A Solution of Concurrent Queue on Local and Distributed Python STM

Marko Popovic, Branislav Kordic, Miroslav Popovic, and Ilija Basicevic

***Abstract* — This paper was motivated by the two open research challenges in the area of STMs. The first challenge is the development of STM based concurrent queues, whereas the second, maybe even greater, challenge is the development of distributed STMs. Python is assumed as a target language. In this paper, four main contributions are made. First, the concurrent queue data structure on the local Python STM is developed. Second, a distributed STM in Python, called Distributed Python STM, is developed. Third, the developed concurrent queue is ported on the Distributed Python STM. Fourth, the developed concurrent queue is verified using unit and system testing. The developed concurrent queue successfully passed all of the unit and the system tests on both local Python STM and Distributed Python STM.**

***Keywords* — multicore systems; parallel programming; Python; transactional memories; distributed transactional memories; concurrent data structures.**

I. Introduction

HERLIHY and Moss introduced the concept of the *transactional memory* (TM) in 1993 [1]. Until today, TM has undergone many intensive research and development cycles, both in academia and industry. TM has the two main advantages: (i) TM supports abstraction and composition thus it makes parallel programming easier, and (ii) TM based parallel programs may outperform *lock* based programs, because individual transactions are executed *optimistically* (i.e. without locking) [2].

Even though TM paradigm simplifies programming, designing TM software is still challenging. As reported by authors of [3], a significant effort was needed to design and implement a concurrent list based on Python STM (PSTM) [4]. Likewise, concurrent queues pose open research challenges. Some of the design considerations are: (i) how to tailor them to different APIs (e.g. whether or not push and pop need to be supported on both ends of the queue), and (ii) how to handle boundary conditions (e.g. whether or not items can appear duplicated or missed), see [5-7].

In this context, this paper makes its first contribution by presenting the solution for a concurrent queue based on PSTM, called *Concurrent Queue on PSTM* (CQ-PSTM) [8]. CQ-PSTM is based on the unbounded total queue implementation in Java [9]. Regarding mentioned considerations from [5-7], CQ-PSTM is a one-way queue that supports push at its tail and pop at its head, which may be viewed as its advantage, because it's simpler, and its limitation, because it cannot be used as a deque. Handling boundary conditions actually depends on a target application domain, and since CQ-PSTM is mainly targeting real-time systems, duplicate and missed items are not allowed.

CQ-PSTM could not be based on the other two concurrent queue solutions from Chapter 10 in [9], namely the *bounded partial queue* ([9], pp. 225-229) and the *unbounded lock-free queue* ([9], pp. 231-233). The former solution uses condition variables for signaling not empty and not full conditions, whereas the latter solution uses CAS (Compare-And-Swap) primitive, and neither condition signaling nor CAS primitive are supported by TMs in general.

Besides STM based concurrent data structures, this paper also deals with distributed STMs (DSTMs), more general STMs designed for distributed systems. Some DSTMs are based on replication and multi-versioning, whereas other DSTMs use global lock, serialization lease, or commit-time broadcasting, and some DSTMs may also execute transactions speculatively [10]. However, all these DSTMs are targeting C/C++ or Java.

This motivated the authors of this paper to develop the first version of a DSTM in Python, called *Distributed Python STM* (DPSTM), and to the best of their knowledge this is the first DSTM in Python. DPSTM based system is a client-server architecture where DPSTM executes on a server machine, whereas application's transactions and their proxies execute on remote processors (computers, mobiles, IoTs, etc.). Like traditional STMs, DPSTM makes no guaranties in case of failures, and this limitation is one of the main directions of the future work.

Next, CQ-PSTM was ported to DPSTM under the name *Concurrent Queue on DPSTM* (CQ-DPSTM). This port was easy because DPSTM's transactions use a proxy object that supports the PSTM API.

At the time of these writings, the solutions presented in this paper where the only solutions of DSTM and concurrent queues based on STM/DSTM in Python, so it was not possible to compare them with other similar

Paper received April 10, 2019; revised July 11, 2019; accepted July 13, 2019. Date of publication July 31, 2019. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Miroslav Lutovac.

