Research Journal of Information Technology

ISSN 1815-7432

Academic Journals Inc.

www.academicjournals.com

Research Journal of Information Technology 2 (4): 201-214, 2010
ISSN 1815-7432

A Replication-Based Distribution Approach for Tuple Space-Based Collaboration of Heterogeneous Agents

[1]Jiankuan Xing, [1]Zheng Qin and [2]Jinxue Zhang
[1]Department of Computer Science and Technology,
Tsinghua University, Beijing 100084, Beijing, People's Republic of China
[2]School of Software, Tsinghua University, Beijing 100084, Beijing,
People's Republic of China

Abstract: Decentralized tuple space implements tuple space model among a series of decentralized nodes and provides the global shared tuple repository. But, in spite of its advantages, large access latency is brought by the frequent remote operations and the conventional performance improving approach that dynamic moving tuples among nodes ceases to be effective if tuple's locations are immutable, such as in LIME. In this study, a replication-based tuple distribution approach is proposed to improve performance of tuple access. A replicable decentralized tuple space model is presented, which gives semantics of tuple space primitives under replication. In this model, tuples are categorized as their use patterns and applied differentiated replication policies. According to the model, replication management modules are implemented based on LIME2. The experiment results showed that the read operation's latency is obviously reduced. However, a large penalty is incurred for writing operations. Therefore, applying differentiated replication policies is absolutely necessary, because no single policy is suitable for all cases.

Key words: Decentralization, tuple space, tuple distribution, replication management, performance, consistency

INTRODUCTION

Collaboration among heterogeneous agents, which can be used for providing autonomous and spontaneous solutions has been become popular. Many collaboration mechanisms and frameworks have been developed and researched. One of them, is the Tuple Space Model, which was first introduced by coordination language Linda (Gelernter, 1985), a classical paradigm for communications of multiple processes. Tuple space provides a multi-agent like architecture, where agents can collaborate through writing, reading or removing tuples in the space. Among various tuple space implementations, two most-known centralized tuple spaces are Sun JavaSpaces (Freeman, 1999) and IBM TSpaces (Wyckoff *et al*., 1998).

LIME (Murphy and Picco, 2004; Murphy *et al*., 2006) implements and extends this model to the decentralized environment. In LIME, local tuple spaces distributed on different nodes are transient shared as the abstract global one, whose contents depends on the connectivity of participating local tuple spaces. LIME allows both location specified and unspecified tuple access. Many other decentralized tuple space systems have been proposed, such as

Corresponding Author: Jiankuan Xing, Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China

SwarmLinda (Tolksdorf and Menezes, 2004; Graff *et al*., 2008), Peer Spaces (Busi *et al*., 2003) and TATO (Mamei and Zambonelli, 2005) etc. However, in these tuple spaces users cannot specify tuple's locations explicitly, which are internally managed by tuple space.

In spite of the convenience in development brought by decentralized tuple space, collaborative applications also require low access latency, which is hard for decentralized tuple spaces to afford because the execution latency of remote operations takes approximately two or three orders of magnitude more than the local ones. In LIME-like tuple space, tuples' locations are unchanged until deletion. Therefore, replication needs to be introduced to meet both the immutable location setting and the feasibility of dynamic distribution.

Replicable LIME (Murphy and Picco, 2006) has added a replication layer to LIME in meeting the needs of TULING (Murphy and Picco, 2004). Replicable LIME adopts profile-based manner to setup replication policies. However in our project, these profiles level is too low, only deciding where to get replicated and updated. We tend to bind replication and tuple space primitives together and improve replication's automation. Another shortcoming brought by Replicable LIME is resulting from its per-replication profile and per-reaction implementation. This is because of LIME's linear reaction matches implementation, which will incur excessive overhead when profile's number is not trivial. GSpace (Russello *et al*., 2005, 2007) adopts category-based replication policies to improve availability with self-adaptation. However, GSpace essentially uses centralized style: one Adaption Module on a host controls the replication policy of a single category of tuples distributed among all other hosts.

In this study, we concentrate on a replication-based tuple distribution approach to decentralized tuple space for performance improving. This study tends to be used in collaborative applications in the highly dynamic environment. Besides, the basic replication-related API, multiple replication policies are introduced to handle replication in different scenarios. We have implemented Replicable Decentralized Tuple Space (RDTS) upon LIME2 (Bellini, 2004) and have taken benchmark experiments among multiple hosts connected by LAN. The experiment results showed that RDTS greatly reduces the latencies of read-intensive scenarios; they also make it clear that when write and update operations are dominant, replication hurt the performance, which proves the necessity of multiple replication policies.

REPLICABLE DECENTRALIZED TUPLE SPACE MODEL

This section describes the model of replicable decentralized tuple space. We use formal statements similar to Klaim (De Nicola *et al*., 1998) to define the semantics of primitives of tuple space with replication.

