The Global Sequence Protocol

a Memory Model for Distributed Systems

Sebastian Burckhardt
`sburckha@microsoft.com`

Daan Leijen
`daan@microsoft.com`

Jonathan Protzenko
`protz@microsoft.com`

Distributed Memory

- A server along with multiple clients;
- Concurrent read and writes on the same data structure;
- Communication issues;
- Think of: memory on a modern processor; cloud storage and Google docs.

Question: what kind of abstraction do we offer to the programmer?

Distributed Memory

- A server along with multiple clients;
- Concurrent read and writes on the same data structure;
- Communication issues;
- Think of: memory on a modern processor; cloud storage and Google docs.

Question: what kind of abstraction do we offer to the programmer?

Answer: a log of $\boxed{\text{updates}}$.

A silly memory model

(But a good excuse to do some formalization)

Our system is: $\langle S, C \rangle$; S is the server, $C(i)$ are the clients.

We execute programs:

$$\begin{array}{rl} e ::= & \text{your-typical-}\lambda\text{-calculus} \\ & \textsf{perform}\ e \\ & \textsf{get}\ () \end{array}$$

Quick typing rules (σ is the type of state):

- $S = \vec{f}_s : \text{list } (\sigma \to \sigma)$
- $C(i) = e : \text{expr}$
- perform $: (\sigma \to \sigma) \to \text{unit}$
- get $: \text{unit} \to \sigma$

A silly memory model (2)

(But a good excuse to do some formalization)

Initially, $S = []$ and we assume $s_0 : \sigma$ is the initial (empty) state.

How does the system reduce? For a context $\mathcal{C}$ and a given client:

$$\begin{array}{ll} \langle \vec{f}_s; \mathcal{C}[\text{perform } f] \rangle & \rightsquigarrow \langle \vec{f}_s \cdot f; \mathcal{C}[()] \rangle \\ \langle \vec{f}_s; \mathcal{C}[\text{get } ()] \rangle & \rightsquigarrow \langle \vec{f}_s; \mathcal{C}[\text{fold}(s_0, \vec{f}_s)] \rangle \end{array}$$

A silly memory model (2)

(But a good excuse to do some formalization)

Initially, $S = []$ and we assume $s_0 : \sigma$ is the initial (empty) state.

How does the system reduce? For a context $\mathcal{C}$ and a given client:

$$\begin{array}{ll} \langle \vec{f}_s ; \mathcal{C}[\text{perform } f] \rangle & \leadsto \langle \vec{f}_s \cdot f ; \mathcal{C}[()] \rangle \\ \langle \vec{f}_s ; \mathcal{C}[\text{get } ()] \rangle & \leadsto \langle \vec{f}_s ; \mathcal{C}[\text{fold}(s_0, \vec{f}_s)] \rangle \end{array}$$

- perform f means: push a *functional* update
- get $()$ means: *compose* all updates to obtain the current state.

A silly memory model (3)

(But a good excuse to do some formalization)

This doesn't work.

- The programming model is great! Actually, it's linearizable. (Programmers love it!)
- But, implementing these operational semantics gives terrible performance (global lock + blocking IO)

A memory model either has strong consistency or good performance.

A better memory model (1)

Let's give up consistency for performance.

(a.k.a. let's put more stuff in-between $\langle \ldots \rangle$)

A most natural idea: local buffers of updates to improve performance.

(still not saying what σ is)

New operational model: $\langle S, C \rangle$

- $S = \vec{f}_s$: list $(\sigma \to \sigma)$ ("the server keeps a list of updates")
- $C(i) = (\vec{f}_l, e)$: list $(\sigma \to \sigma) \times$ expr ("the client keeps a local buffer of updates")

A better memory model (2)

Two updated transitions and a new one:

$$\begin{array}{ll}
\langle \vec{f}_s; \langle \vec{f}_l, \mathcal{C}[\text{perform } f] \rangle\rangle & \rightsquigarrow \langle \vec{f}_s; \langle \vec{f}_l \cdot f; \mathcal{C}[()] \rangle\rangle \\
\langle \vec{f}_s; \langle \vec{f}_l; \mathcal{C}[\text{get } ()] \rangle\rangle & \rightsquigarrow \langle \vec{f}_s; \langle \vec{f}_l; \mathcal{C}[\text{fold}(s_0, \vec{f}_s \cdot \vec{f}_l)] \rangle\rangle \\
\langle \vec{f}_s; \langle f \cdot \vec{f}_l; \mathcal{C}[e] \rangle\rangle & \rightsquigarrow \langle \vec{f}_s \cdot f; \langle \vec{f}_l; \mathcal{C}[e] \rangle\rangle
\end{array}$$

In cloud lingo: “the update has made it to the server”
In processor lingo: “the cache has been drained to the main memory”

The model is more relaxed (more behaviors): allows for a more efficient implementation (non-blocking) at the expense of a more complicated mental model.

