[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



# Fault Tolerant Distributed Python Software Transactional Memory

Marko POPOVIC, Ilija BASICEVIC, Miodrag DJUKIC, Miroslav POPOVIC
*University of Novi Sad, Faculty of Technical Sciences, Trg D. Obradovica 6, 21000 Novi Sad, Serbia*
*marko.popovic@rt-rk.uns.ac.rs*

***Abstract*—Much of the previous research has been done on distributed software transactional memories targeting data centers in Internet clouds, which resulted in nondeterministic and nonrealtime middleware solutions mainly written in Java and C++. On the other hand, embedded systems based on the Internet of Things at the edge of the Internet, such as smart homes, cars, etc., need to operate in realtime, and should, therefore, be deterministic. In order to be smart, these systems use machine learning, and nowadays Python is becoming a leading language in this venue, too. This is the first paper that presents a distributed software transactional memory that is at the same time: (i) fault tolerant, (ii) deterministic, (iii) based on Python, and (iv) extended from a formally verified root. The presented solution consists of a pair of master-slave transaction coordinators and a set of replicated data servers and targets small-to-medium edge networks. Besides intelligent embedded systems, based on the Internet of Things, it can be also used in a wide range of application domains, from SCADA systems to large-scale simulations. The experimental results, presented in the paper, show a superlinear growth of the system throughput as the workload changes from write-only to read-only.**

***Index Terms*—Internet of Things, embedded software, distributed computing, parallel programming, fault tolerance.**

## I. INTRODUCTION

The main goal of this research was to develop a distributed transactional memory for smart embedded systems based on the Internet of Things (IoT), such as smart homes, cars, etc., that would be a natural extension of the existing Python language abstractions. In this section, we: (i) briefly overview the previous research in order to highlight the open challenges to be solved, (ii) present the novel contributions made in this paper, (iii) outline some possible real-world applications, and (iv) present our approach to experimental evaluation.

*Transactional memory* (TM) was introduced by Herlihy and Moss [1] as an extended cache coherence protocol (in particular Goodman's "snoopy" protocol for a shared bus [2]), which aids in-memory transactions that are executed speculatively (i.e. lockless) on multicores. *Software TM* (STM) was introduced by Shavit and Touitou as a software implementation of a TM abstraction [3]. Distributed STM (DSTM) was introduced as an extension of software distributed shared-memory system (S-DSM) for clusters of workstations, see [4]. Similar Java DSTM based on object-level conflict detection and commit-time broadcast of transactions' read/write sets was proposed in [5-6].

Similarly, a dependable DSTM, named D$^2$STM, was proposed in [7], as the final result of an intensive research reported in a series of previous papers [8-11]. D$^2$STM was built on top of JVSTM [8], and it introduced Bloom Filter Certificate (BFC), a commit-time transaction certification procedure that reduces certification overhead at the cost of a user-tuneable increase of the transaction abort probability.

A great variety of techniques used to implement DSTMs include: global lock [4], serialization lease, commit-time broadcasting [7], [12], replication and multi-versioning [4], [13], speculative transaction execution in replicated environments [14], and transaction scheduling using consistent snapshots [4], [13-14]. Generally, DSTMs use one of the three possible execution models: (i) control-flow (objects are immobile and transactions either move or use remote procedure calls) [15], (ii) data-flow (objects move and transactions are fixed to network nodes) [16-20], and (iii) hybrid (combined data-flow and control-flow) [21-22]. However, all these research initiatives mainly target data centers in Internet clouds, and as a result, they are: (i) nondeterministic, thus nonrealtime middleware solutions, (ii) mainly written in Java and C++.

On the other hand, embedded systems based on IoT at the edge of the Internet, such as smart homes, cars, etc. need to operate in realtime and should therefore be deterministic [23-24]. Also, in order to be smart, these systems use machine learning and since Python is becoming a leading language in this venue, they should be based on Python. So, finding a solution that would be (i) deterministic and (ii) Python based were the two main challenges of this paper.

The following line of research addressed these challenges. Python STM (PSTM) was introduced in [25] as the first STM in Python. PSTM was formally verified using three complementary approaches by using: (i) Communication Sequential Processes (CSP) and Process Analysis Toolkit (PAT) [26], (ii) Timed Automata (TA) and UPPAAL [27], and (iii) Push/Pull semantic model [28], respectively. PSTM scheduling architecture, together with the four online transaction scheduling algorithms (Round Robin, Execution Time Load Balancing, Avoid Conflicts, and Advanced Avoid Conflicts) were introduced in [29] as a PSTM contention manager, which was then formally verified in [30].

Distributed PSTM version 1 (DPSTM v1) was introduced in [31] as the first DSTM in Python. DPSTM v1 is a Python custom remote manager that runs on a server, whereas transactions with their proxies are its clients that run on remote processors, such as IoT. Obviously, the main limitations of DPSTM v1 are that it represents both: (i) the performance bottleneck, and (ii) the single point of failure.

DPSTM v2 was introduced in [32] as the next natural step in the DPSTM evolution, which introduced data replication

This work was supported in part by the Serbian Ministry of Education, Science, and Technology Development under grant III 44009-2.

Digital Object Identifier 10.4316/AECE.2020.04003



1582-7445 © 2020 AECE

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



[33] in order to provide good performance, especially for transactions operating in the eventual consistency mode [34]. DPSTM v2 comprises a single Transaction Coordinator (TC) and *n* DPSTM v1 replicas that are called Data Servers (DSs); $DS_1$ is called the *base replica*. The simple and deterministic *replication protocol* resembles the solutions from deterministic databases [35]: upon transaction's request, TC updates all the DSs, always in the order from $DS_1$ to $DS_n$. A transaction operating in the *sequential consistency mode* (SCM) gets (reads) t-variables from TC, whereas a transaction operating in the *eventual consistency mode* (ECM) gets t-variables from its local DS (that may be any $DS_i$). The DPSTM v2 advantages are: (i) good performance for applications with high read-to-update ratio operating in ECM, and (ii) $(n - 1)$ tolerance to DS replica crashes. Obviously, the main limitation of DPSTM v2 is that TC is a single point of failure.

In this paper, we introduce DPSTM v3 as the next natural step in the DPSTM evolution, wherein a single TC is replaced with a pair of TCs operating in the master-slave mode (MSM), in order to provide tolerance to individual TC crashes. Thus DPSTM v3 is a solution that resolves both of the above-mentioned challenges for targeted smart embedded systems based on IoT, which are considered to be small-to-medium edge networks.

### A. Contributions

To the best of our knowledge, DPSTM v3 is the first DSTM that is at the same time: (i) fault tolerant, (ii) deterministic, (iii) based on Python, and (iv) extended from a formally verified root. These four groups of novel (personal) contributions are briefly explained below.

*DPSTM v3 features fault tolerance*: In this paper we propose a novel solution to control a pair of TCs working in master-slave mode by a set of distributed finite state machines (FSMs) that are collocated with transactions' proxies. This solution is ultrarobust, because it can withstand the arbitrary number of individual FSM crashes. We also propose a novel solution for the failover after the master TC crash.

*DPSTM v3 features determinism*: In this paper we propose a novel solution of the adapted deterministic replication protocol (which is based on DPSTM v2). We also propose a novel solution for the deterministic recovery of the broken service after a master TC crash. At this point, we would like to stress the importance of determinism. According to [35], deterministic database solutions have important advantages over nondeterministic solutions, one of the key being that deterministic solutions have better performance especially in case of high contention among transactions.