This paper is revised and expanded version of the paper presented at the 26th Telecommunications Forum TELFOR 2018 [8].

This work was supported by the Ministry of Education, Science and Technology Development of Republic of Serbia under Grant III-44009.

Marko Popovic, Branislav Kordic, Miroslav Popovic, and Ilija Basicevic are with the University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Trg Dositeja Obradovica 6, 21000 Novi Sad, Republic of Serbia (e-mail: marko.popovic@rt-rk.uns.ac.rs, branislav.kordic@rt-rk.uns.ac.rs, miroslav.popovic@rt-rk.uns.ac.rs, ilija.basicevic@rt-rk.uns.ac.rs).

systems in Python.

Hence, both CQ-PSTM and CQ-DPSTM were verified using unit and system testing. For unit testing, nine tests were developed cases, whereas for system testing, five application workloads were developed. Both CQ-PSTM and CQ-DPSTM successfully passed all the unit and the system tests.

The rest of the paper is organized as follows. Section II presents the concurrent queue on PSTM (CQ-PSTM), Section III presents DPSTM, Section IV presents the unit and system testing, and Section V presents the final conclusions of this paper.

II. Concurrent Queue on PSTM

The CQ-PSTM is based on the definition of the *unbounded total queue* in [9] (pp. 229, 230).

A. Concept, Data Structures and API Functions

The CQ-PSTM is an unbounded total queue whose nodes are placed into t-vars and linked in a list using t-var identifications (IDs). The CQ-PSTM and its nodes are represented as Python named tuples *QueueOnPSTM* and *Node*, respectively. The elements of the former are: the queue name (*queueName*), the *head* t-var ID (*tvarIdHead*) and the *tail* t-var ID (*tvarIdTail*), whereas the elements of the latter are: the item (*item*) and the t-var ID of the next element (*next*).

The CQ-PSTM uses two types of t-vars: (i) the t-vars that contain the t-var IDs of queue's nodes, and (ii) the t-vars that contain queue's nodes. The *head* and *tail* are the t-vars of the first type, whereas the sentinel and nodes with data items are the t-vars of the second type.

The CQ-PSTM operation is illustrated in Fig. 1. Fig 1a shows the empty queue comprising: *head*, *tail* and *sentinel*, which are placed into the t-vars with the IDs: ID_h, ID_t, and ID_s, respectively. The queue after the first item *a* is enqueued at its tail is shown in Fig 1b: the item *a* is stored in the t-var ID_a, which is linked at the end of the queue.

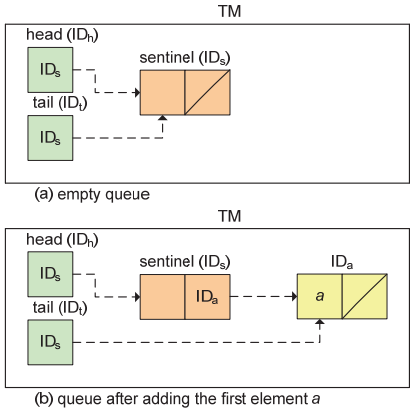

Fig. 1. Queue on PSTM.

The CQ-PSTM API functions are the following: *createQueueOnPSTM*(*q, queueName*), *enq*(*q*, *Q*, *item*), and *deq*(*q*, *Q*).

B. The function createQueueOnPSTM

This function performs the following steps:

1. Create the empty node *sentinel*.
2. Call *createVar* (in PSTM API) to create the new t-var for *sentinel* and store its ID in the variable *tvarIdSentinel*.
3. Call *putVars* (in PSTM API) to set the value of the t-var *tvarIdSentinel* to the node *sentinel*.
4. Create t-var IDs of *head* and *tail* of the queue (*headTvarId* and *tailTvarId*).
5. Call *addVars* (in PSTM API) to add *tvarIdHead* and *tvarIdTail* to PSTM.
6. Call *putVars* to set both *headTvarId* and *tailTvarId* to the value *tvarIdSentinel*, so that *head* and *tail* point to the *sentinel*.
7. Return the tuple representing the new queue (*queueName*, *headTvarId*, *tailTvarId*).

