

Appendix to Partition Consistency: Additional Implementations and Proofs

Steven Cheng, Lisa Higham, and Jalal Kawash

University of Calgary, Alberta, Canada

A Implementing Partition Consistency on the Partially Ordered Broadcast Model

SWFR_{POB(L(K))}^{PC(L)} Implementation
Code for each $p \in P$

1. Transformation's local target variables
Memory $[\hat{p}].x \quad \forall x \in V$, local replica variable, initial value is the initial value of x
writes-processed counts messages **bcasted** by \hat{p} that are also **delivered** by \hat{p} , **init**:0
writes-requested counts locally **bcasted** messages, initially 0

2. Transforming specification threads
Transformation of thread p to $\hat{p}.m$:

SWFR _{POB(L(K))} ^{PC(L)} (READ _{p} (x)) return <i>Memory</i> $[\hat{p}].x$	SWFR _{POB(L(K))} ^{PC(L)} (WRITE _{p} (x, v)) <i>writes-requested</i> \leftarrow <i>writes-requested</i> + 1 if $\exists V_i \in K : x \in V_i$ then bcast ($[x, v, \hat{p}], i$) else bcast ($[x, v, \hat{p}], \perp$) WaitWritesComplete()
WaitWritesComplete() wait until <i>writes-processed</i> \geq <i>writes-requested</i>	WaitWritesComplete ()

3. New target threads
One new thread $\hat{p}.d$, $\forall p \in P$:

$\hat{p}.d$: 1 while <i>TRUE</i> 2 do ApplyWrite()	ApplyWrite() <i>update, l</i> \leftarrow deliver () let $[x, v, source] = update$ <i>Memory</i> $[\hat{p}].x \leftarrow v$ if $source = \hat{p}$ then <i>writes-processed</i> \leftarrow <i>writes-processed</i> + 1
---	---

Fig. 1. Implementing Partition Consistency on Partially Ordered Broadcast Model

Partition Consistency models processes that interact by reading and writing globally shared variables that have only weak consistency guarantees. Specifically, PC(K) enforces agreement between processes on the order of writes to all variables within the same set in K . Our task is to transform each such specified process p to a target process \hat{p} for the Partially Ordered Broadcast model, where

inter-process communication is via the partially ordered broadcast primitive. We achieve this in Figure 1 by 1) creating a label for each set in the partition K , that is $L(K) = \{i : V_i \in K\}$; 2) mapping each p to a thread $\widehat{p}.m$ in the $\text{POB}(L(K))$ model; and 3) adding to each \widehat{p} a delivery thread $\widehat{p}.d$. This implementation of Partition Consistency using partially ordered broadcast generalizes the way that totally ordered broadcast is used to implement SC [15, 16].

The main thread, $\widehat{p}.m$, is identical to p except that each READ and WRITE to a shared variable is replaced by a subroutine call. The transformation of a READ returns the value stored in \widehat{p} 's local memory. The transformation of a WRITE creates a bcast to be delivered to each target process. It has a label corresponding to the set in K , or \perp , containing the variable being written to.

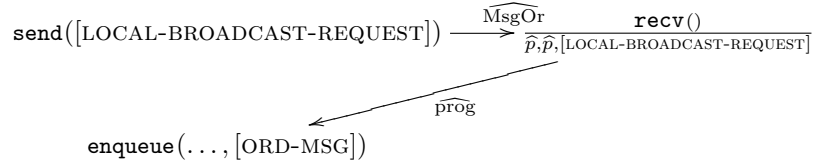
The delivery thread, $\widehat{p}.d$, manages the deliver operations and maintains synchronization with $\widehat{p}.m$ via locally shared variables. It indefinitely checks for updates and applies them to the local memory it shares with $\widehat{p}.m$. The procedure $\text{WaitWritesComplete}$ causes $\widehat{p}.m$ to wait until $\widehat{p}.d$ has applied all the WRITES previously broadcast by $\widehat{p}.m$.

Under the $\text{SWFR}_{\text{POB}}^{\text{PC}}$ transformation, each process has at most one outstanding local write, since every write must be applied locally before the subroutine completes. Every write contains a wait, making these writes “slow”. An alternative is to move the $\text{WaitWritesComplete}$ call from the end of the WRITE to the beginning of the READ . This gives us a fast-write/slow-read (FWSRtrans) implementation described in the full version [6].