*DPSTM v3 features Python nativism*: In this paper we propose a novel architectural solution by which TCs and DSs are implemented as Python's remote managers. Therefore, like DPSTM v1 and v2, DPSTM v3 architecture is also a natural extension of the existing Python language abstractions.

*DPSTM v3 features a formally verified root*: In this paper we propose a novel architectural solution by which DPSTM v3 architecture is constructed as an extension of DPSTM v2, which in turn is an extension of DPSTM v1. Both of these extensions were made transparent to transactions, therefore DPSTM v2 and v3 both inherited *serializability* from DPSTM v1 that was formally verified in [36].

### B. Applications

Although DPSTM v3 primarily targets intelligent embedded systems based on IoT, such as smart homes, it can also be used in a wide range of application domains, from Supervisory Control and Data Acquisition (SCADA) systems to large-scale simulations. Here we briefly outline three possible real-world applications.

In a smart home solution, such as the one proposed in [37-38], DPSTM v3 may be used to cache realtime data collected by gateways, e.g. temperature, pressure, etc., and it can replace state of the art components used for this purpose, such as Redis. The main advantages of DPSTM v3 over Redis are that it is deterministic and based on the formally verified DPSTM v1.

In a small to medium SCADA system, DPSTM v3 could be used as an in-memory realtime transactional database. Interestingly enough, the well-known OASyS SCADA also uses master-slave based solution for its realtime database, much like DPSTM v3. However, to the best of our knowledge, the information about whether this solution was formally verified is not publically available.

In large-scale simulations, DPSTM v3 may be used as a cache for simulation data. As a matter of fact, PSTM was already used for this purpose with the very large Python computational-chemistry simulation program, for the protein structure prediction problem, called DEEPSAM [39] on a many-core server. DPSTM v3 may be used with DEEPSAM on a computer cluster in the future.

### C. Experimental Evaluation Approach

As highlighted in the previous subsection I.B, DPSTM v3 may be used in many real-world applications. However, in all these applications, it is invariably used as some kind of a realtime transactional data cache, so its practical task is always the same: to efficiently support transactions, i.e. atomic series of read and write operations. The system throughput (e.g. the number of transactions or operations in a second) is typically used as a measure of efficiency.

Experimental evaluation for this kind of system components is difficult, because: (i) the number of available real-world applications is always limited (when compared with the wide range of possible applications), and (ii) stakeholders of such systems are normally against publishing such data as they consider them to be business secrets.

Therefore, researchers both in academia and industry commonly use the so-called *synthetic workloads* to characterize (i.e. simulate) a wide range of all possible applications. Typically, these workloads are made as different mixtures of read and write operations. For example, the experimental evaluation of Amazon's Zookeeper was made using this method, e.g. see Fig. 5 in [40], which shows the system throughput (measured as the total number of read and write operations in a second) as a function of the proportion of read operations in a workload. Following this de facto standard method, the test application constructed in this paper generates synthetic workloads as mixtures of primitive read and write transactions, and we

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



present our results analogously to [40].

Another difficulty with the experimental evaluation of this kind of distributed systems is that it heavily depends on the underlying network configurations and technology. As there is a wide range of possible applications, there is also a wide range of possible underlying networks, which is hard to cover.

In the preliminary experimental evaluation made in this paper, we used the network technology that was available in our laboratory (a typical modern LAN), and two network configurations (the fully distributed configuration and the configuration with colocation) that from the DPSTM v3 point of view may be viewed as the two opposite extremes in the whole range of possible network configurations. We plan to conduct more experiments on other network configurations and technologies in our future work.

### *D. Roadmap*

The rest of the paper is organized as follows. Section II introduces DPSTM v2 design, Section III presents DPSTM v3 implementation, Section IV presents DPSTM v3 experimental evaluation, and Section V presents the conclusions of this paper.

## II. DPSTM V3 DESIGN

This section presents DPSTM v3 high-level design. The following three subsections present DPSTM v3 based system architecture, DPSTM v3 application programming interface (API), and DPSTM v3 operation, respectively.

### *A. DPSTM v3 Based System Architecture*

DPSTM v3 based system is a client-multi-server architecture, which comprises *m* clients' application programs, $A_1$, …, $A_m$, and DPSTM v3, which in its turn consists of a pair of TCs operating in MSM, $TC_1$, and $TC_2$, and *n* data servers, $DS_1$, …, $DS_n$, see Fig. 1. During normal system operation, one of the TCs is the master TC (MTC), whereas the other is the slave TC (STC). MTC and STC operate as a *hot-standby* pair, which means that MTC is active, whereas STC is just waiting to take over if MTC crashes. In case of a MTC crash, and while it is under repair, the system continues working with a single TC, which operates as a solo MTC.

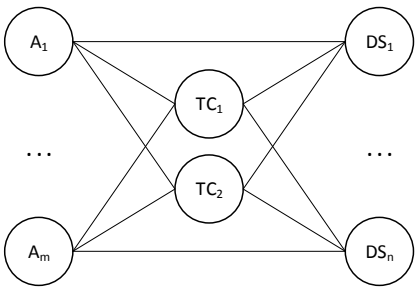


Figure 1. DPSTM v3 based system architecture

Each application program $A_i$ comprises more transactions that operate either in SCM or ECM. A transaction operating in SCM requests services only from the MTC, which in its turn requests services from the DSs, and in this case the system architecture may be seen as a three-tier architecture. Alternatively, a transaction operating in ECM requests services from both MTC and its local DS that is located in the same machine or in some nearby machine, and in this case the system architecture may be seen as a three-tier architecture with an additional direct link between the first and the third tier.

Servers (TCs and DSs) are designed to be Python multi-processing managers with the intention that the architecture in Fig. 1 can be implemented as a natural extension of the existing Python language abstractions.

Fig. 2 shows the simplified UML class diagram of the DPSTM v3 based system. Generally, each application program $A_i$ comprises one or more transactions (TxNs) and the accompanying DPSTM clients that act as the transactions' proxies, each transaction coordinator $TC_j$, comprises the TC stub and the DS client that acts as $TC_j$'s proxy, and each data server $DS_k$ comprises the DS stub and the Dictionary that contains all the t-variables.

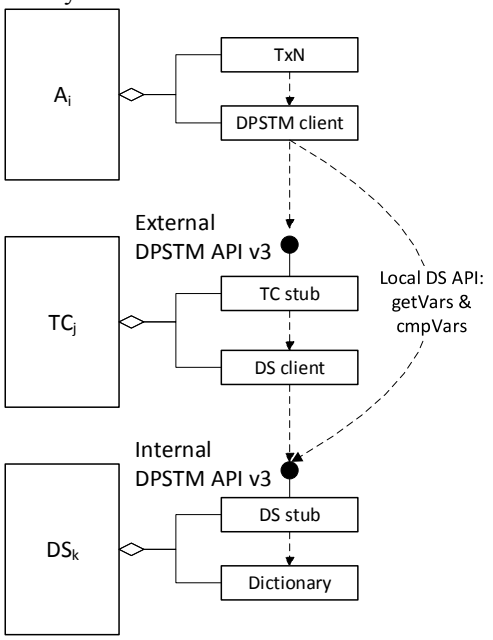


Figure 2. Simplified UML class diagram for the DPSTM v3 based system

A DPSTM client can be configured either for the SCM or for the ECM, whereas a DS client is always configured for the SCM. TC offers the external DPSTM API v3 to all the DPSTM clients, whereas DS offers the complete internal DPSTM v3 to all the DS clients and the local DS API to all the DPSTM clients that are configured for the ECM.