Tuple Space Systems

A tuple space is an unordered container of tuples. A tuple is a data structure consisting of one or more typed fields each containing some value. A tuple template (template for short later), used for tuple matching, is the tuple that have some fields with null value as the typed wildcard. In the following sections, e is used to represent a tuple, $e^{[id]}$ indicates a tuple with a specified id and t represents a tuple template. If t matches e, we say e▹t. For detailed formal definition of tuples, templates and match (Gelernter, 1985; Murphy *et al*., 2006).

Decentralized Tuple Space Model

In our model, decentralized tuple space is a quintuple:

$$DTS=\{T,L,X,G,\gamma\} \tag{1}$$

where, tuples in the tuple set T are distributed among physical locations, expressed by location set L. X is the process set that are running on locations. In this study, we ignored the difference between physical locations and nodes located on them and do not care about logic location since, it is unrelated to our motivation. The local tuple space on location i is donated as LTS_i·γ∈P $(T\cup X)\times L$ represents the DTS configuration. Physical locations are connected by network G:

$$G=\{g_{ij}\},\ i,j\in L \tag{2}$$

g_{ij} = 1 means location i and j are engaged, g_{ij} = 0 means they are disengaged. We assumed the network is symmetric, that is, $g_{ij} = g_{ij}$. Engagement and disengagement indicate whether local context are merged as the global one.

Primitives of Decentralized Tuple Space Without Replication

The behavior in the DTS can be formalized as Eq. 3.

$$\begin{aligned} &X ::= nil \mid \pi.X \mid X_1 \parallel X_2 \\ &\pi ::= out_j^i(?e) \mid rdp_j^i(?t,!\tilde{e}) \mid inp_j^i(?t,!\tilde{e}) \mid up_j^i(?t,?e^*) \end{aligned} \tag{3}$$

where, X is the process executed by an agent and || parallel execution of two processes. Each action in the process has several parameters. Those marked by ? are the input parameters while ! are the output ones.

First, we define the non-blocking local lookup operation v_i (?t,! $\tilde{e}$) in LTS_i:

$$\nu_i(t,\tilde{e}) ::= \begin{cases} \nu_i(?t,!\tilde{e})[e/\tilde{e}], & \text{if } \exists e\in LTS_i \wedge e \rhd t \\ \nu_i(?t,!\tilde{e})[null/\tilde{e}], & \text{otherwise} \end{cases} \tag{4}$$

In this definition, LTS_i is the tuple set which is hold all the tuples (both masters and replicas) in the location i. v is the lookup operation with one input parameter t and one output parameter $\tilde{e}$. [e/ $\tilde{e}$] means the variable $\tilde{e}$ is substituted by e. In this manner, Eq. 4 means that if a matched tuple e is found, it is returned. When nothing matched is found, null is returned and we define {null} = Ø. The subscript i of v is the location where, lookup is executed. In the rest of study, one operation's subscript denotes where, it is executed and superscript represents where it is initiated.

Based on lookup there are four primitives out, rdp, inp and up supported by decentralized tuple space model, performing add, retrieval, take and update operation respectively. up is not one of primitives in the traditional tuple space. Because of the existence of replication, it is necessary to define an atomic update operation. The formal semantics of primitives are listed as follows:

$$out_j^i(?e) ::= LTS_j \oplus \{e\} \tag{5}$$

$$rdp_j^i(?t,!\tilde{e}) ::= \nu_j(?t,!\tilde{e}) \tag{6}$$

$$inp_j^i(?t) ::= \nu_j(?t,!\tilde{e}).LTS_j \ominus \{\tilde{e}\} \tag{7}$$

$$up_j^i(?t,?e^*) ::= rdp(?t,?\tilde{e}).rdp(?t,?\tilde{e}).LTS_j \ominus \{\tilde{e}\} \oplus \{e^*\} \tag{8}$$

Here, we use ⊕ and ⊖ to indicate adding and removing an element from the set. For example, in rdp^i_j, $LTS_i \ominus \{\tilde{e}\}$ means remove the tuple variant $\tilde{e}$ from LTS_i. And considering, $\tilde{e}$ is finally substituted by v, finally the lookup result is removed.

Replication Policies

As replicable LIME, we apply replication policies to groups of tuples. Rather than simply using a template to identify a certain group, our model explicitly adds a tuple category field to our replicable tuple because different tuple categories may be confused if their templates are identical. In the rest, we use C_e to indicate the category of e.

In our model, all local tuple spaces control their own replication policies. In detail, the policy applying is a per tuple category, per location action:

$$Pol_i(C) \in \{SO, RC, FR\} \tag{9}$$

which indicates applying a policy to category C on location i.

Because, the replication will incur excessive overhead in certain cases, our model adopts three different replication policies:

- **Store Originally (SO):** No replicas are made, equivalent to original LIME
- **Read and Cache (RC):** Create a replica when a remote read operation is initiated and no matched tuple (master or replica) is found locally
- **Full Replication (FR):** Create replicas for all new tuples written to the tuple space and push them to other local tuple spaces

For example, if $Pol_i(C_1) = RC$, LTS_i will create a replica in location i if a read operation is initiated and no matched tuple (master or replica) of C_1 can be found locally; if $Pol_i(C_2) = FR$, all other LTS will automatically push the tuples of C_2 to LTS_j once they are written. In this way, the policies on different locations do not influence each other; and the tuples with different categories can use different policies.