B Proofs of remaining lemmas for the Timestamp implementation

Lemma 1. $\text{BCAST}(m, l) \in O$ if and only if $\frac{\text{DELIVER}()}{m, l} \in O|p$.

Proof. Each transformation of a $\text{BCAST}(m, l)$ by process p generates a unique $\text{LOCAL-BROADCAST-REQUEST}$ by \widehat{p} , the transformation of p . Each $\text{LOCAL-BROADCAST-REQUEST}$ results in an enqueue of the corresponding ORD-MSG at the same \widehat{p} :



Each $\text{LOCAL-BROADCAST-REQUEST}$ also results in an enqueue of the corresponding ORD-MSG at every other remote process:

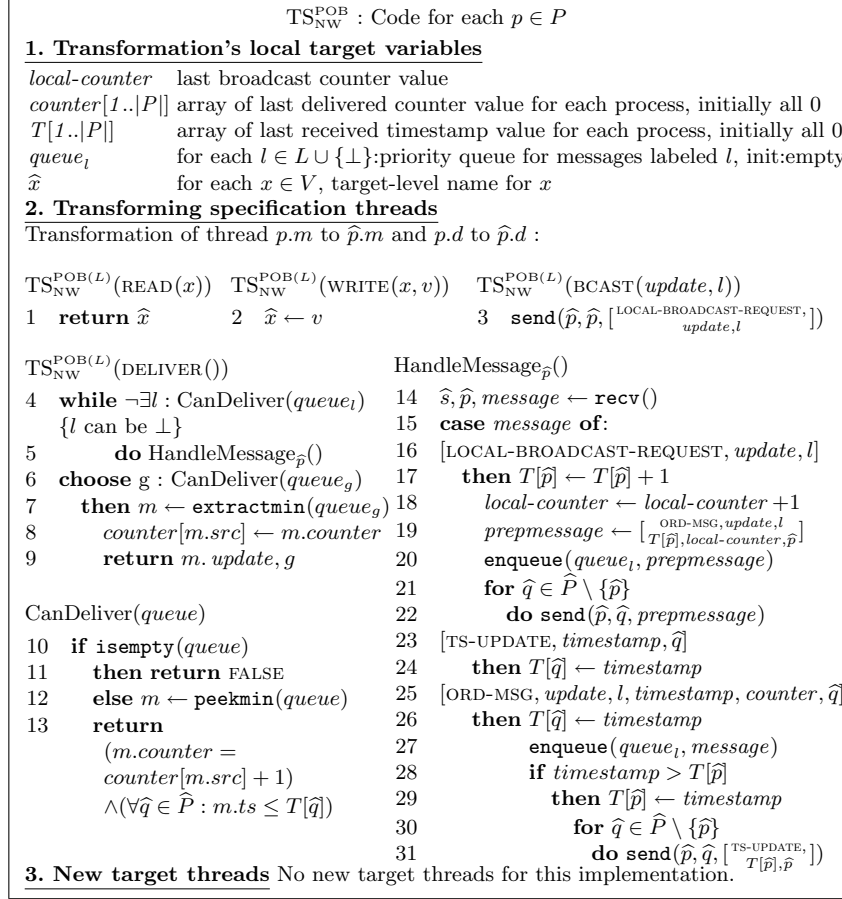
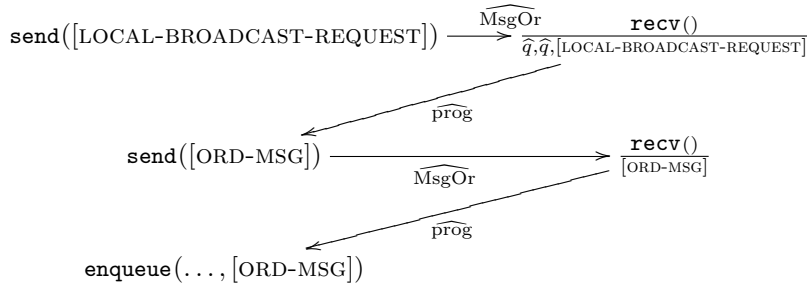


Fig. 2. Timestamp implementation of Partially Ordered Broadcast on Network model



Because these are the only two ways a message is **enqueued** (lines 20 and 27 of HandleMessage), there is a 1-1 correspondence between the set of LOCAL-BROADCAST-REQUESTS and the set of **enqueues** of ORD-MSG for each process. Only **enqueued** messages can be **extractmined** and thus **DELIVERED**. It follows that for every process p , $\frac{DELIVER(l)}{m, l} \in O|p \implies BCAST(m, l) \in O$.