C. The function enq

The function *enq* pseudocode is given in Algorithm 1. The function performs the following steps:

1. Initialize transaction's read, write, and read-write lists (lines 2-4).
2. Create the node for the new item (line 6).
3. Create the t-var for the new node *nodeTvarId*, create the PSTM item for this t-var (*node_item*), and append it to *writeList* (lines 7-10).
4. Get the value (*tail*) and the PSTM item (*tail_item*) for the t-var *tail* (lines 12-13).
5. Create the new PSTM item for *tail* (copy the old ID and version and set the new value to *nodeTvarId*) and append it to *writeList* (lines 14-16).
6. Get the value and the PSTM item for the node at the tail (*tnd* and *tnd_item*), and link it with the new node (lines 18-24).
7. Try to commit the updates as specified by *readWriteList* (lines 26 – 30).

Algorithm 1. The function *enq*

```
01: enq(q, Q, item)
02:  readList ← []              // initialize read list
03:  writeList ← []             // initialize write list
04:  readWriteList ← [readList, writeList]
05:  // Create the new node
06:  node ← Node(item, '')
07:  ret ← createVar(q)
08:  nodeTvarId ← ret[0]
09:  node_item ← (nodeTvarId, 0, node)
10:  writeList.append(node_item)
11:  // Get the tail
12:  tailid ← Q.tvarIdTail
13:  tail, tail_item ← getTvarVerAndValue(q, tailid)
14:  (tid, tver, tval) ←  tail_item
15:  new_tail_item ← (tid, tver, nodeTvarId)
16:  writeList.append(new_tail_item)
17:  // Get the t-var at the tail of the queue
18:  tnd, tnd_item ← getTvarVerAndValue(q, tail)
19:  // Link t-var at the tail and the new node
20:  new_tnd ← Node(tnd.item, nodeTvarId)
21:  delete tnd        // delete old tnd
22:  (tid, tver, tval) = tnd_item
23:  new_tnd_item = (tid, tver, new_tnd)
24:  writeList.append(new_tnd_item)
25:  // Do the Commit
26:  ret ← commitVars(q, readWriteList)
27:  if ret = ['yes']
28:    then return True
29:  removeVars(q, [nodeTvarId])
30:  return False
```

D. The function deq

The function *deq* pseudocode is given in Algorithm 2. The function performs the following steps:

1. Get the value and the PSTM item for *head*, and gets the value and the PSTM item for the node at the head, *hnd* and *hnd_item* (lines 3-6).
2. Check whether the queue is empty, and if yes, return the tuple (False, None) (lines 8-9).
3. Try to get the value and the PSTM item for the node pointed to by *sentinel* (*hnnd* and *hnnd_item*). If the node has meanwhile been deleted, return the tuple (**False**, **None**). Otherwise, store the field *item* of *hnnd* to the variable *result* (lines 11-14).
4. Initialize transaction's read, write, and read-write lists (lines 17-19).
5. Append the *hnd_item* and *hnnd_item* to the *readList* (lines 21-22).
6. Create new item for head and add it to *writeList* (lines 24-26).
7. Try to commit the updates as specified by *readWriteList* (lines 27 – 33).

Algorithm 2. The function *deq*

```
01: deq(q, Q)
02:   // Get the head
03:   hdid ← Q.tvarIdHead
04:   head, head_item ← getTvarVerAndValue(q, hdid)
05:   // Get the t-var at the head of the queue (sentinel)
06:   hnd, hnd_item ← getTvarVerAndValue(q, head)
07:   // Check whether the queue is empty
08:   try if hnd.next = '' then return (False, None)
09:   except return (False, None)
10:   # Get the head next node (node next to sentinel)
11:   hnnd, hnnd_item ←
                        getTvarVerAndValue(q, hnd.next)
12:   // Get the item form the head next node (hnnd)
13:   try result ← hnnd.item
14:   except return (False, None)
15:   // Update the head t-var, to point to hnnd
16:   // Prepare read-write list for commitVars
17:   readList ← []              // initialize read list
18:   writeList ← []             // initialize write list
19:   readWriteList ← [readList, writeList]
20:   // Add items for t-vars that we read to readList
21:   readList.append(hnd_item)
22:   readList.append(hnnd_item)
23:   // Create new item for head and add it to writeList
24:   (hid, hver, hvalue) ← head_item
25:   new_head_item ← (hid, hver, hnd.next)
26:   writeList.append(new_head_item)
27:   // Do the Commit
28:   ret ← commitVars(q, readWriteList)
29:   if ret ≠ ['yes']
30:     then return (False, result)
31:   // Remove tvar containing the old sentinel
32:   removeVars(q, [head])
33:   return (True, result)    // return the item
```