Decentralized Tuple Space Primitives with Replication

Here, the primitive's semantics under replication consideration are introduced. Our model uses e_r to express one replica of master tuple e and does not distinguish the replicas in different local tuple space. The location set where, e is replicated is defined as:

$$\mathcal{L}(e) = \{i \mid \forall i, e_r \in LTS_i\} \tag{10}$$

and the location set where a policy Pol (C) is adopts is represented as:

$$\mathcal{L}(\widetilde{Pol}, C) = \{i \mid \forall i, Pol_i(C) = \widetilde{Pol}\}, \widetilde{Pol} \in \{SO, RC, FR\} \tag{11}$$

where, $\widetilde{Pol}$ is a variant indicating a certain policy.

No matter which policy is applied, inp and up have to handle the replicas' take and update besides the masters:

$$inp^i_j(?t, !\tilde{e}) ::= v_j(?t, !\tilde{e}).LTS_j \ominus \{\tilde{e}\}.\left(\prod_{\forall k \in \mathcal{L}_T(\tilde{e})} LTS_k \ominus \{\tilde{e}_r\}\right) \tag{12}$$

$$up_j^i(?t,?e^*) ::= v_j(?t,!\tilde{e}).LTS_j \ominus \{\tilde{e}\} \oplus \{e^*\}.\left(\prod_{\forall k \in \mathcal{L}_T(t)} LTS_k \ominus \{\tilde{e}_t\} \oplus \{e_t^*\}\right) \quad (13)$$

In the above formulas, Π means the following actions are connected by ‖ and thus will be executed in parallel, as Eq. 14 indicates.

$$\prod_{1 \le k \le n} P_k = P_1 \| P_2 \| \cdots \| P_n \quad (14)$$

When Pol_i (C) = RC, rdp first tries to access the matched replica cached locally. If not found, it performs conventional remote retrieval operation and at last create new replica for the result if it is found:

$$rdp_j^i(?t,!\tilde{e}) ::= \begin{cases} v_i(?t,!\tilde{e}), & \exists e_t \in LTS_i \wedge e_t \rhd t \\ v_j(?t,!\tilde{e}).LTS_i \oplus \{\tilde{e}_t\}, & \exists e \in LTS_j \wedge e \rhd t \text{ or not found} \end{cases} \quad (15)$$

When Pol_i (C) = FR, edp can assume the there must be local replica which can be directly retrieved.

$$rdp_j^i(?t,!\tilde{e}) ::= v_i(?t,!\tilde{e}) \quad (16)$$

Meanwhile, the out operation will automatically push tuple replicas to anywhere that registered FR to the category that the written tuple belongs to.

$$out_j^i(?e) ::= LTS_j \oplus \{e\}.\left(\prod_{\forall k \in \mathcal{L}(FR,\mathcal{C}_e)} LTS_k \oplus \{e_t\}\right) \quad (17)$$

DESIGN AND IMPLEMENTATION

This section briefly describes the design and implementation of our RDTS system, which was built upon LIME2 and LighTS (Balzarotti *et al*., 2007). The RDTS managed to implement everything by use of tuples, reactions and replication itself, demonstrating their expressiveness and versatility.

Figure 1 shows the overall architecture of our RDTS system. Replication management modules were implemented in a multi-layers style. The Local Replication Manager is responsible of single-tuple replication and Replication Policy Manager above takes care of policies information of tuple categories. RDTS registers two reactions, Replication Policy Reaction and Command Reaction, which handle replication commands and trigger policy switch, to implement remote replication protocols. In RDTS, only the kernel of LIME is remained, including the local operation processor and communication protocols, to avoid the overhead of data format conversion. LIME and LighTS-based replica space together provide the local tuple space abstraction mentioned in the model. The key points of RDTS' design and implementation are described in the following sub-sections in detail.

Storage of Replicas

RDTS separates the spaces of master and replica tuples. All replicas are put into a private tuple space implemented by LighTS. This design is led by the following considerations:

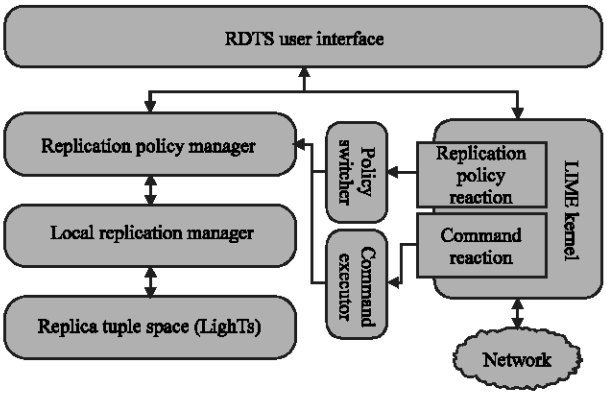

Fig. 1: The schematic of RDTS architecture. The round rectangles mean components; the cloud shape means the network; the arrowed lines mean data access relationship (A points at B means A get data from B)