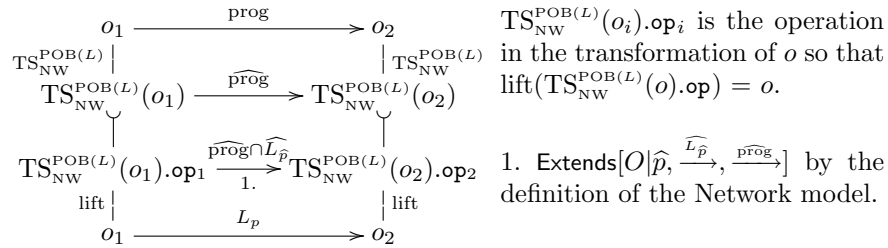
If \widehat{p} enqueues an ORD-MSG, $m_{\text{ORD-MSG}}$ with timestamp ts , at line 20 (the ORD-MSG arose from \widehat{p} 's own LOCAL-BROADCAST-REQUEST), then \widehat{p} next sends a copy of $m_{\text{ORD-MSG}}$ also with timestamp ts to every other process (line 21 in HandleMessage). Each process $\widehat{q} \neq \widehat{p}$ will eventually receive its copy of $m_{\text{ORD-MSG}}$ and handle it beginning at line 25. After enqueueing this message at line 27, if $T[\widehat{q}] < ts$, \widehat{q} updates its $T[\widehat{q}]$ to ts and sends a TS-UPDATE message with timestamp ts to every other process (line 30). Otherwise, \widehat{q} has already sent an ORD-MSG or TS-UPDATE message with timestamp at least ts to every process except itself. Whenever a process \widehat{q} receives an ORD-MSG or TS-UPDATE message with timestamp ts from a process \widehat{r} , \widehat{q} updates its local $T[\widehat{r}]$ to ts (lines 24 and 26). Thus, eventually, for each process $\widehat{q} \in \widehat{P}$, \widehat{q} 's value for $T[\widehat{r}]$ is at least ts . Since processes send messages with strictly increasing timestamps and messages are received in FIFO order, every element of every local T array strictly increases. Therefore, eventually every ORD-MSG will forever satisfy the timestamp part of the CanDeliver requirement. Furthermore, if some priority queue $_i$ contains an ORD-MSG that satisfies the timestamp part of the CanDeliver requirement, then the highest priority ORD-MSG in queue $_i$ does, because within each queue, priority decreases with increasing timestamp.

Consider the set of messages in a process's collection of priority queues, such that each is of highest priority in its queue and satisfies the timestamp part of the CanDeliver requirement. We just established that this set cannot remain empty. Let m be the message with the least (ts, \widehat{source}) pair when it is not empty. Since \widehat{source} sends messages in order of increasing timestamp, channels are FIFO, and each priority queue is ordered by increasing timestamp, every other message from \widehat{source} with timestamp smaller than ts must have been delivered, so $counter[\widehat{source}] + 1$ must be equal to $m.counter$. Therefore, m satisfies CanDeliver.

Provided only a finite number of messages are BCAST (or, in longlived computations, given a weak fairness constraint) every message that satisfies CanDeliver, will be **extractmined** and then **DELIVERED**. Therefore, $\text{BCAST}(m, l) \in O \implies \frac{\text{DELIVER}()}{m, l} \in O|q$ for every process q . \square

Lemma 2. $\forall p \in P : \text{Extends}[O|p, \xrightarrow{L_p}, \xrightarrow{\text{prog}}]$

Proof. Let $o_1, o_2 \in O|p$ such that $o_1 \xrightarrow{\text{prog}} o_2$.



\square