During a system startup, the clients establish authentication secured connections with their peer servers. More precisely, a DS client establishes connections with all the DSs, a DPSTM client configured for the SCM establishes connections with both TCs, and a DPSTM client configured for the ECM establishes connections with both TCs and its local DS.

During normal system operation, the DPSTM client configured for the SCM simply delegates all the TxN's requests to the master TC. Alternatively, the DPSTM client configured for the ECM delegates the TxN's requests for reading and comparing t-variables (see functions getVars and cmpVars in Section II.B) to its local DS, and all the other TxN's requests to the master TC.

### *B. DPSTM v3 APIs*

As already mentioned in Section II.A and shown in Fig. 2, the following three different APIs exist within the DSPTM v3: (i) the external DPSTM API v3, (ii) the internal DPSTM

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



API v3, and (iii) the local DS API.

The external DPSTM API v3 was created from DPSTM API v2 (see [27]) by introducing the new argument *txnid* at the end of the argument list of each API function (*txnid* is the transaction ID; these IDs are used internally to distinguish API calls issued by different transactions). However, DPSTM API v2 was kept as the internal interface between TxN and DPSTM client, within $A_i$ (see Fig. 2), in order to aid the easy porting of legacy software from previous DPSTM versions to DPSTM v3.

The four most important functions in the external DPSTM API v3 are: (i) addVars declares (adds) a new t-variable, (ii) putVars sets the initial value of a given t-variable, (iii) getVars returns (reads) the values of given t-variables, which are then used as TxN's local t-variables within the TxNs local processing function, and (iv) commitVars commits updates (writes) made on local t-variables to global (shared) t-variables kept in the DS's dictionary.

The internal DPSTM API v3 extends the external DPSTM API v3 with the following two functions: (i) getLastOp returns the tuple (*opc*, *opa*, *opr*, *txnid*) where *opc*, *opa*, *opr*, and *txnid* are the last executed operation code (i.e. API function name), its arguments, its return value, and transaction ID, respectively, and (ii) execLastOp executes the operation specified by the tuple (*opc*, *opa*, *opr*, *txnid*), i.e. it executes *opc* with arguments *opa*.

The local DS API reduces the internal DPSTM API v3 to the following two functions: (i) getVars (see above), and (ii) cmpVars that compares local and the corresponding global t-variables kept in the DS's dictionary.

### C. DPSTM v3 Operation

This subsection provides an overview of DPSTM v3 operation by presenting the following three most important scenarios: (i) a pair of application processes operating in SCM, (ii) a pair of application processes operating in ECM, and (iii) an example of a failover after MTC crashes. In the figures in this subsection (Fig. 3 to 5) the API function name commitVars is shortened to comVars in order to keep them readable. For simplicity, let's assume $n = 2$, i.e. that there are two data servers ($DS_1$ and $DS_2$) in the system.

Fig. 3 shows a typical scenario with a pair of application processes, $A_1$ and $A_2$, operating in SCM. At the beginning, $A_1$ and $A_2$ simultaneously issue their getVars calls to request their copies of t-variables – these calls are shown in Fig. 3 as $getVars_1$ and $getVars_2$, respectively. Let's assume that MTC is idle and that $getVars_1$ arrives at MTC before $getVars_2$, and thus $getVars_1$ gets enqueued in MTC's queue before $getVars_2$. After $getVars_1$ gets at the top of MTC's queue, MTC serves $getVars_1$ by reading t-variables required by $A_1$ and returning their copies to $A_1$ over the return value $retVal_1$. Next, MTC serves $getVars_2$ by reading t-variables required by $A_2$ and returning their copies to $A_2$ over the return value $retVal_2$. Note how MTC serializes $getVars_1$ and $getVars_2$ in order to provide them with the last versions of the required t-variables.

At the end, $A_1$ and $A_2$ simultaneously issue their commitVars calls to commit their updates of t-variables – these calls are shown in Fig. 3 as $comVars_1$ and $comVars_2$, respectively. Let's assume that $comVars_1$ gets enqueued in MTC's queue before $comVars_2$. After $comVars_1$ gets at the top of MTC's queue, MTC serves $comVars_1$ by delegating this request to $DS_1$ first. After receiving the return value $retVal_{11}$, MTC checks whether $comVars_1$ was successful (i.e. if $retVal_{11}$ contains 'yes'), and if yes (as assumed in Fig. 3) MTC delegates the same request to $DS_2$ next. After receiving the return value $retVal_{12}$, MTC checks it and returns the final return value $retVal_1$ to $A_1$. In the case when $comVars_1$ delegated to $DS_1$ is unsuccessful (which was not the case in Fig. 3), MTC immediately returns the corresponding return value $retVal_1$ to $A_1$.

After servicing $comVars_1$, MTC services $comVars_2$ analogously as it serviced $comVars_1$, and in the end, sends $retVal_2$ to $A_2$. Note how MTC again serializes $comVars_1$ and $comVars_2$ in order to provide atomicity of the corresponding updates. Also, note how MTC delegates $comVars_1$ (and $comVars_2$) to $DS_1$ and to $DS_2$, in order to provide (i.e. conduct) data replication.

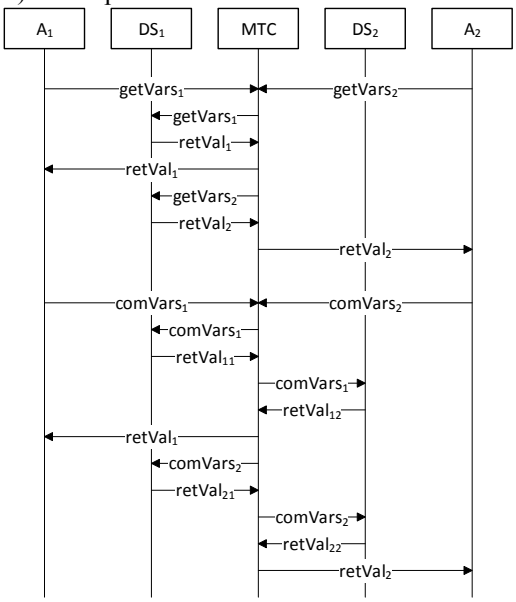


Figure 3. A pair of application processes operating in SCM

Fig. 4 shows a typical scenario with a pair of application processes, $A_1$ and $A_2$, operating in ECM, where $A_1$ is a producer kind of process that periodically updates some t-variables, and $A_2$ is a consumer type of process that periodically reads some t-variables (using getVars) from its local data server $DS_2$ and does some data processing based on their values, which do not need to be fresh (i.e. up to date). For simplicity, Fig. 4 shows only one producer and one consumer, but generally, there might be more producers and consumers in the system. Also for simplicity, Fig. 4 shows only the most important part of the system operation, which focuses on commitVars call from $A_1$ and getVars call from $A_2$, where these two calls are serviced in parallel by MTC and $DS_2$, respectively.

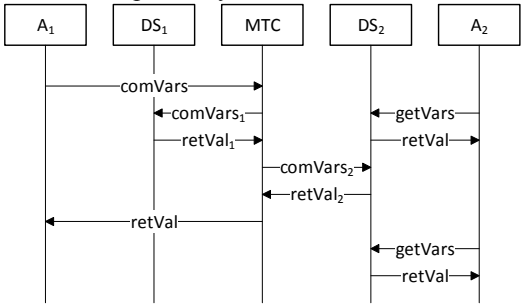


Figure 4. A pair of application processes operating in ECM.