- Access is accelerated if users specify what they required is a master or replica explicitly in template, since only part of tuples are looked up linearly. If not specified, the lookup process will take place first in replica tuple space; and if not found, the master tuple space is searched. In the worst case, the lookup is equivalent to the design which stores both replicas and masters in the same place. Meanwhile, the tuple field that marks whether a replica or not is not necessary any more
- Replication management is more convenient. With the separated replica tuple space, we can easily take statistics of categorized replica's counts and sizes, as well as replication operations counts and timing. This is useful for us to monitor the runtime status of replicas

Share Tuples and Signal Tuples

In Replicable DTS, tuples are classified as in two main categories: shared tuples and signal tuples. Shared tuples, as the name indicates, are used for data sharing and thus replicable. There are several meta-fields in a shared tuple to support replication:

$$\langle \text{tid}, \text{curr}, \text{dest}, \text{rid}, \text{cat}, \text{ver}, [\text{user fields}] \rangle$$

where tid, curr and dest are LIME meta-fields, which means tuple ID, current location and destination respectively. Rid is the replication layer ID. A master tuple and all its replicas are identical in this ID. Cat is the tuple's category while ver is the tuple's current version, which is started with 1 and monotone increasing.

Signal tuples, on the contrary, are used to transferred commands and messages among local tuple spaces. They are never actually written into the tuple space, only for reactions triggering purpose. The format of a signal tuple is shown below. The only field that needs attention is src, which indicates the source of this signal. Because, it is not replicable, fields like rid, xat and ver are no longer necessary.

$$\langle \text{tid}, \text{curr}, \text{dest}, \text{src}, [\text{user fields}] \rangle$$

Shared and signal tuples are designed for generic use of collaborative applications built upon RDTS. For example, shared tuples can be used for sharing of documents, images or

context information, while signal tuples are suitable for instant messages, task assignments or beacon-style self-identification. We also adopt these two basic tuples in RDTS' implementation of replication management and protocols, as described in the following two sub sections.

Replication Policy Tuples

Replication policy tuple (RPTuple for short), implemented as a shared tuple, represents a replication policy. RPTuple's fields are shown below:

$$\langle[\text{shared tuple meta-fields}], \text{tcat}, \text{tloc}, \text{pol}\rangle$$

tcat and tloc indicates the target category and location where, the policy pol is applied. Local tuple space on each location maintains its own replication policy to certain categories while others need to know other's policies and thus, do something necessary (such as push replicas when FR is applied). Therefore, if users want to set up a replication policy, they need only to create a new RPtuple and out it into DTS.

The broadcast of policies are realized by replication itself. That is, RPtuple itself is applied FR on each location. When, two LTS first engaged, they exchange so-called RP^2 tuple:

$$\langle[\text{shared tuple meta-fields}], \text{RPTuple}, \text{self}, \text{FR}\rangle$$

RP^2 tuple's transfer will be handled by the replication policy reaction, which triggers exchange of all RPTuples and starts up further replication related operations.

Command Tuples

In RDTS, remote replication-related operations are implemented by a series of command tuples, based on signal tuples. Two benefits are implicitly achieved. First, failures during command transfer can be recovered through LIME's reconciliation mechanism, which is able to automatically synchronize the inconsistencies as soon as it is possible. Also, tuple commands can be handled by reactions, which guarantee commands' atomicity and independence. In RDTS, we define five command tuples for replication protocols, which are listed in Table 1.

These commands are self-explanatory by their names. The second field of each command marks its command type as a literal string. It is worthy note, that AddReplicaRequestCommand and RemoveReplicaRequestCommand only add and remove replica requests, not replicas themselves. For example, if LTS_i has just retrieved a tuple e from LTS_j and want its replica, LTS_i replicate e by itself and out an AddReplicaRequestCommand

Table 1: Command Tuples in RDTS

Command name	Tuple format
AddReplicaRequestCommand	⟨[signal tuple meta-fields],"Add",rids,evs⟩
RemoveReplicaRequestCommand	⟨[signal tuple meta-fields],"Rem",rids⟩
ClearReplicaCommand	⟨[signal tuple meta-fields],"Clr",rids⟩
UpdateReplicaCommand	⟨[signal tuple meta-fields],"Upd",newTuples⟩
PushReplicaCommand	⟨[signal tuple meta-fields],"Psh",newTuples⟩

tuple to LTS_j, which then makes a record on LTS_i that LTS_j just made a replica of e. Similarly, RemoveReplicaRequestCommand only removes the record on the LTS where, master tuple is hold.

