo el almacenamiento 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 validador 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 asíncronico.

```
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 Almacenamiento

Vamos a ocultar la representación del almacenamiento y proveer solo una API para crear un nuevo almacenamiento 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 almacenamiento 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 almacenamiento y el identificador de proceso del proceso validador.

Vamos a implementar el handler de forma tal que el cliente pueda hacer **read** asíncronos al almacenamiento. 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 almacenamiento 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 validador.

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 almacenamiento. 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 almacenamiento. 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 almacenamiento 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 validador y ver si hay algún conflicto en nuestro conjunto de lectura. Si no lo hay, el validador 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 validador es responsable de hacer la validación final de las transacciones. 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 validador 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 validador 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**: una lista de operaciones de lectura que fueron realizadas. El validador debe asegurarse que las entradas del conjunto de lecturas no fue cambiado.
- **Writes**: las operaciones pendientes de escritura, que si la transacción es válida, se aplicaran al almacenamiento.
- **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 almacenamiento 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 validador 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 validador 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}.
```