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



At the beginning, $A_1$ issues its commitVars call to MTC (shown as comVars in Fig. 4), and MTC in its turn uses the already described deterministic replication protocol to updated $DS_1$ and $DS_2$ (in that order). MTC conducts this protocol by sequentially delegating comVars to $DS_1$ and $DS_2$ (shown as $comVars_1$ and $comVars_2$ in Fig. 4, respectively). Note that after the base replica $DS_1$ is updated, the system enters the inconsistent state, because some t-variables in $DS_1$ and $DS_2$ have different versions and values. However, the system will return back to the consistent state immediately after $DS_2$ is also updated – this property of the system is called eventual consistency. Also note that while the system is in the inconsistent state, there are at most two versions of a t-variable undergoing updating – its current version in the base replica (and other DSs that are already updated) and its previous version in DSs that are not already updated (in Fig. 4 this may only be $DS_2$).

Consequently, an application process that uses its own local DS (in Fig. 4, $A_2$ uses $DS_2$) may either get the previous version or the current version of a t-variable. Fig. 4 illustrates both cases. The first getVars call from $A_2$ is serviced by $DS_2$ while $DS_1$ services $comVars_1$, and while $DS_2$ still has not been updated, so the corresponding retVal returned to $A_2$ contains the previous version (and value) of a requested t-variable. Alternatively, the second getVars call from $A_2$ is serviced by $DS_2$ after $DS_2$ has been updated, so the corresponding retVal returned to $A_2$ contains the current version (and value) of a requested t-variable.

Obviously, using local data servers leads to better performance (at the cost of the possible use of previous versions of t-variables). Note that a system configuration in which application processes may (statically or dynamically) select their local servers from a set of (geographically) close DSs, opens an opportunity for load balancing getVars calls. At this time, a DPSTM v3 based system designer must manually perform static load balancing by specifying local DSs in configuration files. We plan to develop automatic load balancing solutions in our future work, possibly by using some of the techniques presented in [41].

Fig. 5 shows a typical scenario of a failover that is initiated by an application process $A_1$ after regular servicing of its commitVars call (or any other API call with side effects) was broken by the MTC crash (that happened during the regular replication protocol and at some point after the base replica was successfully updated). Besides this particular scenario, there are three more rather similar scenarios, which will be covered later in Section III.

In the scenario in Fig. 5, TxN's DPSTM client, within $A_1$, at the beginning internally starts a timer and issues its commitVars call to MTC, which in turn starts executing the regular data replication protocol and successfully updates the base replica $DS_1$ (see $comVars_1$ and the corresponding $retVal_1$ in Fig. 5). Before issuing $comVars_2$, MTC gets crashed, and therefore it never sends retVal to TxN's DPSTM client. Note that at this point the system is in the inconsistent state because data servers contain different data.

After a specified timeout period expires, the internal timer expires, thus TxN's DPSTM client detects the MTC crash and initiates the failover (i.e. switching STC into new MTC) by issuing setStatus call to STC, with the argument 'master'. At this point, STC switches to MTC and starts the procedure of fixing a broken service (i.e. broken replication protocol execution) in order to return the system into the consistent state (where all data servers contain the same data).

The new MTC fixes the broken commitVars call service as follows. In the first step, MTC issues getLastOp call to the base replica $DS_1$ (shown as $getLastOp_1$ call in Fig. 5) and gets the data about the last operation $op_1$ (i.e. the last API call service) performed by $DS_1$ within the return value $reatVal_1$. In the second step, MTC issues getLastOp call to $DS_2$ (shown as $getLastOp_2$ call in Fig. 5), gets the data about the last operation $op_2$ performed by $DS_2$, compares the data about $op_2$ and $op_1$ and discovers that they are different, i.e. that the system is in the inconsistent state. In the third step, MTC requests $DS_2$ to perform the operation $op_1$ (that was performed by the base replica) by issuing execLastOp call to $DS_2$ (shown as $execLastOp_2$ in Fig. 5), gets the return value of that operation within the return value $retVal_2$ from $DS_2$, and asserts that it is the same as the return value for $op_1$. Note that generally in the case on $n$ data servers, the steps two and three above are repeated $(n - 1)$ times. Note also that after this procedure is finished, the system is returned back to the consistent state.

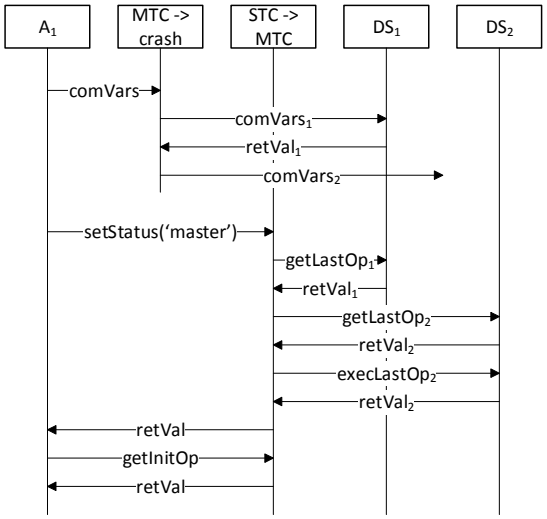


Figure 5. A typical scenario of a failover caused by MTC crash

After the new MTC has fixed the commitVars call, it returns the return value retVal to TxN's DPSTM client within $A_1$, which in turn issues getInitOp call to the new MTC in order to get the data about the fixed operation, i.e. $op_1$, and returns the $op_1$'s return value to TxN within $A_1$. The system will continue to work with a single MTC until the crashed TC is repaired and put back into operation.

## III. DPSTM v3 Implementation

Although MSM is widely studied and used, in order to introduce a pair of TCs operating in MSM in DPSTM, we had to (i) rethink who and how is governing MSM, then to (ii) adapt the DPSTM v2 replication protocol, and finally to (iii) solve the problem of fixing broken services. The solutions for these three problems are given in the following four subsections (the solution for the first problem is given in subsections III.A and III.B, and so on).

### *A. Distributed MS-FSM*

The distributed *master-slave finite state machine* (MS-FSM) solution assumes that underlying communications

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



infrastructure is reliable, i.e. in terms of CAP theorem [42], it sacrifices network partition-tolerance for consistency and availability.

The distributed MS-FSM is implemented as a set of per-transaction MS-FSMs, where each MS-FSM is contained within the corresponding DPSTM client's object. An individual MS-FSM is implemented as a pair of TC proxy objects and the two DPSTM client's functions that perform MS-FSM state transitions, namely getTCs and doOp. These two functions are hierarchically organized such that getTCs performs transitions based on the current state reported by TCs, whereas doOp performs only transitions related to failover and it uses getTCs when needed. Thus getTCs is subordinated to doOp. For clarity, this subsection describes getTCs and Section III.B describes doOp.

The function getTCs constructs and returns the pair of TC proxy objects (*tcm*, *tcs*), where *tcm* and *tcs* are the proxies for MTC and STC, respectively. Objects *tcm* and *tcs* are stored in the corresponding DPSTM client object's fields.

Initially, a transaction calls the DPSTM client's constructor and passes its transaction ID (*txnid*) as an argument, and the constructor in its turn calls the function getTCs in order to fill in the corresponding fields. The function getTCs then effectively both creates the pair of proxy objects (*tcm*, *tcs*) and updates the states of two TCs in the system (internally represented by the objects $tc_1$ and $tc_2$, respectively) by executing the required state transition in accordance with Table I.