The field evs in AddReplicaRequestCommand is an array of excepted versions. It is used for the local tuple space that handles the replication request to decide whether to send back replica or not. In more detail, consider the case that an AddReplicaRequestCommand tuple initialized by LTS_i is sent to LTS_j, attached by rids and evs. The replicas $\{e_r^{[ridk]} | \forall ev_k = -1, 1 \le k \le |evs|\}$ will be returned back through PushCommand, where ev_k and rid_k mean the k-th excepted version in evs and k-th rid in rids, respectively. In this case -1 is a special value, which means LTS_i wants to acquire replica tuples from LTS_j. If LTS_i has already owned the replica (for instance, retrieves a tuple and make its replica), evs contain the version numbers that LTS_i expects LTS_j has. If the replication source tuple's actual version is greater than the corresponding expected version, LTS_j immediately update those replicas. In other words, in this scenario replicas $\{e_r^{[rid_k]} | \forall ev_k < ver^{[rid_k]}, 1 \le k \le |evs|\}$ are sent back to LTS_i through UpdateCommand.

EXPERIMENT RESULTS AND ANALYSIS

Here, we evaluate the performance of RDTS. We deployed RDTS over 12 nodes which run Windows XP SP2 and JDK 1.6 update 10 and were equipped with 3.06 GHz dual core Pentium D and 1G memory and connected via 100 Mbit LAN.

The experiment measured the execution performance of tuple space operations under replication protocols. We implemented operation generating agents which periodically initialize operations including rdp, out and up. As what the above model indicated, only non-blocking operations were taken into account in order to avoid introducing the interference of registering and deregistering the reactions by the blocking operations. Figure 2 shows the deployment schematic of these agents together with RDTS middleware in the experiments, which mimicked a typical collaboration scenario. In each node (or physical location), 3 to 6 agents run concurrently. Parts of nodes were responsible of tuple writers and updaters, while others were pure tuple readers. The writers and updaters manipulated the tuples in the local tuple space only and the readers performed the remote read operations.

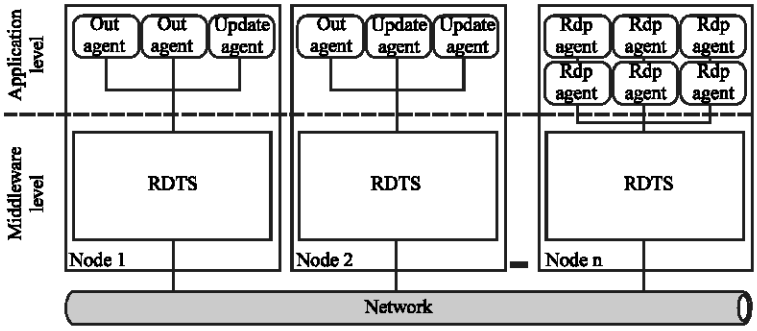

Fig. 2: The deployment schematic of experiment. The rectangle means a physical node; the rectangles with RDTS label mean RDTS middleware; round rectangles mean agents; the pipe shape means the network; the lines mean the data channels which connect agents, middleware and the network

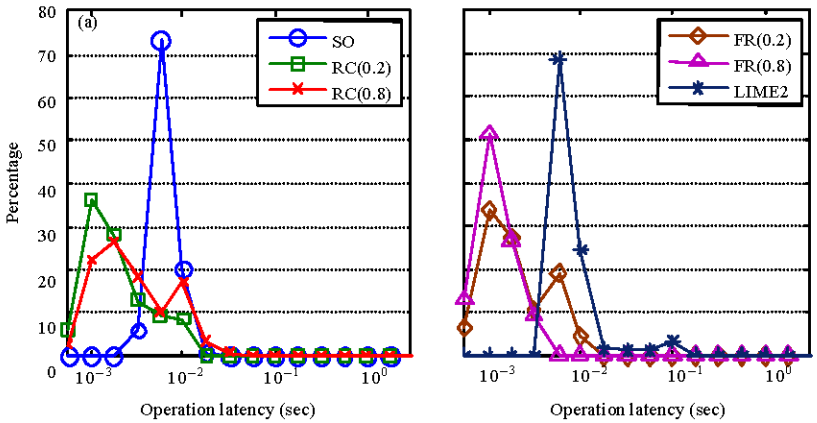

Fig. 3: (a, b) rdp latency distribution with 10 kB tuple size. The x-axis means the operation latency distribution intervals, stepped by $10^{0.25}$ sec in logarithmic scale, the y value with the x value 10^{n} means the percentage of operations whose latency is less or equal than 10^{n} sec and greater than $10^{n-0.25}$ sec

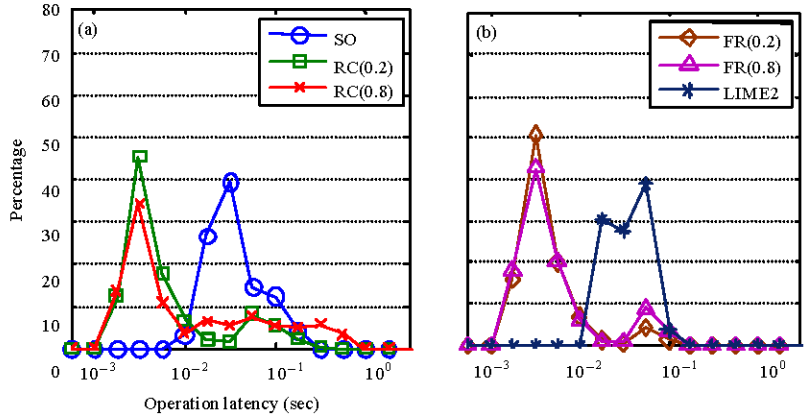

Fig. 4: (a, b) rdp latency distribution with tuple size 500 kB. The x-axis means the operation latency distribution intervals, stepped by $10^{0.25}$ sec in logarithmic scale, the y value with the x value 10^{n} means the percentage of operations whose latency is less or equal than 10^{n} sec and greater than $10^{n-0.25}$ sec