III. Distributed Python STM

This section presents DPSTM. The next subsections describe the DPSTM based system architecture, the DPSTM API, the DPSTM operation, and the DPSTM implementation.

A. DPSTM based system architecture

DPSTM-based system architecture is a client-server type of architecture. Hardware infrastructure of a DPSTM-based system is a TCP/IP network, such as Internet, which is used to connect remote processors, such as computers, mobile devices, IoT (Internet of Things) devices, etc., to a central server computer, see the bottom part of Fig. 2.

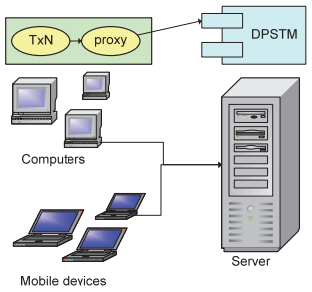

Fig. 2. DPSTM based system architecture.

The remote processors host distributed application processes that act as DPSTM clients, whereas the server computer hosts the DPSTM, see the top part of Fig. 2. An application process executes a transaction (TxN), which in turn uses the proxy to request service from DPSTM. Proxies and DPSTM communicate over an authentication-secured connection.

Like PSTM, DPSTM maintains the system dictionary of t-variables. The dictionary item is a triple (*key*, *ver*, *val*), where *key* is the t-variable's key, *ver* is the current t-variable's version, and the *val* is the current t-variable's value. The triple (*key*, *ver*, *val*) is stored in the dictionary as a key-value pair whose key is *key* and value is the tuple (*ver*, *val*). The DPSTM server serves incoming requests atomically – it receives the request, does the required processing, and sends back the consequent return value. That is the key idea behind this architecture.

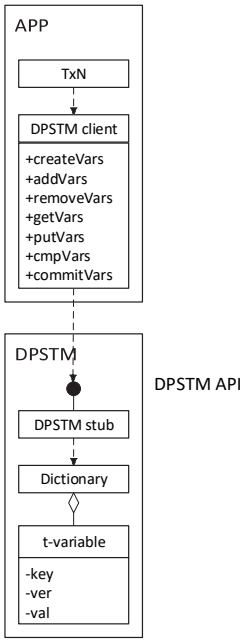

Fig. 3. DPSTM based system class UML diagram.

The simplified UML class diagram of the DPSTM based system architecture is shown in Fig. 3. The DPSTM is a singleton in the system, and normally it is used by more applications (APPs).

Within APP, the DPSTM client (a.k.a. proxy) provides the DPSTM API to a transaction (TxN). The DPSTM API is the set of public functions (*createVars*, *addVars*, etc.) that are used by TxN in order to request services from DPSTM. The DPSTM client's functions delegate their work to the DPSTM stub's functions with the same name.

From APP's point of view, the DPSTM stub provides the

same DPSTM API to the DPSTM client. Although the syntax of both APIs offered by the DPSTM client and the DPSTM stub is the same, their semantics is different. Unlike the DPSTM client's functions which just delegate their work, the DPSTM stub's functions provide the requested services by manipulating the system dictionary. The system dictionary comprises a set of t-variables (a.k.a. dictionary items). A t-variables has the three attributes, namely *key*, *ver*, and *val*.

A distributed DPSTM-based application comprises a set of transactions that operate on their local variables and their copies of t-variables. During their lifecycle, transactions typically acquire their copies of t-variables at the beginning, do some processing, and update (i.e. commit) some of t-variables at the end by making use of the DPSTM API.