TABLE I. DPSTM CLIENT'S MS-FSM TRANSITIONS BY GETTCS

| Current state | | Next state | | Output (*tcm*, *tcs*) | |
|---|---|---|---|---|---|
| $TC_1$ | $TC_2$ | $TC_1$ | $TC_2$ | *tcm* | *tcs* |
| OFF | OFF | OFF | OFF | None | None |
| slave | OFF | master | OFF | $tc_1$ | None |
| OFF | slave | OFF | master | $tc_2$ | None |
| slave | slave | master | slave | $tc_1$ | $tc_2$ |
| master | OFF | master | OFF | $tc_1$ | None |
| OFF | master | OFF | master | $tc_2$ | None |
| master | slave | master | slave | $tc_1$ | $tc_2$ |
| slave | master | slave | master | $tc_2$ | $tc_1$ |
| master | master | OFF | OFF | Error | Error |
| Note: TC state transition from master to slave is forbidden. | | | | | |

An individual TC may be in one of the following three possible states: OFF (i.e. not operational), slave, and master. Therefore, the pair of TCs, ($TC_1$, $TC_2$), theoretically could be in one of the nine possible states (3 x 3 = 9), see Table I. The ninth state (master, master) is forbidden (entering this state corresponds to a fatal system error). Similarly, the first state (OFF, OFF) is also not a valid operational state, because the system cannot operate without TCs.

By design, the allowed state transitions of an individual TC are: (i) from OFF to OFF or slave, (ii) from slave to slave or master, and (iii) from master to master. The state transition (i) is taken by a TC on its own, whereas the state transitions (ii) and (iii) are made within getTCs by setStatus calls on TC's proxy. Also by design, the state of an individual TC cannot change from master to slave.

Note that not all the state transitions in Table I change the state of the TCs. More precisely, the state will be changed only for the following three current states: (slave, OFF), (OFF, slave), and (slave, slave); for other valid current states, the next state is the same as the current state. Note also that if more DPSTM clients issue the same setStatus call in parallel, the first serviced call would be successful and the rest of them would be harmless and without effect.

### *B. Distributed Failover*

The primary task of the function doOp is to service an API call issued by a transaction. By design, servicing an API call is called an operation, *Op* = (*code*, *args*, *ret*, *txnid*), where *code*, *args*, *ret*, and *txnid* are the API function name (e.g. addVars, putVars, etc), the API function arguments, the API return value, and the transaction ID, respectively. When given an instance *op* of *Op*, we use common dot notation to refer to *op*'s elements, e.g. *op.code* is *op*'s code, etc.

The secondary task of the function doOp is to detect MTC crash and initiate the failover by executing the required state transition in accordance with Table II.

TABLE II. DPSTM CLIENT'S MS-FSM TRANSITIONS BY DOOP

| Current state | | Next state | | Output (*tcm*, *tcs*) | |
|---|---|---|---|---|---|
| $TC_1$ | $TC_2$ | $TC_1$ | $TC_2$ | *tcm* | *tcs* |
| master | slave | OFF | master | $tc_2$ | None |
| slave | master | master | OFF | $tc_1$ | None |

The pseudocode for the function doOp is given in Fig. 6. There are five possible paths through the function doOp, namely path 1, path 2.1, path 2.2, path 3.1, and path 3.2, The paths 2.1 and 2.2 are the subpaths of the path 2, which is called the *regular failover*. Similarly, the paths 3.1 and 3.2 are the subpaths of the path 3, which is called the *fast failover*.

```
01: doOp(op)
02:   try
03:     delegate op to tcm  // path 1: normal operation
04:   exception
05:     if tcs != None
06:       try
07:         // path 3: fast failover
08:         tcs.setStatus('master')
09:         initOp = tcs.getInitOp()
10:         tcm := tcs, tcs := None
11:         if op = initOp
12:           return initOp.ret     // path 3.1
13:         else
14:           delegate op to tcm  // path 3.2
15:       exception // falls to path 2
16:     // path 2: regular failover
17:     tcm, tcs = getTCs()
18:     initOp = tcm.getInitOp()
19:     if op = initOp
20:       return initOp.ret     // path 2.1
21:     else
22:       delegate op to tcm  // path 2.2
```

Figure 6. The function doOp

Note that more DPSTM clients may execute their copies of doOp functions in parallel. Also, for one of them, its operation (i.e. service) is broken because of MTC crash. Let $c_1$ and $c_2$ be two Boolean variables assigned to a DPSTM client *c*, such that $c_1$ is true if *c*'s operation was broken, and $c_2$ is true if *c*'s *tcs* is not None (i.e. *tcs* is linked to STC).

Obviously, there are four possible cases for a client *c* executing doOp: (1) $c_1$ is true and $c_2$ is true (path 3.1), (2) $c_1$

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



is false and $c_2$ is true (path 3.2), (3) $c_1$ is true and $c_2$ is false (path 2.1), and (4) $c_1$ is false and $c_2$ is false (path 2.2).

Let's now clarify how transitions in Table II take place. Without loss of generality, assume that the current state of *c*'s MS-FSM is (master, slave) and that because of MTC crash the real system state has changed to (OFF, slave). If $c_2$ is true and STC is in service, *c* takes the path 3 and directly changes its MS-FSM state to (OFF, master) by executing line 10 in Fig. 6. The analogous analysis may be conducted when the current state of *c*'s MS-FSM is (slave, master).

Note that if $c_2$ is false, *c* takes the path 2, where it calls getTCs, and within getTCs performs a state transition according to Table I.

### C. Adapted Deterministic Replication Protocol

MTC uses the adapted deterministic replication protocol when servicing API calls with side effects, i.e. causing write operations on dictionaries within DSs (addVars, putVars, commitVars, etc.). The core of this protocol is rather simple, and practically the same as the original protocol used by DPSTM v2: MTC sequentially delegates an API call to be serviced to each DS, one by one, and always in the order from $DS_1$ (base replica) to $DS_n$. Recall how MTC serviced the incoming commitVars calls in the typical scenario shown in Fig. 3.

The difference between the adapted and the original protocol is in the way how DSs service the API calls. In the original protocol, a DS just performs the requested operation on its dictionary, whereas in the adapted protocol a DS also saves this operation as the last operation it performed (DS keeps just a single last operation, not the history/log of operations).

### D. Procedure for Fixing Broken Service

The new MTC (i.e. former STC) uses the procedure for fixing a broken service of an API call (i.e. operation) in order to return the system from an inconsistent state back to a consistent state, as already explained and illustrated in Fig. 5 in Section II.C.

The core of this procedure is rather simple. At the beginning, the new MTC gets the last operation performed by the data server $DS_1$ (base replica) by calling the function getLastOp on $DS_1$'s proxy, and stores it in its variable *initOp* (for the new MTC, this is the initial operation it starts from). After that, it performs the loop to check (and fix) the rest of DSs (fixing a DS means aligning it with $DS_1$, so $DS_1$ is always fixed). Within the loop, the new MTC gets the last operation performed by $DS_i$, ($i$ = 2, …, $n$), *lastOp*, and compares it with *initOp*. If *lastOp* is not equal to *initOp*, the new MTC requests $DS_i$ to perform *initOp* by calling the function execLastOp on the $DS_i$'s proxy; otherwise, it just continues into the next loop iteration.