A better memory model (3)

This formalization:

1. is abstract (instantiate σ with a memory store: get TSO)
2. is suitable for the programmer (claim)
3. is not suitable for the implementor (why?)

A word about orders

When talking about memory models, we like to order events.

Some (partial) orders:

- ar (arbitration order) is the final one everyone agrees on
- rb (returns before) is the side-channel, i.e. the "wall-clock" order (may or may not be observable)
- vis (visibility) means: if $(a, b) \in$ vis, then the update a from client 1 is visible to client 2 before it performs b
- so (session order) is the local (per-client) order
- hb (happens before) is so and vis

A better memory model: TSO

- The model is still eventually consistent (there is an ar)
- No longer linearizable (rb $\not\subseteq$ ar); no longer sequentially consistent (vis $\neq$ ar, a.k.a. there is no single order)
- "If I see things in this order, it's arbitrated in this order" (hb $\subseteq$ ar)
- "If I see things in this order, others see them in this order" (hb $\subseteq$ vis)

A formalization of TSO; an operational vision (as opposed to equational).

Sebastian Burckhardt.
Principles of Eventual Consistency
In *Foundations and Trends in Programming Languages*

Things you don't want (1)

Here's a sample execution.

$\vec{f}_s$	$\vec{f}_l$	e
$[]$	$[]$	perform a
$[]$	a	perform b
$[]$	$a \cdot b$	print $\vec{f}_s \cdot \vec{f}_l$
$b \cdot a$	$[]$	print $\vec{f}_s \cdot \vec{f}_l$

If the memory model allows this execution, then so (the session order) is not consistent with ar (the arbitration order), i.e. $so \not\subset ar$.

Furthermore, if another client sees $b \cdot a$, then so is not consistent with vis (the visibility order), i.e. $so \not\subset vis$.

Things you don't want (2)

Here's a sample execution.

	client 1		*client* 2	
$\vec{f}_s$	$\vec{f}_l$	e	$\vec{f}_l$	e
$[]$	$[]$	perform a	$[]$	$()$
$[]$	a	$()$	$[]$	$()$
a	$[]$	$()$	$[]$	$()$
a	$[]$	$()$	$[]$	perform b
a	$[]$	$()$	b	print $\vec{f}_s \cdot \vec{f}_l$
$b \cdot a$	$[]$	$()$	$[]$	print $\vec{f}_s \cdot \vec{f}_l$

If the memory model allows this execution, then vis (the visibility order) is not consistent with ar (the arbitration order), i.e. $vis \not\subset ar$.

A better memory model: not for the implementor

This is what we observe in processors; what the user thinks about.

A better memory model: not for the implementor

This is what we observe in processors; what the user thinks about.

We don't know how Intel engineers implement it in silicon. This doesn't explain how to implement it in a networked context. The model doesn't convey the fact that some updates are in transit.

A better memory model: not for the implementor

This is what we observe in processors; what the user thinks about.

We don't know how Intel engineers implement it in silicon. This doesn't explain how to implement it in a networked context. The model doesn't convey the fact that some updates are in transit.

A new model for 1) accurately reflecting the reality of a networked setting and 2) providing detailed implementation guidelines at a reasonable level of detail while 3) remaining understandable by the user.