B. DPSTM API

One of the important DPSTM design objectives was to keep DPSTM API essentially the same as PSTM API [3], in order to aid easy porting of PSTM-based software to DPSTM. The main difference between these APIs is that PSTM API is module-oriented (specified as a set of module functions), whereas DPSTM API is object-oriented (specified as a set of proxy object's functions). Each PSTM API function has a client-to-server queue object, as its first argument, whereas the corresponding DPSTM API function is called on a proxy object.

More precisely, let *q* be the queue object and *args* other arguments of a PSTM API function *apifun*, and let *p* be the proxy object offering the DPSTM API, then the PSTM API function call:

apifun(*q*, *args*)

corresponds to the DPSTM API function call:

p.apifun(*args*)

The DPSTM API provides the following proxy object's functions:

1. *createVar*()
2. *addVars*(*keys*)
3. *removeVars*(*keys*)
4. *getVars*(*keys*)
5. *putVars*(*vars*)
6. *cmpVars*(*vars*)
7. *commitVars*(*read_write*)

In the list above, *keys* is a list of t-variable's keys (IDs), *vars* is a list of dictionary items, and *read_write* is the list [*read*, *write*] where *read* is the list of dictionary items that were only read and *write* is the list of dictionary items that were (also) written into. For simplicity, we consider terms *dictionary item* and *t-variable* as synonyms, so we use these terms interchangeably.

Functions *createVar* and *removeVars* were added to PSTM API after [3] was published. The former returns the new and unique t-variable ID, whereas the latter deletes t-variables whose keys are in the list *keys*. The rest of the DPSTM API functions operate exactly the same as their PSTM API counterparts, for more details see [3].

C. DPSTM operation

At the beginning, the main application function typically declares and initializes the t-variables, which will be used by transactions, by using the DPSTM API functions *addVars* and *putVars*, respectively, see Fig. 4a. The former function creates the initial dictionary item, whereas the latter set t-variable's initial value.

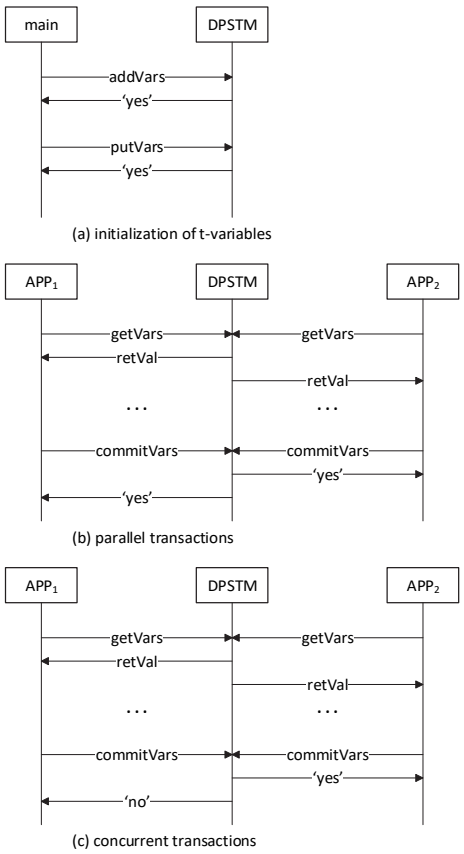

Fig. 4. DPSTM UML sequence diagrams for typical scenarios.

Each transaction on its own, typically starts by getting its copies of t-variables from DPSTM by calling the DPSTM API function *getVars*. A transaction then proceeds by some processing on its local copies of t-variables and perhaps some additional simple local variables. Optionally, it may get more t-variables, create and initialize additional t-variables, etc. At its end, a transaction typically updates t-variables that it modified during its local processing by calling the DPSTM function *commitVars*.

When pairs of transactions are executed simultaneously, they may be executed as *parallel transactions* in case when they do not share t-variables at all, see Fig. 4b, or as *concurrent transactions* in case when they share some of the t-variables, see Fig. 4c. In the former case both transactions get committed, whereas in the latter case one gets committed and the other gets aborted. When more than two transactions are executed simultaneously, some of them may get executed as parallel and some as concurrent transactions.