Operation Latency under Replication

The results of operation latency can be grouped by tuple size, ranged among 10, 50, 100 and 500 kB. We only present 10 and 500 kB in this section for conciseness. The data is illustrated as the operation latency distribution among various time intervals that users perceive with logarithm x-axis, from 10^{-3} to 10^{0} sec.

Figure 3a and b to 6a and b show the latency distribution for rdp, out and up, with different replication policies and tuple sizes. For RC and FR, we employ the concept Miss Rate (MR), which are shown as the bracketed value after policy in the legends of these figures. MR means the expected proportion of no local replica found scenarios. Therefore, the higher MR value implies more actual remote operations.

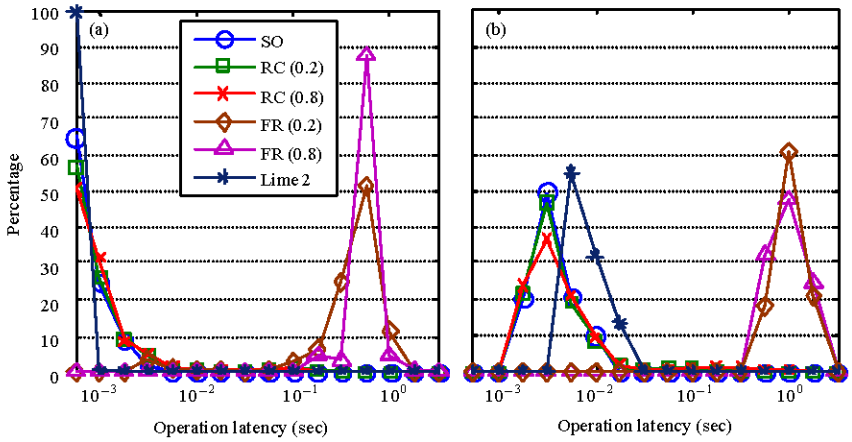

Fig. 5: (a, b) out latency distribution. The x-axis means the operation latency distribution intervals, stepped by $10^{0.25}$ sec in logarithmic scale, the y value with the x value 10^{n} means the percentage of operations whose latency is less or equal than 10^{n} sec and greater than $10^{n-0.25}$ sec

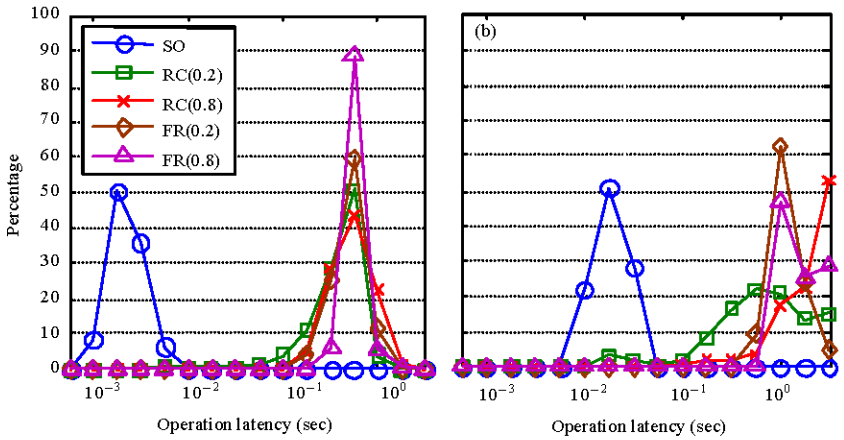

Fig. 6: (a, b) up latency distribution. The x-axis means the operation latency distribution intervals, stepped by $10^{0.25}$ sec in logarithmic scale, the y value with the x value 10^{n} means the percentage of operations whose latency is less or equal than 10^{n} sec and greater than $10^{n-0.25}$ sec

Figure 3 and 4 show the experiment results of rdp latency distribution. Considering the mess, if the distribution plots of rdp with various replication policies were drawn on the one figure, we separated them into Fig. 3 and 4. The latency distribution line of SO policy behaves very similar to the one of LIME2. This confirms that our implementation does not introduce conspicuous overhead when no replicas get involved. When RC or FR is applied, the line's peak locates at $10^{-2.75}$ sec in the 10 kB case and $10^{-2.5}$ sec in the 500 kB case. Comparing the distribution lines for FR and RC, FR owns the similar latency to the RC ones since, they are all actually locally executed. And FR achieves a higher local replica access count, as Fig. 7 showed, because when tuples are written, they are pushed in advance.