GSP: the Global Sequence Protocol

(a.k.a. “TSO for networks”)

GSP: the Global Sequence Protocol

(a.k.a. “TSO for networks”)

1. the model

GSP: the Global Sequence Protocol

(a.k.a. "TSO for networks")

1. the model
2. comparison with TSO

GSP: the Global Sequence Protocol

(a.k.a. "TSO for networks")

1. the model
2. comparison with TSO
3. implementation

Yet another operational model

As before, the system is $\langle S, C \rangle$ where

- $S = \vec{f}_s : \text{list } (\sigma \to \sigma)$
  (“the server keeps a list of updates”)
- $C(i) : \text{list } (\sigma \to \sigma) \times \text{list } (\sigma \to \sigma) \times \text{list } (\sigma \to \sigma) \times \text{expr}$
  (“the client keeps… a bunch of stuff”)

Yet another operational model

As before, the system is $\langle S, C \rangle$ where

- $S = \vec{f}_s : \text{list } (\sigma \to \sigma)$
  ("the server keeps a list of updates")
- $C(i) : \text{list } (\sigma \to \sigma) \times \text{list } (\sigma \to \sigma) \times \text{list } (\sigma \to \sigma) \times \text{expr}$
  ("the client keeps... a bunch of stuff")

$C(i) = (\vec{f}_c, \vec{f}_i, \vec{f}_p, e)$ where:

- $\vec{f}_c$ is the list of confirmed updates
- $\vec{f}_i$ is the list of in-flight updates
- $\vec{f}_p$ is the list of pending updates

Yet another operational model

As before, the system is $\langle S, C\rangle$ where

- $S = \vec{f}_s$: list $(\sigma \to \sigma)$
  (“the server keeps a list of updates”)
- $C(i)$: list $(\sigma \to \sigma) \times$ list $(\sigma \to \sigma) \times$ list $(\sigma \to \sigma) \times$ expr
  (“the client keeps… a bunch of stuff”)

$C(i) = (\vec{f}_c, \vec{f}_i, \vec{f}_p, e)$ where:

- $\vec{f}_c$ is the list of confirmed updates
- $\vec{f}_i$ is the list of in-flight updates
- $\vec{f}_p$ is the list of pending updates

$\vec{f}_i$ is important to account for behaviors observed within a networked setting.

Important system transitions

All the standard λ-calculus reduction rules

+

$$\begin{array}{lcll}
\langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, \mathcal{C}[\text{perform } f] \rangle\rangle & \rightsquigarrow & \langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i \cdot f, \vec{f}_p \cdot f, \mathcal{C}[()] \rangle\rangle & \text{Update} \\
\langle \vec{f}_s, \langle \vec{f}_c, f \cdot \vec{f}_i, \vec{f}_p, \mathcal{C}[e] \rangle\rangle & \rightsquigarrow & \langle \vec{f}_s \cdot f, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, \mathcal{C}[e] \rangle\rangle & \text{Process} \\
\langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, f \cdot \vec{f}_p, \mathcal{C}[e] \rangle\rangle & \rightsquigarrow & \langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c \cdot f, \vec{f}_i, \vec{f}_p, \mathcal{C}[e] \rangle\rangle & \text{Echo} \\
\langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, \mathcal{C}[e] \rangle\rangle & \rightsquigarrow & \langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c \cdot f, \vec{f}_i, \vec{f}_p, \mathcal{C}[e] \rangle\rangle & \text{Echo-Other} \\
 & & & (f \notin \vec{f}_p) \\
\langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, \mathcal{C}[\text{get } ()] \rangle\rangle & \rightsquigarrow & \langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, \mathcal{C}[\text{fold}(s_0, \vec{f}_c \cdot \vec{f}_p)] \rangle\rangle & \text{Read}
\end{array}$$

Invariants:

- $\vec{f}_c$ is a prefix of $\vec{f}_s$
- $\vec{f}_i$ is a suffix of $\vec{f}_p$

(apologies)

High-level points about GSP

We are at a lower-level than the previous model.

- We model local, cached knowledge of the state ($\vec{f}_c$).
- We model network transitions and acknowledgement (allows for retries)
- This provides much more precise guidelines for implementing.

With a correct implementation of GSP:

- eventually, $\vec{f}_i$ and $\vec{f}_p$ are empty, and $\vec{f}_c$ is the same for all clients (program that terminates);
- every update eventually makes it to all other clients; every redex eventually reduces (infinite executions, e.g. web services).

GSP vs. TSO (1)

GSP is weaker than TSO, i.e. allows more executions.

Worded differently, any TSO execution is admissible on GSP.

How?