## IV. DPSTM v3 Experimental Evaluation

This section presents DPSTM v3 experimental evaluation, which is based on measuring the system throughput, which is here defined as the number of transactions per second, for the six different workloads. These workloads are different mixtures of the two kinds of primitive transactions, namely the *read transactions* and the *write transactions*. As their names suggest, the former perform read operations (using getVars), whereas the latter perform write operations (using putVars). The experimental evaluation is made as a benchmark with two typical network configurations, namely the *fully distributed network configuration* (wherein each system component runs on a separate machine) and the *collocated network configuration* (wherein application processes and their local data servers are collocated).

The two main goals of the experimental evaluation were to demonstrate that: (i) the system throughput increases superlinearly (i.e. linearly or better) as the portion of read operations increases from 0% to 100% (and the portion of write operations symmetrically decreases from 100% to 0%), and (ii) the system throughput for the collocated network configuration is much better than for the fully distributed network configuration. The results of the experimental evaluation in Section IV.D show that both goals are successfully achieved.

### A. Benchmark Application

The benchmark application is itself a typical DPSTM based application, which consists of a master process and a group of five worker processes. The master process performs system initialization, system orchestration, and calculation and reporting of the final test results (system throughput and its standard deviation). On the other hand, each worker process performs its portion of the specified workload, calculates its individual throughput, and returns its result to the master process.

The master process synchronizes and communicates with the worker processes over DPSTM. The master and worker processes synchronize using an asymmetric barrier like mechanism based on DPSTM, which is implemented by the two functions, namely masterBarrier and workerBarrier. As their names suggest, the former function is used by the master process, whereas the latter is used by a worker process. Besides the synchronization, DPSTM is used both by a worker process to deliver its result by writing it into the corresponding t-variable, and by the master process to collect the results of the worker processes by reading the corresponding t-variables.

According to the globally known schedule, the benchmark application conducts the six phases that correspond to the six different workloads, i.e. mixtures of read and write transactions. These workloads are the following: (i) 0% reads plus 100% writes, (ii) 20% reads plus 80% writes, (iii) 40% reads plus 60% writes, (iv) 60% reads plus 40% writes, (v) 80% reads and 20% writes, and (vi) 100% reads plus 0% writes.

In each phase, a worker process acts either as a *reader* process that performs read transactions or a *writer* process that performs write transactions. So, an alternative way to specify the six workloads listed above is to specify the number of reader processes and the number of writer processes for each phase. For example, the workload (iii) above, with 40% reads plus 60% writes, corresponds to the phase with two reader processes and three writer processes.

The two most important parameters used by the benchmark application are NTXNS and NMESR. NTXNS is the number of either read or write transactions performed by a single worker process per each phase (in this paper, this parameter was set to 10000). NMESR is the number of

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



execution time measurements taken for each phase (i.e. the corresponding workload), which are then used to calculate the average throughput for a phase (in this paper, this parameter was set to 3).

The execution of each phase is organized as follows. The master process calls masterBarrier twice in a row – the first time to wait for all the worker processes and signal them to start the phase, and the second time to wait for them to finish the phase. After the second masterBarrier call returns, it calculates the phase statistics. On the other hand, each worker process first calls workerBarrier to wait for a start signal from the master process, then it performs its part of the workload for the current phase (i.e. it performs NMESR packets of NTXNS transactions), and calls workerBarrier to signal that it finished its task. At the end of the sixth phase, the master process stores and reports the final results.

### *B. Test Network Configurations*

As already mentioned, the experimental evaluation was made on the two test network configurations – the fully distributed network configuration (FDNC) and the collocated network configuration (CLNC). Both test network configurations comprise: two DSs ($DS_1$ and $DS_2$), two TCs ($TC_1$ and $TC_2$), one benchmark application's master process (M), and five benchmark application's worker processes ($W_1$ to $W_5$) – altogether ten system components. For the sake of load balancing, in both test network configurations, M, $W_2$, and $W_4$ use $DS_1$ as their local DS, whereas $W_1$, $W_3$, and $W_5$ use $DS_2$ as their local DS.

The main difference between the two test network configurations is in the way the ten system components are deployed on the separate machines in the target physical network. In FDNC, each system component is deployed to its own machine (i.e. the ten components are deployed on ten separate machines), whereas in CLNC, benchmark application's processes are collocated with their local data servers and TCs are deployed on their own machines. So, in CLNC four machines are used altogether – two machines by two TCs, one machine by $DS_1$, M, $W_2$, and $W_4$, and one machine by $DS_2$, $W_1$, $W_3$, and $W_5$.

### *C. Experimental Setup*

The experimental setup characteristics are as follows. LAN characteristics: Computer sockets in the laboratory are connected to a single Cisco Catalyst WS-C2960-48TT-L switch. This switch provides 10/100Mbps user access ports and 1Gbps uplinks (uplinks were not used by the benchmark application). Network cables are UTP category CAT5e. Computer characteristics: CPU - Intel Corei7-7700 CPU @ 3.60GHz (4 cores x64), MB - Asus Prime B250M-C, RAM - 1 module x 16 GB DDR4 @ 2133MHz, HDD - 2TB (Toshiba), Graphics - Intel(R) HD Graphics 630. Software versions: OS - Windows 10 Professional (x64). Python version 3.7.3 (Release Date: March 25, 2019) [MSC v.1916 64 bit (AMD64)] on win32.

### *D. Results and Discussion*

The experimental results are given in Table III and illustrated in Fig. 7. The rows in Table III correspond to the six workloads, whereas the columns in Table III correspond to: the portion of read operations (r), the system throughput for FDNC (st-1), the standard deviation for st-1 (stdev-1), the system throughput for CLNC (st-2) and the standard deviation for st-2 (stdev-2).

The horizontal axis in Fig. 7 shows the values for the portion of read operations (r), whereas the vertical axis in Fig. 7 shows the values of the system throughput in transactions per second (txns/s). The solid and the dashed curves show the values of st-1 and st-2, respectively.

TABLE III. EXPERIMENTAL RESULTS

| r [%] | st-1 [txns/s] | stdev-1 [%] | st-2 [txns/s] | stdev-2 [%] |
|---|---|---|---|---|
| 0 | 1579 | 0.23 | 1498 | 0.59 |
| 20 | 2320 | 0.75 | 13937 | 1.47 |
| 40 | 2949 | 0.28 | 27187 | 1.03 |
| 60 | 3920 | 3.90 | 39428 | 1.28 |
| 80 | 4432 | 1.90 | 52195 | 0.23 |
| 100 | 5636 | 1.45 | 54329 | 2.49 |

As indicated by the values of st-1 in Table III (the solid curve in Fig. 7), st-1 increases superlinearly as r increases, from 1579 txns/s up to 5636 txns/s (and the values of stddev-1 are quite acceptable). In the case of st-1, the main limitation of the system is the capacity of the communication equipment.

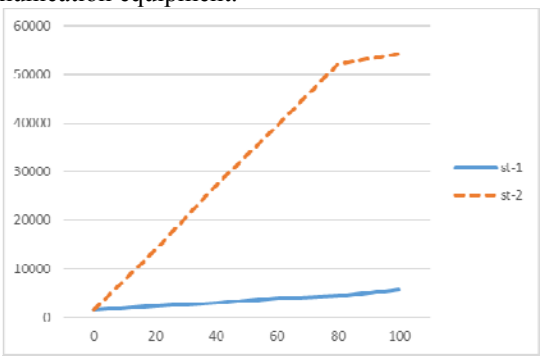


Figure 7. Illustration of experimental results from Table III

Similarly, as indicated by the values of st-2 in Table III (the dashed curve in Fig. 7) st-2 increases linearly as r increases, up to the value of r = 80%, and from r = 80% to r = 100% the st-2 increase is slower because the machines hosting the application processes and their local data servers reach their capacity at r = 80%. These machines have only 4 cores and they have to host four system components plus the local operating system (and its processes). For this reason, there are not enough cores available to run all the system components in parallel, and consequently, the machines cannot deliver the required service at the required pace. In the future work, we plan to conduct more experiments to justify this assumption.