D. DPSTM implementation

DPSTM implementation is based on the abstraction of a manager provided by Python 3.x multiprocessing package. Generally, *managers* maintain data which can be shared over a network among different processes running on different machines. A manager object controls a server process which manages shared data objects, whereas client

processes can access the shared data objects by using proxies. The server and proxy processes are also called the *remote manager process* and the *local manger process*, respectively.

The optional arguments of the base class *BaseManager*'s constructor are *address* and *authkey*, where *address* is the address given as a pair (IP address, port) on which the manager process listens for new connections, and *authkey* is the authentication key which is used to check the validity of incoming connections to the server process. The main *BaseManager*'s functions (i.e. methods) are the following:

1. *register*, which can be used for registering a new type of shared object or callable with the manager class.
2. *get_server*, which returns a server object, which in turn provides the function *serve_forever* that is typically used to start the remote manager process.
3. *connect*, which connects a local manager object to a remote manager process.

Besides the *basic manager* (represented by the class *BaseManger*), Python 3.x multiprocessing package introduces the notions of customized managers and remote managers. A *customized manager* is created by writing a new subclass of *BaseManager*, which is then registered as a new type of shared object. On the other hand, a *remote manager* is a manager that executes on a remote machine (from a client's point of view), and it is created by calling the *BaseManger* constructor with some particular values for the arguments *address* and *authkey*.

In terms of the notions introduced above, DPSTM is implemented as a *remote customized manager*. Here are the details how this is done. DPSTM is represented by the new class *DpstmManager*, which is a subclass of *BaseManager*. This class maintains the system dictionary of t-variables and the next unique t-variable ID as its private data fields, and it exports the DPSTM API defined in section IV.B.

The two types of Python applications within the DPSTM based system are distinguished, namely the DPSTM server application and the DPSTM client application. The DPSTM server application performs the following steps at its start-up:

1. Register the class *DpstmManager* as a new type *DPSTM* by using the function *register*.
2. Make *DpstmManager*'s instance *m* by providing values for the arguments *address* and *authkey*.
3. Get server object *s* by calling *m.get_server*()
4. Start server process by calling *s.serve_forever*().

The DPSTM client application performs the following steps at its start-up:

1. Register for using the new type *DPSTM*.
2. Make *DpstmManager*'s instance *m* by providing values for the arguments *address* and *authkey*.
3. Connect to the remote manager process by calling *m.connect*()
4. Get the proxy object *dpstm* by calling *DPSTM*'s constructor on the object *m*.

After start-up, a transaction calls desired *DPSTM*'s functions on the object *dpstm*, e.g. to get t-variables it calls *dpstm.getVars*(*keys*).

IV. Unit and System Testing

At the time of these writings there were no standard TM benchmarks for Python publically available. The existing standard benchmarks, such as STAMP and STMBench7, could not be directly used, because they are written for different languages (C++ and Java) and for STMs with different APIs. Therefore, CQ-PSTM and CQ-DPSTM were verified using unit and system testing, which is presented in the following text.

A. CQ-PSTM unit and system testing

CQ-PSTM was verified using unit and system testing. Nine test cases were developed for unit testing, and five application workloads were developed for system testing. CQ-PSTM successfully passed all of these unit and system tests.

For unit testing, the following test cases were used: (1) create one CQ-PSTM queue, (2) create 100 CQ-PSTM queues, (3) enqueue one item into a CQ-PSTM queue, (4) enqueue 100 items into a CQ-PSTM queue and check its length, (5) try to dequeue a non-existing item from the empty CQ-PSTM queue, (6) dequeue an existing item from the CQ-PSTM queue with one item, (7) create a child process and check whether the child process successfully enqueues one item, (8) create a child process and check whether the child process successfully dequeues one item, and (9) enqueue 100 items into CQ-PSTM queue, dequeue the first 50 items, and check whether the second 50 items remain in the CQ-PSTM queue.

