International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:4, 2010

# Exploiting Self-Adaptive Replication Management on Decentralized Tuple Space

Xing Jiankuan, Qin Zheng, and Zhang Jinxue

***Abstract*—**Decentralized Tuple Space (DTS) implements tuple space model among a series of decentralized hosts and provides the logical global shared tuple repository. Replication has been introduced to promote performance problem incurred by remote tuple access. In this paper, we propose a replication approach of DTS allowing replication policies self-adapting. The accesses from users or other nodes are monitored and collected to contribute the decision making. The replication policy may be changed if the better performance is expected. The experiments show that this approach suitably adjusts the replication policies, which brings negligible overhead.

***Keywords*—**Decentralization, Replication Management, Self Adaption, Tuple Space.

## I. Introduction

TUPLE Space model, which was first introduced by coordination language Linda[1], is a classic paradigm for communications of multiple processes. Tuple space provides a multiagent like architecture, where agents can collaborate through writing, reading or removing tuples in the space. Tuple space's generative features, the referential and temporal decoupling between processes, provide promising foundations for collaboration in the high dynamic, unstable environments. Initially, tuple space systems are implemented in the centralized style, as the original model implied. The typical cases are Sun JavaSpaces[2] and IBM TSpaces[3].

LIME[4, 5] is a representative implementation of Decentralized Tuple Space (DTS). In LIME, Local Tuple Spaces (LTS) distributed on different nodes are transient shared as the abstract global one, whose contents change over time, depending on the connectivity of participating local tuple spaces. LIME allows both location specified and unspecified tuple access. Tuple accesses with specified destination location avoid unnecessary transversal of nodes if the upper applications know where to access the tuple.

However, in spite of the benefits brought by the DTS, applications also require low access latency, which is difficult for DTS because the execution latency of remote operations takes approximately two or three orders of magnitude more than the local ones. This defect may pay off the advantages of decentralization.

We have developed a replication management approach[6] to solve this problem, where differentiating replication policies were employed to meet the needs of different tuple access patterns. And we observed that it was difficult to manually configure the suitable replication policy. On the one hand, it is hard to estimate the runtime status of DTS; on the other hand, the tuple access pattern may change over time. Therefore, it is necessary to exploit the self-adaptive approach to make DTS can switch the policy automatically.

This paper concentrates on the self-adaptive replication management of DTS for the collaborative applications among a small scale of decentralized nodes. This paper is outlined as follows: Section II briefly describes the model of DTS; Section

This paper is supported by National Defense Pre-Research Plan of China (402040202).

Xing Jiankuan is with the Department of Computer Science and Technology, Tsinghua University, Beijing, China P. R. (corresponding author to provide phone: 86-13581568806; fax: 86-010-62795399; e-mail: xjk05@mails.thu.edu.cn).

Qin Zheng is with the Department of Computer Science and Technology, Tsinghua University, Beijing, China P. R. (e-mail: qingzh@mail.tsinghua.edu.cn).

Zhang Jinxue is with the School of Software, Tsinghua University, Beijing, China P. R. (e-mail: zhang-jx08@mails.tsinghua.edu.cn).

III presents the detailed replication management protocols; Section IV describes the criterion values, rules and actions used in the adaption; Section V describes the setup of experiments and analyze the results; Section VI briefly introduces the proposed related work to our proposal; And Section VII concludes this paper and gives the future research directions.

## II. Decentralized Tuple Space

### A. Tuple Space

A tuple space can be defined as an unordered collection of tuples. A tuple is a structure consisting of one or more typed field, such as $\langle 1, \text{"abc"}, 2.3\rangle$. The field with the concrete value is called *actual*. A tuple template is a tuple that is used for matching tuple. A tuple template often contains one or more typed wildcard field, called *formal*. For example, the template $\langle 1, \text{string?}, 2.3\rangle$ matches any tuple whose first field is 1, third field is 2.3, and second field is any string value. A template can contain no *formals* at all, which means the matched tuple must be exactly the same as the template.