Finally, by comparing st-2 to st-1, we see that the values for st-2 are the order of magnitude higher, and this justifies the benefit of collocating application processes with their local data servers.

### *E. Advantages and Limitations*

In Section I.C we explained our approach to experimental evaluation, which is based on synthetic workloads that are made as mixtures of read and write transactions. In this subsection we discuss the advantages and limitations of that approach from the perspective of the particular experimental evaluation made for DPSTM v3 in this section, starting with

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]



the advantages.

The first advantage is that it provides good coverage of the whole range of possible applications. In this particular evaluation, we used six different workloads, which seem to be sufficient here because the curves in Fig. 7 are rather smooth. In principle, we may introduce more workloads if we want to get even better coverage of the whole range of applications.

The second advantage is that by using this approach we can get the overall trends, as well as lower and upper bounds. For example, by looking at the curves in Fig. 7, and data in Table III, we can see that the system throughput increases as the portion of read operations increases. This means that our DPSTM v3 performs well and as expected.

The third advantage is that we may use the result of this approach (the obtained data) to estimate the performance of an arbitrary workload by using interpolation. For example, by using the values of st-2 for r = 40% and r = 60%, we can estimate the value of st-2 for r = 50%, and make a statement that we expect the value of st-2 for r = 50% to be around 33307 (read and write transactions in a second). We may even make such estimations for dynamic workloads that change their behavior over time, if we have their profiles, i.e. if we know how the mixture of read and write operations change in these workloads over time. This possibility is rather important for the engineering of systems based on DPSTM v3, in particular when calculating the system capacity.

We now turn to the limitations of this particular experimental evaluation. The first limitation is that it covers only two network configurations. We assume that these two configurations are the two opposite extremes of the whole range (i.e. dimension) of possible network configurations, and that the values of throughput for these two network configurations represent the lower and the upper bounds on throughput, respectively. If this assumption is true, then we could expect that the curve representing the performance of an arbitrary network configuration would be somewhere between the curves st-1 and st-2 in Fig. 7. We plan to justify this assumption in our future work by evaluating DPSTM v3 on additional network configurations.

The second limitation of this particular experimental evaluation is that it was made for one particular experimental setup using particular hardware and network technology, namely quad-core desktop PCs connected to a wired LAN with 100Mbps access ports. On the other hand, IoT systems typically use wireless network technologies, such as WiFi, Zigbee, etc. Therefore, we plan to evaluate DPSTM v3 on these wireless network technologies in our future work.

## V. Conclusion

To the best of our knowledge, this is the first paper that presents a distributed software transactional memory that is at the same time: (i) fault tolerant, (ii) deterministic, (iii) based on Python, and (iv) extended from a formally verified root. The presented solution consists of a pair of master-slave transaction coordinators and a set of replicated data servers, and it targets small-to-medium Internet edge networks. Besides intelligent embedded systems based on the IoT, at the edge of the Internet, such as smart homes, cars, etc., it can also be used in a wide range of application domains, from SCADA systems to large-scale simulations.

The main novel contributions of the paper are: (i) the novel solution to control the pair of TC working in master-slave mode by a set of distributed finite state machines, (ii) the novel solution for the failover after the master TC crash, (iii) the novel solution of the adapted deterministic replication protocol, (iv) the novel solution for the deterministic recovery of the broken service after the master TC crash, (v) the novel architectural solution by which TCs and DSs are implemented as Python's remote managers, (vi) the novel architectural solution by which DPSTM v3 architecture is constructed as an extension of the formally verified root architecture.

The experimental results show that: (i) the system throughput increases superlinearly as the portion of read operations increases from 0% to 100% (and the portion of write operations symmetrically decreases from 100% to 0%), and (ii) the system throughput for the collocated network configuration is much better than for the fully distributed network configuration.

The advantages of the experimental evaluation approach can be summarized in the following way: (i) it provides good coverage of the whole range of possible applications, (ii) it provides the overall trends, as well as lower and upper bounds, and (iii) it enables performance (i.e. throughput) estimation for an arbitrary workload by using interpolation.

The limitations of the particular DPSTM v3 experimental evaluation presented in this paper are the following: (i) it covers only two network configurations, and (ii) it was made for one particular experimental setup using particular hardware and network technology.

In the future work, we plan to: (i) continue experimental evaluation on other network configurations, as well as using other hardware and network technology, and (ii) research other possible DPSTM architectures.

## References

[1] M. Herlihy, J.E.B. Moss, "Transactional memory: architectural support for lockfree data structures," in Proc. 20th Annual International Symposium on Computer Architecture, pp. 289–300, 1993. doi:10.1145/165123.165164

[2] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in Proc. of the 12th International Symposium on Computer Architecture, pp. 124–131, 1983. doi:10.1145/800046.801647

[3] N. Shavit, D. Touitou, "Software transactional memory," in Proc. 14th ACM Symposium on Principles of Distributed Computing, pp. 204–213, 1995. doi:10.1145/224964.224987

[4] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster, " in Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 198–208, 2006. doi:10.1145/1122971.1123002

[5] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C.C. Kirkham, I. Watson, "DiSTM: A software transactional memory framework for clusters," in Proc. 37th International Conference on Parallel Processing, pp. 51-58, 2008. doi:10.1109/ICPP.2008.59

[6] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C.C. Kirkham, I. Watson, "Investigating software transactional memory on clusters," in Proc. 22nd International Parallel and Distributed Processing Symposium, pp. 1-6, 2008. doi:10.1109/IPDPS.2008.4536340

[7] M. Couceiro, P. Romano, N. Carvalho, L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in Proc. 15th Pacific Rim International Symposium on Dependable Computing, pp. 307-313, 2009. doi:10.1109/PRDC.2009.55

[8] J. Cachopo, A. Rito-Silva, "Versioned boxes as the basis for memory transactions," Science of Computer Programming, vol. 63, no. 2, pp. 172-185, 2006. doi:10.1016/j.scico.2006.05.009

[Downloaded from www.aece.ro on Thursday, July 03, 2025 at 20:09:59 (UTC) by 172.70.127.46. Redistribution subject to AECE license or copyright.]