- when the server processes an update, dispatch it to all clients (Process followed by all Echo-* rules)
- therefore, $\forall i, \vec{f}_c(i) = \vec{f}_s$ (remove $\vec{f}_c$)
- therefore, $\forall i, \vec{f}_i(i) = \vec{f}_p(i)$ (remove $\vec{f}_p$)
- then: get the previous model, i.e. TSO

GSP vs. TSO (2)

The difference lies within the relative ordering of operations.

We take $\sigma = \text{list int}, s_0 = []$.

```
perform (fun s -> me :: s);
print (me ^ ”got” ^ get ())
```

If one can observe traces, then here's a trace $\in$ GSP \TSO:

```
1 got [1; 2]
2 got [2]
```

GSP vs. TSO (3)

Here's the GSP execution.

Server	Client 1				Client 2			
$\vec{f}_s$	$\langle\vec{f}_c$	$\vec{f}_i$	$\vec{f}_p$	$e\rangle$	$\langle\vec{f}_c$	$\vec{f}_i$	$\vec{f}_p$	$e\rangle$
$[]$	$\langle[],$	$[],$	$[],$	perform $\ldots\rangle$	$\langle[],$	$[],$	$[],$	$()\rangle$
$[]$	$\langle[],$	$[1],$	$[1],$	$()\rangle$	$\langle[],$	$[],$	$[],$	$()\rangle$
$[1]$	$\langle[],$	$[],$	$[1],$	$()\rangle$	$\langle[],$	$[],$	$[],$	$()\rangle$
$[1]$	$\langle[1],$	$[],$	$[],$	$()\rangle$	$\langle[],$	$[],$	$[],$	$()\rangle$
$[1]$	$\langle[1],$	$[],$	$[],$	$()\rangle$	$\langle[],$	$[],$	$[],$	perform $\ldots\rangle$
$[1]$	$\langle[1],$	$[],$	$[],$	$()\rangle$	$\langle[],$	$[2],$	$[2],$	$()\rangle$
$[1;2]$	$\langle[1],$	$[],$	$[],$	$()\rangle$	$\langle[],$	$[],$	$[2],$	$()\rangle$
$[1;2]$	$\langle[1;2],$	$[],$	$[],$	$()\rangle$	$\langle[],$	$[],$	$[2],$	$()\rangle$
$[1;2]$	$\langle[1;2],$	$[],$	$[],$	print$\rangle$	$\langle[],$	$[],$	$[2],$	$()\rangle$
$[1;2]$	$\langle[1;2],$	$[],$	$[],$	$()\rangle$	$\langle[],$	$[],$	$[2],$	print$\rangle$

GSP vs. TSO (4)

Here’s the TSO execution.

Server	Client 1		Client 2	
$\vec{f}_s$	$\langle \vec{f}_l,$	$e \rangle$	$\langle \vec{f}_l,$	$e \rangle$
$[]$	$\langle [],$	$() \rangle$	$\langle [],$	$() \rangle$
$[]$	$\langle [],$	perform $\ldots \rangle$	$\langle [],$	$() \rangle$
$[]$	$\langle [1],$	$() \rangle$	$\langle [],$	$() \rangle$
$[1]$	$\langle [],$	$() \rangle$	$\langle [],$	$() \rangle$
$[1]$	$\langle [],$	$() \rangle$	$\langle [],$	perform $\ldots \rangle$
$[1]$	$\langle [],$	$() \rangle$	$\langle [2],$	$() \rangle$
$[1; 2]$	$\langle [],$	$() \rangle$	$\langle [],$	$() \rangle$
$[1; 2]$	$\langle [],$	print$\rangle$	$\langle [],$	$() \rangle$
$[1; 2]$	$\langle [],$	$() \rangle$	$\langle [],$	print$\rangle$

With TSO, once an update makes it to the server, it becomes visible to all the clients.

GSP vs. TSO (5)

If one cannot observe the ordering in traces, but only the set of traced events, then GSP and TSO are equivalent.

Intuition: one can always reorder a GSP trace so that it also could've happened under TSO (complicated proof by Sebastian).

For instance, in the previous example...

Implementation concerns

This is all very high-level, abstract and nice. But you don't send functions over the network. (Security, practicality.)

Usually, client and server link the same library. You send a code pointer; i.e. a data type.

With specialization, comes optimizations: if both the server and client are aware of the type of data, they can compress it.