There are four primitives defined in the original tuple space model: **out**, **in**, **rd**, and **eval**. **out** is to write a tuple into the space. **in** and **rd** both specify a template and indeterminately get a matched tuple. If no matched tuple can be found, these two primitives (and thus the processes which invoke them) will block until some matched ones are **out** into the space. The difference between **in** and **rd** is that **in** will delete the gotten tuple from the space whereas **rd** not. **eval** is similar to **out**. But **eval** writes an active tuple, which will start a new process after the tuple writing. **eval** was omitted by many literatures since it introduces executable data. And this paper also omits it. In the extended tuple space models, some variants of original primitives are introduced. Such as **inp** and **rdp**, the probing version of **in** and **rd**, return **null** immediately if no matched tuple is found rather than block.

### B. Decentralized Tuple Space

Decentralized tuple space expands traditional tuple space into the decentralized environment. In a typical scenario of decentralization, there are a set of nodes participating in the tuple space, denoted as $\mathcal{A} = \{A_1, \ldots, A_{|\mathcal{A}|}\}$, none of which is responsible of coordinating others. The network is denoted by $\mathcal{G} = \{g_{ij}\}$, whose element indicates whether node $A_i$ and $j$ are able to connect ($g_{ij} = 1$) or not ($g_{ij} = 0$). And $\mathcal{N}_i$, the neighbor node set for node $A_i$, can be defined as $\mathcal{N}_{A_i} = \{A_j | g_{ij} = 1 \vee (\exists k, A_k \in \mathcal{N}_{A_i} \wedge A_k \in \mathcal{N}_{A_j})\}$. We use $\mathcal{N}_i$ to represent $\mathcal{N}_{A_i}$ in the rest of this paper. Therefore, the tuple space seen by the node $A_i$ at one moment is the transient union of LTSes hold by $A_i$ and $\mathcal{N}_i$.

When node $A_i$ initiates primitives, it should explicitly specify the destinations. For example, $\mathbf{out}_j^i(e)$ means the tuple $e$ is initiated by node $A_i$ and written into the LTS of node $A_j$. $\mathbf{rd}_{DTS}^i(t)$ retrieves a certain tuple which matches template $t$ in the entire tuple space area. And $\mathbf{in}_{\mathcal{N}_i}^i(t)$ deletes the tuple matching $t$ in node $A_i$'s neighbor. Because of the restrictions of connectivity, the first two cases above may be temporally unable to execute, but the model guarantees that these primitives are sent to the destination once the connections between source and destination nodes are available.

## III. Replication Management

### A. Replication and Consistency

Any tuple can have one or more replicas located at different nodes. To mark the replication relationship, the master tuple and all its replicas share the identical ID. A tuple's master and replica will not exist at the same node simultaneously. All tuples are versioned, and the version number is monotone increasing along the update done by the extended **up** primitive.

Considering the decoupling of time and space, the global consistency model is not adopted in DTS; otherwise mutual exclusive access is inevitable and distributed transactions are bound to be involved. This betrays purely decentralized semantics because that no node should play the special role of transaction manager; and no guarantee could be made that unannounced disconnection would not happen during a transaction. Moreover, it hurts performance in that the transaction will block all other primitives' processing among the entire DTS.

Therefore, a node-level consistency model is employed. When a replica is created in node $A_j$ from the source tuple at node $A_i$ (the source tuple may be the master or another replica), a consistency link is generated and tracked both in $i$ and $j$ for this tuple. When **up** is initiated and sent to the specified destination, if a matched tuple is found, master or replica, it gets updated and propagates the update through consistency links to

other nodes. During update process, no global atomicity is involved. And the update is realized in a "best effort" style. To achieve this consistency model, a protocol has been designed and implemented in [6].

### B. Replication Policies Management

To control the replicas' distribution and overhead incurred by consistency, we introduce Replication Policies. A replication policy is the rule that decides how a replica should be created. In the current system, three replication policies are defined.

- **SO (Store Originally)** means that no replicas are made at all. All the primitives are executed where the destination indicated.
- **RC (Read and Cache)** means that when remote **rd** series primitives are initiated, it first check whether some matched local replica exists. If not, the primitive is executed as normal, but create a replica automatically when the retrieve returns. The new replica is cached for the following reads.
- **FR (Full Replication)** means all the tuples are replicated in all the nodes in DTS. Therefore, no remote reads are necessary. Meanwhile, all the tuples written anywhere and all updates must be pushed to all other nodes if connection is available.