[9] J. Cachopo, A. Rito-Silva, "Combining software transactional memory with a domain modeling language to simplify web application development," in Proc. 6th International Conference on Web Engineering, pp. 297–304, 2006. doi:10.1145/1145581.1145640

[10] P. Romano, N. Carvalho, L. Rodrigues, "Towards distributed software transactional memory systems," in Proc. 2nd Workshop on Large-Scale Distributed Systems and Middleware, pp. 1-4, 2008. doi:10.1145/1529974.1529980

[11] N. Carvalho, J. Cachopo, L. Rodrigues, A. Rito-Silva, "Versioned transactional shared memory for the FenixEDU web application," in Proc. 2nd Workshop on Dependable Distributed Data Management, pp. 15–18, 2008. doi:10.1145/1435523.1435526

[12] R. L. Bocchino, V. S. Adve, B. L. Chamberlain, "Software transactional memory for large scale clusters," in Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 247–258, 2008. doi:10.1145/1345206.1345242

[13] J. Kim, B. Ravindran, "Scheduling transactions in replicated distributed software transactional memory," in Proc. 13th IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing, pp. 227–234, 2013. doi:10.1109/CCGrid.2013.88

[14] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues, "On speculative replication of transactional systems," Journal of Computer and System Sciences, vol. 80, no. 1, pp. 257–276, 2014. doi:10.1016/j.jcss.2013.07.006

[15] M. M. Saad, B. Ravindran, "Snake: Control flow distributed software transactional memory," in Proc. Stabilization, Safety, and Security of Distributed Systems, pp. 238–252, 2011. doi:10.1007/978-3-642-24550-3_19

[16] M. Herlihy, Y. Sun, "Distributed transactional memory for metric-space networks," Distributed Computing, vol. 20, no. 3, pp. 195–208, 2007. doi:10.1007/s00446-007-0037-x

[17] G. Sharma and C. Busch, "Distributed transactional memory for general networks," Distributed Computing, vol. 27, no. 5, pp. 329–362, 2014. doi:10.1007/s00446-014-0214-7

[18] B. Zhang, B. Ravindran, R. Palmieri, "Distributed transactional contention management as the traveling salesman problem," in Proc. Structural Information and Communication Complexity, pp. 54–67, 2014. doi:10.1007/978-3-319-09620-9_6

[19] C. Busch, M. Herlihy, M. Popovic, G. Sharma, "Fast scheduling in distributed transactional memory," in Proc. Symposium on Parallelism in Algorithms and Architectures, pp. 173–182, 2017. doi:10.1145/3087556.3087565

[20] C. Busch, M. Herlihy, M. Popovic, G. Sharma, "Time-communication impossibility results for distributed transactional memory," Distributed Computing, vol. 31, no. 6, pp. 471–487, 2018. doi:10.1007/s00446-017-0318-y

[21] D. Hendler, A. Naiman, S. Peluso, F. Quaglia, P. Romano, A. Suissa, "Exploiting locality in lease-based replicated transactional memory via task migration," in Proc. International Symposium on Distributed Computing, pp. 121–133, 2013. doi:10.1007/978-3-642-41527-2_9

[22] R. Palmieri, S. Peluso, B. Ravindran, "Transaction execution models in partially replicated transactional memory: The case for data-flow and control-flow," in: R. Guerraoui, P. Romano (Eds.) Transactional Memory – Foundations, Algorithms, Tools, and Applications, LNCS, vol. 8913, Springer, pp. 341–366. 2015. doi:10.1007/978-3-319-14720-8_16

[23] S. Bagchi, M.-B. Siddiqui, P. Wood, H. Zhang, "Dependability in Edge Computing," Communications of ACM, vol. 63, no. 1, pp. 59-66, 2020. doi:10.1145/3362068

[24] K. Gilly, S. Filiposka, A. Mishev "Supporting Location Transparent Services in a Mobile Edge Computing Environment," Advances in Electrical and Computer Engineering, vol. 18, no. 4, pp. 11-22, 2018. doi:10.4316/AECE.2018.04002

[25] M. Popovic, B. Kordic, "PSTM: Python Software Transactional Memory," in Proc. 22nd IEEE Telecommunications Forum, pp. 1106-1109, 2014. doi:10.1109/TELFOR.2014.7034600

[26] A. Liu, H. Zhu, M. Popovic, S. Xiang, L. Zhang, "Formal Analysis and Verification of the PSTM Architecture Using CSP", The Journal of Systems and Software, vol. 165, 2020. doi:10.1016/j.jss.2020.110559

[27] B. Kordic, M. Popovic, S. Ghilezan, "Formal verification of python software transactional memory based on timed automata", Acta Polytechnica Hungarica, Journal of Applied Sciences, vol. 16, no. 7, pp. 197-216, 2019. doi:10.12700/APH.16.7.2019.7.12

[28] M. Popovic, M. Popovic, S. Ghilezan, B. Kordic, "Formal verification of python software transactional memory serializability based on the push/pull semantic model," in Proc. 6th Conference on the Engineering of Computer Based Systems, Article No. 6, pp. 1-8, 2019. doi:10.1145/3352700.3352706

[29] M. Popovic, B. Kordic, M. Popovic, I. Basicevic, "Online algorithms for scheduling transactions on python software transactional memory," Serbian Journal of Electrical Engineering, vol. 16, no. 1, pp. 85-104, 2019. doi:10.2298/SJEE1901085P

[30] C. Xu, X. Wu, H. Zhu, M. Popovic, „modeling and verifying transaction scheduling for software transactional memory using CSP," in Proc. 13th Theoretical Aspects of Software Engineering Symposium, pp. 240-247, 2019. doi:10.1109/TASE.2019.00009

[31] M. Popovic, B. Kordic, M. Popovic, I. Basicevic, "A solution of concurrent queue on local and distributed python STM," Telfor Journal, vol. 11, no. 1, pp. 64-69, 2019. doi:10.5937/telfor1901064P

[32] M. Popovic, M. Popovic, B. Kordic, I. Basicevic, "A solution of python distributed stm based on data replication," in Proc. 27th IEEE Telecommunications Forum, pp. 1-4, 2019. doi:10.1109/TELFOR48224.2019.8971069

[33] M. van Steen, A.S. Tanenbaum, Distributed Systems, 3rd edition, Published by Maarten van Steen, pp. 355-421, 2017.

[34] W. Vogels, "Eventually consistent," Communications of the ACM, vol. 52, no. 1, pp. 40–44, 2009. doi:10.1145/1435417.1435432

[35] D.J. Abadi, J.M. Faleiro, "An overview of deterministic database systems," Communications of the ACM, vol. 81, no. 9, pp. 78–88, 2018. doi:10.1145/3181853

[36] M. Popovic, M. Popovic, S. Ghilezan, B. Kordic, "Formal Verification of local and distributed python software transactional memory," Revue Roumaine des Sciences Techniques. Ser. Electrotechnique et Energetique, vol. 64, no. 4, pp. 423–428, 2019.

[37] E. Nan, U. Radosavac, M. Matić, I. Stefanović, I. Papp and M. Antić, "One solution for voice enabled smart home automation system," in Proc. IEEE 7th International Conference on Consumer Electronics - Berlin, pp. 132-133, 2017. doi:10.1109/ICCE-Berlin.2017.8210611.

[38] E. Nan, U. Radosavac, I. Papp and M. Antić, "Architecture of voice control module for smart home automation cloud," in Proc. IEEE 7th International Conference on Consumer Electronics - Berlin, pp. 97-98, 2017. doi:10.1109/ICCE-Berlin.2017.8210601.

[39] B. Kordic, M. Popovic, M. Popovic, M. Goldstein, M. Amitay, D. Dayan, "A protein structure prediction program architecture based on a software transactional memory," in Proc. 6th Conference on the Engineering of Computer Based Systems, Article No. 1, pp. 1-9, 2019. doi:10.1145/3352700.3352701

[40] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in Proc. of the 2010 USENIX conference on USENIX annual technical conference, pp. 145-158, 2010.

[41] L. Martinovic, D. Capko, A. Erdeljan, "Load balancing of large distribution network model calculations," Advances in Electrical and Computer Engineering, vol. 17, no. 4, pp. 11-18, 2017. doi:10.4316/AECE.2017.04002

[42] S. Gilbert, N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM SIGACT News, vol. 33, no. 2, pp. 51-59, 2002. doi:10.1145/564585.564601