For system testing, the following application workloads were used: (1) one enqueuer and one dequeuer (OEOD), which exchange *nItems* items over the queue, (2) *nP* enqueuers (NPE), which enqueue *nItems*/*nP* items each, (3) *nP* dequeuers (NPD), which dequeue *nItems*/*nP* items each, (4) *nE* enqueuers and *nD* dequeuers (NEND), where enqueuers enqueue *nItems*/*nE* items each, and dequeuers dequeue *nItems*/*nD* items each, and (5) *nP* processes (NPP), where each process firstly enqueues *nItems* into the queue and then dequeues *nItems* from the queue. All the workloads start from the empty CQ-PSTM queue, except NPD which starts from the queue with *nItem* items.

Conflicts are analyzed using Bernstein's conditions. Generally, enqueuer's read and write sets are: $R_e = \{\}$ and $W_e = \{tail, N_t, N_n\}$, where N_t is the node at tail and N_n is the new node. Dequeuer's read and write sets are: $R_d = \{sentinel, N_h\}$ and $W_d = \{head\}$, where N_h is the node at head. Therefore, an enqueuer and a dequeuer may conflict on N if $N = N_t = N_h$, two enqueuers may conflict on *tail* and N_t, and two dequeuers may conflict on *head*.

The initial system testing was conducted in [8] by using the PC with Intel Core i7-3770@3.40GHz quad core, with 16GB of shared memory, and OS Linux. The aim was to test the CQ-PSTM at maximal concurrency by executing workload's child processes on separate processor cores. Hence, one of the cores was reserved for the OS and PSTM server, and the number of child processes in the workloads was up to 3 (1, 2, or 3).

In this paper, the same system testing was conducted by using the PC with Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz CPU with 12 dual-threaded cores (i.e. 24 HW

threads), 64 GB of shared memory, and OS Linux. The number of retries in Tab. I in this paper compared to Tab. I in [8] is smaller, because interference from OS on the 12-cores is less than on 4-cores in [8].

TABLE I. OVERVIEW OF CQ-PSTM SYSTEM TESTS

Application Workload		Retries		Test Verdict
		Y_e	Y_d	
OEOD		0	2	Passed
NPE	*nP*=2	49	-	Passed
	nP=3	102	-	Passed
NPD	*nP*=2	-	98	Passed
	nP=3	-	166	Passed
NEND	(*nE*,*nD*) = (1,2)	0	117	Passed
	(*nE*,*nD*) = (2,1)	59	4	Passed
NPP	*nP*=2	56	94	Passed
	nP=3	94	120	Passed

The overview of CQ-PSTM system tests is given in Tab. I. The system test parameters are the following:

- *nItems* is the number of items to be transferred over the queue; it is set to 99 (in OEOD) or 120 (otherwise, to make it divisible by the number of child processes);
- *nP* is the number of child processes; it is used in the workloads NPE, NPD, and NPP, and is set to 2 or 3;
- *nE* and *nD* are the number of enqueuers and the number of dequeuers, respectively; they are used in the workload NEND, and are set as a pair (*nE*, *nD*) either to (1, 2) or (2, 1), because up to 3 cores were used by child processes;
- Y_e and Y_d are the average of the total number of retries made by enqueuers and dequeuers, respectively;

B. CQ-DPSTM unit and system testing

CQ-DPSTM was verified using the same unit tests and application workloads with the same parameters that were used for CQ-PSTM. Thus the analysis of possible conflicts is also the same, so it was natural to expect similar results.

The system testing was conducted using two PCs connected over intranet in the Institute RT-RK. The first PC that hosted an application workload was the same PC that we used for CQ-PSTM testing, and the second PC that hosted DSPTM was a quad 2-threaded multicore with 8 GB of memory and OS Linux. This simple test setup was chosen because: (i) it was easy to implement, and (ii) more importantly, partitioning an application workload and running these partitions on more PCs cannot generate a much more difficult service demand for DPSTM, because network software in the server machine serializes all the traffic towards DPSTM anyhow.