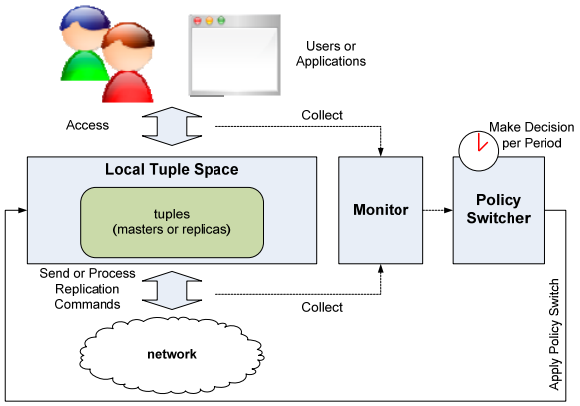


Fig. 1 The schematics of replication policies' adaption

The application of replication policies is denoted as $P_A^C$, where $A$ is the node on which the policy is applied; $C$ is a tuple category, which consists of a group of tuples which share the similar access pattern. Each tuple belongs to one and only one category. This triple defines exactly when and where the replicas are created. For example, $P_i^{C_1} =$ **FR** means that the tuples in DTS belonging to the category $C_1$ should make the full replication to the node $A_i$. This application also implies that differentiating policies can be configured to the same tuple category for different nodes.

TABLE I
SYMBOLS USED IN THE ADAPTION RULES

| Symbol | Meaning |
|---|---|
| $o_A^C(i)$ | The total write number of tuples of category $C$ in node $A$ until the $i$-th period. |
| $r_A^C(i)$ | The total read number of tuples of category $C$ in node $A$ until the $i$-th period. |
| $u_A^C(i)$ | The total update number of tuples of category $C$ in node $A$ until the $i$-th period. |
| $d_A^C(i)$ | The total deletion number of tuples of category $C$ in node $A$ until the $i$-th period. |
| $\tilde{r}_A^C(i)$ | The total read number of replicas of category $C$ in node $A$ until the $i$-th period. |
| $n_A^C(i)$ | Estimation of the total number of tuples of category $C$ in node $A$ in the $i$-th period. |
| $\hat{n}_A^C(i)$ | The total number of *used* replicas of category $C$ in node $A$ in the $i$-th period. |
| $\gamma_A^C(i)$ | The number of replicas of category $C$ in node $A$ converts from unused to used in the $i$-th period. |
| $\mu_A^C(i)$ | The total number of *unused* replicas of category $C$ in node $A$ in the $i$-th period. |
| $\varphi_A^C(i)$ | The total number of *used* replicas of category $C$ in node $A$ in the $i$-th period. |
| $P_A^C(i)$ | The replication policy applied in the node $A$ for $C$ in the $i$-th period |

## IV. Self Adaption of Replication Policies

### A. Overview of Self Adaption

We separate the adaption system into three parts, as shown in Fig. 1. The local tuple space is responsible of handling access requests from users or upper applications, and also sending or processing the replication-related commands initiated in other nodes. All these requests essentially can be considered as reads or writes to the tuples. These data are the collected by the monitor as the basis for the policy switch. After collection, some criterion values are calculated. These data' symbols are listed in TABLE I, and their usage and calculation is illustrated in the following equations. Periodically, according to these data, the policy switcher makes the decision and thus executes the switch process.

TABLE II

ACTIONS TAKEN IN POLICY SWITCH OF TUPLE CATEGORY $C$ IN NODE $A$

| From | To | Action |
|---|---|---|
| **FR** | **SO** | Remove all the replicas of $C$ in $A$. |
| **FR** | **RC** | Remove all the *unused* replicas of $C$ in $A$. |
| **RC** | **SO** | Remove all the replicas of $C$ in $A$. |
| **RC** | **FR** | Create replica links for all tuples of $C$ in $A$ which hasn't yet made replication. |
| **SO** | **RC** | Do nothing. The replicas will be created when they are first read. |

We divide the adaption into two phases, no replication (**SO**) and how to replicate (**RC** or **FR**), as shown in Fig. 2. The switch decision maker first decides whether the replication should be enabled. Then if it is true, gives the further decision to figure out which replication policy is better. The following two sub sections give the more detailed description of how the decision is made. Notably the direct SO to FR switch is not allowed according to the above two-phase decision making. After the final decision is made and the replication policy must be changed, the switch action is then executed as TABLE II shown.