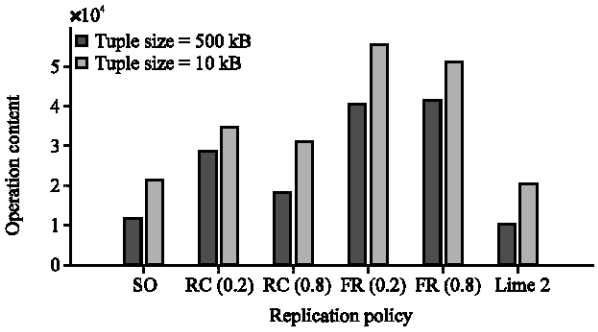

Fig. 7: The count of rdp with various replication policies. The x-axis means different replication policy while the y-axis means the operations' count that are executed in the same period time

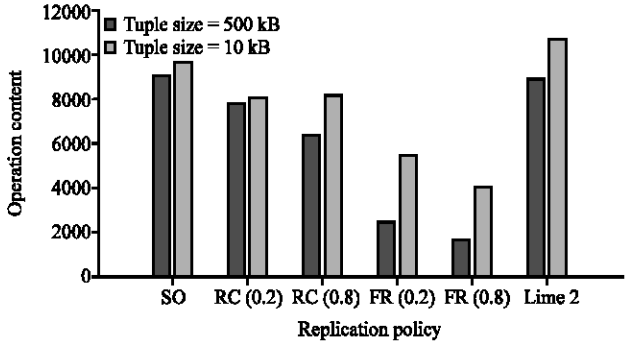

Fig. 8: The count of out with various replication policies. The x-axis means different replication policy while the y-axis means the operations' count that are executed in the same period time

MR also affects the performance of rdp under RC. In Fig. 3 and 4, lines for RC with MR 0.2 indicate that RC has more opportunities to hit local replicas and thus handle more remote rdp locally compared to the ones with MR 0.8. However, lines for FR show that FR wasn't affected by MR because a replica should be at local node, no matter whether it had been read before or not.

Figure 5 shows local out's latency distributions. As per the model specified earlier, out behaves in the same way with SO and RC, proved by the lines whose dominated peaks stand in the same position. The optimization we did for transferring large tuples works is shown by the 500 kB out line in Fig. 5. However, when FR is applied, local out is responsible for pushing all tuples to other nodes who registered FR. In the current implementation, a simple one-by-one push is adopted because application-level multicast is beyond this paper's purpose. In the 10 kB case, approximate 300 m sec~1 sec was consumed to finish this job; in the 500 kB, the main latency was beyond 1 sec. These data indicate that FR will greatly hurt the performance of local out when lots of tuple needs to be transferred. Figure 8 also proves the performance penalty that the out count with FR is highly decreased.

Figure 6 gives the latency distributions of up. As the distribution lines shown in Fig. 6, the dominated latency for SO stands at $10^{-2.75}$sec in the 10 kB case and $10^{-1.75}$sec in the 500 kB case. This is because up with SO is purely a local operation. In other policies where,

replication is introduced, the dominated latency of up lays at $10^{-0.25}$ sec in the 10 kB case and 10^{0}-$10^{0.75}$sec in the 500 kB case. In these cases up pushes those tuples of new versions one by one to other nodes who add replica request. Figure 5 and 6 together expose that a small increment of out and up will incur larger penalty.

DISCUSSION

The experiment results showed that the replication's effects on the typical operation's performance. The measurements confirmed that the impact of replication management module itself is negligible compared to the original LIME2. Replication essentially moves part of the burden from tuple readers to writers. Therefore, if the read operations are far more than writers (out or up), replication successfully reduced latency. Collaborative applications may gain many benefits from it since, the collaborative algorithm requires less time to finish the communication. However, when writers are dominant, replication brings large overhead. In this case, replication should be shutdown.

Many tuple space systems used for data sharing assumes that the data written are immutable. LIME also adopts this paradigm. In this kind of systems, keeping the replica-master link is unnecessary and update is no longer needed. However, lots of data keep changing in collaborative applications, for which we cannot make that assumption. An alternative strategy is to explicitly classify mutable and immutable data and only manage mutable ones with full replication support. Our replication management actually follows this strategy and separates tuple classification and replication policy classification. One certain replication policy applying to one certain tuple category controls the burden balance between readers and writers. Under different scenarios, users can choose the most suitable replication policy.

Another worthy concern is the implementation of tuple push. Because, the network layer of LIME2 currently only support sending tuple to one target, we simply beyond it implemented a one-by-one push, which was not scalable since, it is related to the number of push targets. The more targets, the more time is spent on tuple sending. Even worse, the push initiator cannot decide how many to push. This contradicts the RDTS' original intention: every node is responsible of controlling itself. In this perspective, a push targets-unrelated multicast is absolutely necessary before RDTS is ready for practical use. A degree-limited tree-based multicast is an ideal approach to improve our system.

CONCLUSIONS AND FUTURE WORK

In this study, we made the following contribution for RDTS. First, by introducing tuple lookup and remote transfer, we exposed the detailed actions taken by RDTS non-blocking primitives. We also provided a simple replication management mechanism and added it into DTS; and defined the primitive's behavior under three replication policies. The result was that different replica distributions were achieved according to policies to leverage different characteristics in operation latencies.

