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 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:

\[
e ::= \text{your-typical-\(\lambda\)-calculus}
\]

\[
\begin{align*}
\text{perform } e \\
\text{get } ()
\end{align*}
\]

Quick typing rules (\( \sigma \) is the type of state):

- \( S = \vec{f}_s : \text{list } (\sigma \rightarrow \sigma) \)
- \( C(i) = e : \text{expr} \)
- \( \text{perform } : (\sigma \rightarrow \sigma) \rightarrow \text{unit} \)
- \( \text{get } : \text{unit} \rightarrow \sigma \)
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 $C$ and a given client:

$$
\langle \vec{f}_s; C[\text{perform } f] \rangle \rightsquigarrow \langle \vec{f}_s \cdot f; C[(()] \rangle \\
\langle \vec{f}_s; C[\text{get } ()] \rangle \rightsquigarrow \langle \vec{f}_s; C[\text{fold}(s_0, \vec{f}_s)] \rangle
$$
Initially, $S = []$ and we assume $s_0 : \sigma$ is the initial (empty) state.

How does the system reduce? For a context $C$ and a given client:

\[
\langle \vec{f}_s; C[\text{perform } f] \rangle \rightsquigarrow \langle \vec{f}_s \cdot f; C[()] \rangle \\
\langle \vec{f}_s; C[\text{get } ()] \rangle \rightsquigarrow \langle \vec{f}_s; C[\text{fold}(s_0, \vec{f}_s)] \rangle
\]

- **perform $f$** means: push a *functional* update
- **get ()** means: *compose* all updates to obtain the current state.
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.
Let’s give up consistency for performance.
(a.k.a. let’s put more stuff in-between ⟨...⟩)

A most natural idea: local buffers of updates to improve performance.
(still not saying what σ is)

New operational model: ⟨S, C⟩

- \( S = \vec{f}_S : \text{list} (\sigma \rightarrow \sigma) \) (“the server keeps a list of updates”)
- \( C(i) = (\vec{f}_l, e) : \text{list} (\sigma \rightarrow \sigma) \times \text{expr} \) (“the client keeps a local buffer of updates”)

J. Protzenko et al. — JFLA’16
A better memory model (2)

Two updated transitions and a new one:

\[
\langle \vec{f}_s; \langle \vec{f}_l, C[perform \ f]\rangle \rangle \leadsto \langle \vec{f}_s; \langle \vec{f}_l \cdot f; C[()]\rangle \rangle
\]
\[
\langle \vec{f}_s; \langle \vec{f}_l; C[get ()]\rangle \rangle \leadsto \langle \vec{f}_s; \langle \vec{f}_l; C[fold(s_0, \vec{f}_s \cdot \vec{f}_l)]\rangle \rangle
\]
\[
\langle \vec{f}_s; \langle f \cdot \vec{f}_l; C[e]\rangle \rangle \leadsto \langle \vec{f}_s \cdot f; \langle \vec{f}_l; C[e]\rangle \rangle
\]

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 $\sigma$ 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 \((a, b) \in \text{vis}\), 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.

\[
\vec{f}_s \quad \vec{f}_l \quad e
\]

\[
\begin{array}{c}
\quad \quad \quad \quad \quad \quad \text{perform } a \\
\quad \quad \quad \quad \quad \quad \text{perform } b \\
\quad \quad \quad \quad \quad \quad \text{print } \vec{f}_s \cdot \vec{f}_l \\
\quad \quad \quad \quad \quad \quad \text{print } \vec{f}_s \cdot \vec{f}_l
\end{array}
\]

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.

<table>
<thead>
<tr>
<th></th>
<th>client 1</th>
<th></th>
<th>client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \vec{f}_s )</td>
<td>( \vec{f}_l )</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>perform a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a ()</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>()</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>()</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>()</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>()</td>
</tr>
<tr>
<td>( b \cdot a )</td>
<td></td>
<td></td>
<td>()</td>
</tr>
</tbody>
</table>

If the memory model allows this execution, then \( \text{vis} \) (the visibility order) is not consistent with \( \text{ar} \) (the arbitration order), i.e. \( \text{vis} \nsubseteq \text{ar} \).
This is what we observe in processors; what the user thinks about.
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

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

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

- \( S = \vec{f}_S : \text{list } (\sigma \rightarrow \sigma) \)
  (“the server keeps a list of updates”)

- \( C(i) : \text{list } (\sigma \rightarrow \sigma) \times \text{list } (\sigma \rightarrow \sigma) \times \text{list } (\sigma \rightarrow \sigma) \times \text{expr} \)
  (“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

- \( S = \vec{f}_s : \text{list} \ (\sigma \rightarrow \sigma) \)  
  ("the server keeps a list of updates")
- \( C(i) : \text{list} \ (\sigma \rightarrow \sigma) \times \text{list} \ (\sigma \rightarrow \sigma) \times \text{list} \ (\sigma \rightarrow \sigma) \times \text{expr} \)  
  ("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

\( \vec{f}_i \) is important to account for behaviors observed within a networked setting.
Important system transitions

All the standard $\lambda$-calculus reduction rules

\[
\langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, C[\text{perform } f] \rangle \rangle \leadsto \langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, f, C[()] \rangle \rangle
\]

Update

\[
\langle \vec{f}_s, \langle \vec{f}_c, f \cdot \vec{f}_i, \vec{f}_p, C[e] \rangle \rangle \leadsto \langle \vec{f}_s, \vec{f} \cdot \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, C[e] \rangle \rangle
\]

Process

\[
\langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, f \cdot \vec{f}_p, C[e] \rangle \rangle \leadsto \langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c \cdot f, \vec{f}_i, \vec{f}_p, C[e] \rangle \rangle
\]