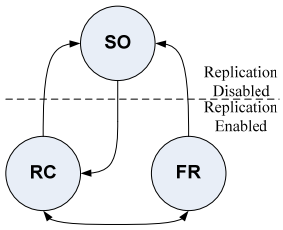


Fig. 2 Switch of replication policies

### B. Decide Whether Enable or Disable Replication

Whether to enable or disable replication is decided by the proportion of reads and writes to the tuples of a certain category. In the $i$-th period, the monitor of node $A_k$ collects $r_k^C(i)$, $\hat{n}_k^C(i)$, and fetches $o_A^C(i)$ and $d_A^C(i)$ from all neighbors, $\forall A \in \mathcal{N}_k$. Then all the proportion of tuple reads and writes $p_k^C$ for each category $C$ are calculated as (1).

$$p_k^C(i) = \frac{\alpha \cdot \frac{r_k^C(i)}{\hat{n}_k^C(i)}}{\alpha \cdot \frac{r_k^C(i)}{\hat{n}_k^C(i)} + \beta \cdot \frac{\sum_{\forall A \in \mathcal{N}_k} u_A^C(i)}{\sum_{\forall A \in \mathcal{N}_k} n_A^C(i)}} \quad (1)$$

As (1) shown, we care about the average read and write counts per replica instead of the total ones. $\alpha$ and $\beta$ are the weights which are subject to $\alpha + \beta = 1$.

$n_A^C(i)$ is not directly collected but estimated by (2). This is to avoid the mess of read and write counting during data collection.

$$n_A^C(i) = n_A^C(i-1) - \left(d_A^C(i-1) - d_A^C(i-2)\right) + \left(o_A^C(i) - o_A^C(i-1)\right) \quad (2)$$

To prevent the jitter which may incur oscillating switch, we employ $p$'s mobile average $\bar{p}$ and square deviation $\sigma(p)$ as the criterion values for decision making. The mobile average $\bar{p}$ is the unweighted mean of $p$ in pervious $D$ periods, as (3).

$$\bar{p}_k^C(i) = \left(\sum_{m=i-D}^{i} p_k^C(m)\right)/D \quad (3)$$

$$\sigma(p_k^C(i)) = \sqrt{\left(\sum_{m=i-D}^{i} (\bar{p}_k^C(i) - p_k^C(m))^2\right)/D} \quad (4)$$

Predefine three thresholds $\theta_{SO}$, $\theta_{REP}$ ($\theta_{SO} < \theta_{REP}$) and $\theta_{MSE}$. If (5) is satisfied, $P_k^C(i+1) = \mathbf{SO}$.

$$P_k^C(i) \in \{\mathbf{RC}, \mathbf{FR}\} \wedge \bar{p}_k^C(i) < \theta_{SO} \wedge \sigma(p_k^C(i)) \leq \theta_{MSE} \quad (5)$$

Otherwise, if (6) is satisfied, the decision maker will further decide whether to choose RC or FR, illustrated in the next subsection.

$$P_k^C(i) = \mathbf{SO} \wedge \bar{p}_k^C(i) > \theta_{REP} \wedge \sigma(p_k^C(i)) \leq \theta_{MSE} \quad (6)$$

### C. Decide whether to Apply RC or FR

The reason for automatically creating replicas for all the tuples of a tuple category and pushing their updates (**FR**) is that most of these actions are useful. In other words, if the replicas are actually *used*, we tend to keep using **FR**. In this observation, we separate the created replicas as two kinds, *used* or *unused*. For the node $A_k$, the *used* replicas are the ones who are read after creation by the primitives with the destination $A_k$, whereas the other replicas are the *unused* ones. To make the decision, we care about the proportion of becoming *used* replicas and the total replicas in one period, $q_k^C$.

$$q_k^C(i) = \frac{\gamma_k^C(i)}{\gamma_k^C(i) + \mu_k^C(i)} \quad (7)$$

In (7), $\mu_k^C(i)$ cannot be directly monitored. Instead, it is calculated by (8).

$$\mu_k^C(i) = \left(\sum_{\forall A \in \mathcal{N}_k} n_A^C(i)\right) - \varphi_k^C(i) \quad (8)$$

Also for oscillating avoidance, $q_k^C(i)$'s mobile average $\bar{q}_k^C(i)$ and square deviation $\sigma(q_k^C(i))$ for the previous $D$ periods are generated, similar as (3) and (4).