Second, we implemented a complementary replication management component following RDTS model and hided replication under tuple space primitive operations. Shared and signal tuples were used as the generic foundation for all collaboration usage, beyond which RPtuple and command tuples were created and employed in replication information maintenance and interactive protocols. Furthermore, RPtuple's flooding was also done by replication itself. From the above, this implementation demonstrated that RDTS model's venerability in building collaborative applications.

Finally, we conducted the experiments and measured the benefits in rdp while, overhead in out and up in different replication policies. The conclusion was that the gain of enabling replication overcomes the overhead if proportion of reads and writes reaches certain threshold.

The experiments showed two improvement points on which further points we will concentrate on:

- Configuring the replication policy for each tuple category is a tight job because it may be relevant to the dynamic changed running status of tuple space, such as the proportion of reads and writes as well as the replica miss rate. These parameters can be monitored or calculated within tuple space and enable running of a self-adaptive replication manager. Our model and experiments' results could be the foundation to define a set of adaption rules
- Simple one-by-one application-level multicast incurs too much performance overhead, so it is necessary to employ a more efficient multicast structure, such as a dynamic mesh with degree limited multicast tree

As part of our future work, we plan to implement Decentralized Replicable Adaptive Tuple Space for Collaboration Work, or DracoTS, a middleware for decentralized collaboration applications, such as collaborative task execution of detection range-limited Unmanned Aerial Vehicle (UAV) without central command. The challenge of DracoTS is to make tradeoff between providing a collection of easy-to-use API which allows users to avoid thinking about the internal of RDTS and leaves a configuration interface for fine-grained tuning for performance.

ACKNOWLEDGMENTS

This study is part of simulator of decentralized UAV (Unmanned Aerial Vehicle) collaboration project and is supported by National Defense Pre-research Plan of China (No. 402040202); and many thanks to the review of Amanda Dotson, in Applied Physics Department of Physics, University of Maryland, Baltimore County.

REFERENCES

Balzarotti, D., P. Costa and G.P. Picco, 2007. The LighTS tuple space framework and its customization for context-aware applications. J. Web Intell. Agent Syst., 5: 215-231.

Bellini, L., 2004. Lime II: Reengineering a Mobile Middleware. Politecnico di Milano, Milano, Italy.

Busi, N., C. Manfredini, A. Montresor and G. Zavattaro, 2003. PeerSpaces: Data-driven coordination in peer-to-peer networks. Proceedings of the ACM Symposium on Applied Computing, Melbourne, Florida, March 09-12, ACM, New York, USA., pp: 380-386.

De Nicola, R., G.L. Ferrari and R. Pugliese, 1998. Klaim: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng., 24: 315-330.

Freeman, E., 1999. JavaSpaces Principles, Patterns and Practice. Pearson Education, New Jersey, ISBN: 0201309556.

Gelernter, D., 1985. Generative communication in linda. ACM Trans. Programm. Languages Syst., 7: 80-112.

Graff, D., R. Menezes and R. Tolksdorf, 2008. On the performance of swarm-based tuple organization in LINDA systems. Proceedings of the IEEE World Congress on Computational Intelligence, (CCI'08), IEEE Press, Hongkong, China, pp: 2709-2716.

Mamei, M. and F. Zambonelli, 2005. Spatial Computing: The TOTA Approach. Springer, Berlin, pp: 307-324.

Murphy, A.L. and G.P. Picco, 2004. Using Coordination Middleware for Location-Aware Computing: A Lime Case Study. In: Coordination Models and Languages, De Niconla, R. *et al*. (Eds.). LNCS. 2949, Springer-Verlag, Berlin, Hamburg, ISBN: 978-3-540-21044-3, pp: 263-278.

Murphy, A. and G.P. Picco, 2006. Using Lime to Support Replication for Availability in Mobile Ad Hoc Networks. In: Coordination Models and Languages, Ciancarini, P. and H. Wiklicky (Eds.). LNCS. 4038, Springer-Verlag, Berlin, Heidelberg, ISBN: 978-3-540-34694-4, pp: 194-211.

Murphy, A., G.P. Picco and G.C. Roman, 2006. LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Software Eng. Methodol., 15: 279-328.

Russello, G., M. Chaudron and M. van Steen, 2005. Dynamically Adapting Tuple Replication for Managing Availability in a Shared Data Space. Springer, Berlin, pp: 109-124.

Russello, G., M.R.V. Chaudron, M. van Steen and I. Bokharouss, 2007. An experimental evaluation of self-managing availability in shared data spaces. Sci. Comput. Prog., 64: 246-262.

Tolksdorf, R. and R. Menezes, 2004. Using Swarm Intelligence in Linda Systems. Springer, Berlin, Heidelberg, pp: 49-65.

Wyckoff, P., S.W. McLaughry, T.J. Lehman and D.A. Ford, 1998. T spaces. IBM Syst. J., 37: 454-474.