Echo

\[
\langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, C[e] \rangle \rangle \leadsto \langle \vec{f}_c \cdot f \cdot \vec{f}_s, \langle \vec{f}_c \cdot f, \vec{f}_i, \vec{f}_p, C[e] \rangle \rangle
\]

Echo-Other

\[
\langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, C[get()] \rangle \rangle \leadsto \langle \vec{f}_s, \langle \vec{f}_c, \vec{f}_i, \vec{f}_p, C[\text{fold}(s_0, \vec{f}_c \cdot \vec{f}_p)] \rangle \rangle
\]

Read

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, \( \forall i, \tilde{f}_c(i) = \tilde{f}_s \) (remove \( \tilde{f}_c \))
- therefore, \( \forall i, \tilde{f}_i(i) = \tilde{f}_p(i) \) (remove \( \tilde{f}_p \))
- then: get the previous model, i.e. TSO
The difference lies within the relative ordering of operations.

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

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

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

1 got [1; 2]
2 got [2]
Here’s the GSP execution.

<table>
<thead>
<tr>
<th>Server</th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vec{f}_s )</td>
<td>( \langle \vec{f}_c \rangle )</td>
<td>( \langle \vec{f}_c \rangle )</td>
</tr>
<tr>
<td>( [] )</td>
<td>( [1], [1], () )</td>
<td>( [1], [1], () )</td>
</tr>
<tr>
<td>( [1] )</td>
<td>( [1], [1], () )</td>
<td>( [1], [1], () )</td>
</tr>
<tr>
<td>( [1] )</td>
<td>( [1], [1], () )</td>
<td>( [1], [1], () )</td>
</tr>
<tr>
<td>( [1] )</td>
<td>( [1], [1], () )</td>
<td>( [1], [1], () )</td>
</tr>
<tr>
<td>( [1; 2] )</td>
<td>( [1], [1], () )</td>
<td>( [1; 2], [2], () )</td>
</tr>
<tr>
<td>( [1; 2] )</td>
<td>( [1; 2], [1], () )</td>
<td>( [1; 2], [2], () )</td>
</tr>
<tr>
<td>( [1; 2] )</td>
<td>( [1; 2], [1], () )</td>
<td>( [1; 2], [2], () )</td>
</tr>
<tr>
<td>( [1; 2] )</td>
<td>( [1; 2], [1], () )</td>
<td>( [1; 2], [2], () )</td>
</tr>
</tbody>
</table>

**perform** ... **print**
**GSP vs. TSO (4)**

Here's the TSO execution.

<table>
<thead>
<tr>
<th>Server</th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\vec{f}_s)</td>
<td>(\langle \vec{f}_l, e \rangle)</td>
<td>(\langle \vec{f}_l, e \rangle)</td>
</tr>
<tr>
<td>([])</td>
<td>(\langle [], () \rangle)</td>
<td>(\langle [], () \rangle)</td>
</tr>
<tr>
<td>(\langle [], \text{perform} \ldots \rangle)</td>
<td>(\langle [], () \rangle)</td>
<td>(\langle [], () \rangle)</td>
</tr>
<tr>
<td>([1])</td>
<td>(\langle [], () \rangle)</td>
<td>(\langle [], () \rangle)</td>
</tr>
<tr>
<td>([1])</td>
<td>(\langle [], () \rangle)</td>
<td>(\langle [], \text{perform} \ldots \rangle)</td>
</tr>
<tr>
<td>([1])</td>
<td>(\langle [], () \rangle)</td>
<td>(\langle [], () \rangle)</td>
</tr>
<tr>
<td>([1; 2])</td>
<td>(\langle [], () \rangle)</td>
<td>(\langle [], () \rangle)</td>
</tr>
<tr>
<td>([1; 2])</td>
<td>(\langle [], \text{print} \rangle)</td>
<td>(\langle [], () \rangle)</td>
</tr>
<tr>
<td>([1; 2])</td>
<td>(\langle [], () \rangle)</td>
<td>(\langle [], \text{print} \rangle)</td>
</tr>
</tbody>
</table>

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

- \( S : \text{list } u \)
- \( C(i) : \text{list } u \times \text{list } u \times \text{list } u \times \text{expr} \)

The client and server agree on a interpretation function \( \text{ff} : \text{list } u \rightarrow \sigma \) and a compression function \( \text{k} : \text{list } u \rightarrow \text{list } u \).

Now:

- a prefix of the state has type \( \sigma \) (has been evaluated)
- a segment of the state has type \( \text{list } u \) (has been compressed)
Implementing it (1)

The naïve implementation.

``` OCaml
let uc = ref []
let up = ref []

let perform f =
    up := !up @ [f];
    send f

let get () =
    ff (!uc @ !up)

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

let get () =
  ff (List.flatten (!$\tilde{u}_c @ !\tilde{u}_p$))
```
Implementing it (3)

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

let pull () =
  (* pop from \( \tilde{u}_p \) if needed *)
  \( \tilde{u}_c \) := !\( \tilde{u}_c \) @ !in_buffer;
  in_buffer := []
```
Implementing it (4)

Synchronization primitives?

```plaintext
let flush () =
  while (\uvec{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](http://example.com)  
Global Sequence Protocol: A Robust Abstraction for Replicated Shared State.  
*ECOOP 2015*
Implementing it (6)

Remember that $\sigma$ does not model the entire state of the server.

Rather, $\sigma$ is the specifically shared data structure (a log, a key-value store, etc.).
Some examples for $\sigma$

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

The notion of a data race depends on $\sigma$ 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 $\sigma$ 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 $\sigma$ 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.
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.