TABLE II. OVERVIEW OF CQ-DPSTM SYSTEM TESTS

Application Workload		Retries		Test Verdict
		Y_e	Y_d	
OEOD		0	2	Passed
NPE	*nP*=2	66	-	Passed
	nP=3	123	-	Passed
NPD	*nP*=2	-	94	Passed
	nP=3	-	214	Passed
NEND	(*nE*,*nD*) = (1,2)	0	102	Passed
	(*nE*,*nD*) = (2,1)	63	0	Passed
NPP	*nP*=2	65	99	Passed
	nP=3	123	183	Passed

The overview of CQ-DPSTM system tests is given in Tab. II. All the results are averaged values of 5 repeated runs. As expected, the results in Tab. II are rather similar to the results in Tab. I. The differences between values in Tables I and II may be explained by the variations in the traffic over the intranet.

V. CONCLUSION

The main contributions of this paper are the following: (i) developed the concurrent queue on PSTM (CQ-PSTM), (ii) developed the first version of distributed PSTM (DPSTM), (iii) ported the concurrent queue to DPSTM (CQ-DPSTM), and (iv) verified both CQ-PSTM and CQ-DPSTM using unit and system testing. For unit testing, nine test cases were developed, whereas for system testing, five application workloads were developed. Both CQ-PSTM and CQ-DSPTM successfully passed all of the unit and the system tests.

The main advantage of this paper is that the solutions presented in this paper are the first solutions of DSTM and concurrent queues based on STM/DSTM in Python. At the same time, this is the main limitation of the paper, because at the time of these writings there were no other similar systems in Python to be compared with.

In their future work, the authors of this paper plan to develop the second, more robust, version of DPSTM. They also plan to continue research on other concurrent data structures in this context.

REFERENCES

[1] M. Herlihy and J.E.B. Moss, "Transactional memory: architectural support for lockfree data structures," Proc. of the *20th Annual International Symposium on Computer Architecture*, ACM, New York, NY, pp. 289–300, 1993.

[2] T. Harris, J.R. Larus, and R. Rajwar, *Transactional Memory*, 2nd edition, Morgan and Claypool, 2010.

[3] M. Popovic, B. Kordic, M. Popovic, and I. Basicevic, "A Solution of Concurrent List on PSTM," Proc. *5th IcETRAN*, Article RTI2.1, pp. 1-6, 2018.

[4] M. Popovic, and B. Kordic, "PSTM: Python Software Transactional Memory," Proc. *22nd IEEE Telecommunications Forum (TELFOR)*, pp. 1106-1109, 2014.

[5] N.S. Arora, R.D. Blumofe, and C.G. Plaxton, "Thread scheduling for multi-programmed multiprocessors. Proc. *10th Symposium on Parallel Algorithms and Architectures*, pp. 119–129, 1998.

[6] M.M. Michael and M.L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," Proc. *15th Annual ACM Symposium on Principles of Distributed Computing*, pp. 267–275, 1996.

[7] M.M. Michael, M.T. Vechev, and V.A. Saraswat, "Idempotent work stealing," Proc. *14th Symposium on Principles and Practice of Parallel Programming*, pp. 45–54, 2009.

[8] M. Popovic, B. Kordic, M. Popovic, and I. Basicevic, "A Solution of Concurrent Queue on PSTM," Proc. *26th Telecommunications Forum (TELFOR)*, pp. 735–738, 2018.

[9] M. Herlihy and N. Shavit, *The art of multiprocessor programming*, 2nd edition, Morgan Kaufmann, 2008.

[10] P. Romano, R. Palmeiri, F. Quaglia, and L. Rodrigues, "On speculative replication of transactional systems", *Journal of Computer and System Sciences*, vol. 80, no. 1, pp. 257-276, 2014.

[11] A. Liu, M. Popovic, H. Zhu, "Formalization and Verification of the PSTM Architecture", Proc. *24th Asia-Pacific Software Engineering Conference*, pp 427-435, 2017.

[12] B. Kordic, M. Popovic, S. Ghilezan, I. Basicevic, "An Approach to Formal Verification of Python Software Transactional Memory", Proc. *5th European Conference on the Engineering of Computer Based Systems*, Article No. 13, pp. 1-10, 2017.