A specialized operational model

Still GSP, but now u is our type of updates.

New typing rules:

- S : list u
- $C(i)$: list $u \times$ list $u \times$ list $u \times$ expr

The client and server agree on a interpretation function ff : list $u \to \sigma$ and a compression function k : list $u \to$ list u. Now:

- a prefix of the state has type σ (has been evaluated)
- a segment of the state has type list u (has been compressed)

Implementing it (1)

The naïve implementation.

```
let $\vec{u}_c$ = ref []
let $\vec{u}_p$ = ref []

let perform f =
  $\vec{u}_p$ := !$\vec{u}_p$ @ [f];
  send f

let get () =
  ff (!$\vec{u}_c$ @ !$\vec{u}_p$)

let _ =
  on_receive (fun { client_id; u } ->
    if client_id = me then begin
      assert (List.hd !$\vec{u}_p$ = u);
      $\vec{u}_p$ := List.tl !$\vec{u}_p$
    end;
    $\vec{u}_c$ := !$\vec{u}_c$ @ [u]
  )
```

Implementing it (2)

Several problems with this implementation:

- no support for atomicity
- confusing programming model (when are updates pulled in?)
- more operations needed (check confirmation)

Implementing it (3)

We can make GSP transactional by batching updates in rounds for atomicity and efficiency. We use an outgoing buffer and a new `push` operation.

We can simplify the programming model by using an incoming buffer and a new `pull` operation. Well-suited for evented / reactive applications.

Implementing it (3)

We pick $\sigma = \text{list } u$.

```
let in_buffer = ref []
let out_buffer = ref []

let perform u =
  out_buffer := !out_buffer @ [u]

let push () =
  let u = !out_buffer in
  $\vec{u}_p$ := !$\vec{u}_p$ @ [u];
  out_buffer := [];
  send u

let get () =
  ff (List.flatten (!$\vec{u}_c$ @ !$\vec{u}_p$))
```

Implementing it (3)

```
let _ =
  on_receive (fun { client_id; u } ->
    in_buffer := !in_buffer @ u
  )

let pull () =
  (* pop from $\vec{u}_p$ if needed *)
  $\vec{u}_c$ := !$\vec{u}_c$ @ !in_buffer;
  in_buffer := []
```

Implementing it (4)

Synchronization primitives?

```
let flush () =
  while (\vec{u}_p <> [])
    (* call network code to receive / send *)
```

`flush` guarantees our local vision is a prefix of the server's (i.e. $\vec{f}_p$ is empty).

Then, one can use "`perform; flush`" or "`flush; get`". It's as if these operations were performed on the server.

Equivalent of fences.

Implementing it (5)

We can improve performance by:

- making the server keep track of “how much” each client knows;
- evaluating the update log (via *ff*) up to the minimum round number;
- compressing rounds before sending them off.

A disconnected client can either ask for a resumption from its last known round and get a diff, or get a complete state if the server has compressed already.

S. Burckhardt, D. Leijen, J. Protzenko and M. Fähndrich
Global Sequence Protocol: A Robust Abstraction for Replicated Shared State.
ECOOP 2015

Implementing it (6)

Remember that σ does not model the entire state of the server.

Rather, σ is the specifically shared data structure (a log, a key-value store, etc.).

Some examples for σ

- $\sigma =$ ref int (shared counter)
- $\sigma =$ list σ (shared log)
- $\sigma =$ hash map...

The notion of a data race depends on σ and the operations we perform over it: a shared counter, or an append-only log have no conflicts. The ordering of updates is the conflict resolution procedure.

A word about conflict resolution

Sometimes you do need to handle conflict resolution. What is a race?

We assume that the type σ can handle conflict resolution in its data representation.

Some tricks:

- consider that types always have a default value (no if-empty-then)
- agree on a merge function.

A word about compare-and-swap

The type σ could possibly support an update *u* of the compare-and-swap variety.

Then, one would have to call `flush` then read the state to figure out whether the operation was successful.

Distributed memory models

(That's a conclusion.)

A good mental model is a series of updates. Functional, core, atomic.

Depending on your setting, use a more or less sophisticated model.

The theory of eventual consistency allows one to precisely state the properties of a memory model.

Implementing your model requires a greater level of detail and the addition of programmer-friendly primitives.