We then predefine another two thresholds $\theta_{RC}$ and $\theta_{RC}$

($\theta_{RC} < \theta_{FR}$). If (9) is satisfied, $P_k^C(i+1) = \mathbf{FR}$; if (10) is hold, $P_k^C(i+1) = \mathbf{RC}$; otherwise, the replication policy remains unchanged.

$$P_k^C(i) = \mathbf{RC} \wedge \bar{q}_k^C(i) > \theta_{FR} \wedge \sigma(q_k^C(i)) \le \theta_{MSE} \quad (9)$$

$$P_k^C(i) = \mathbf{FR} \wedge \bar{q}_k^C(i) < \theta_{RC} \wedge \sigma(q_k^C(i)) \le \theta_{MSE} \quad (10)$$

## V. Experiment and Analysis

### A. System Implementation and Experiment Setup

We have implemented the adaption modules upon LIME2[7], a simplified and reengineered edition of LIME, and LighTS[8], a light weight tuple space for single host. The architecture is shown in Fig. 3. We created the Replication Manager (RM) between the tuple space interface and LIME2, in which the Adaption Controller (AC) worked as the daemon along the tuple space. AC consisted of Monitor, Decision Maker and Policy Switcher, which worked together to adapt the replication policies managed by Replication Policy Manager. Tuples in the single host were stored in the Local Tuple Space, which were actually separated into two LighTS instances. The separation made convenient the monitoring the access patterns on replicas and masters respectively.

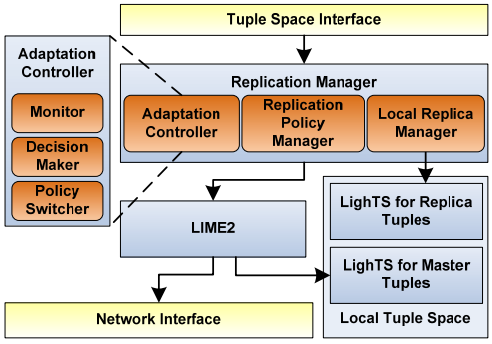


Fig. 3 Architecture of DTS with self-adaptive replication policy switch

The experiment was running a series of simulation with changing tuple access patterns upon several nodes. We deployed our adaptive DTS on 10 nodes, each of which run one or two agents, responsible of reading or writing (including out and update), as Fig. 4 shown. Each node was equipped with 3.06GHz dual core Pentium D, 1G memory, and JDK 6 update 10. These nodes were connected via 100Mbit LAN.

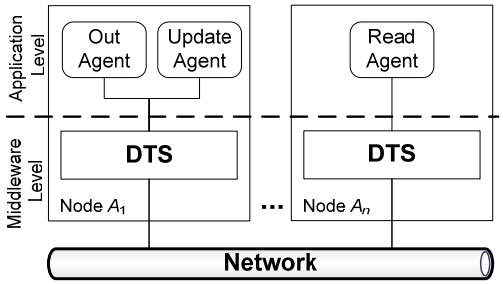


Fig. 4 Experiment deployment diagram

### B. Self Adaption

In this experiment, we showed the effect of self adaption by simulating the changing tuple access pattern. This was achieved by adjust agents' parameters, which is listed in TABLE III. These settings were inspired by the log output of a collaborative application using our DTS. The Read Agent had two parameters, *interval* and *prob*. *interval* defined the interval between two reads operations' start. And *prob* indicated the expected proportion to read the tuple that is not replicated locally. Out Agent and Update Agent also had *interval*, same as Read Agent. Besides, they had parameters $\mu$ and $\sigma$ to control the tuple size they generate, which follows normal distribution. In the experiment described here, the tuple size parameters were set as $\mu = 5000$ bytes and $\sigma = 200$ bytes. There were parameters about adaption, which were set as that $D = 4$ and period length is 30 seconds. Each experiment length was 30 minutes, and thus contained total 60 periods. We started to record the data from the fourth period.

TABLE III
Parameter Setting to Simulate the Changing Access Pattern

| Read Agent | Out Agent | Update Agent |
|---|---|---|
| Period: 1~15<br>*interval*=500ms<br>*prob*=0.2 | Period: 1~3<br>*interval*=500ms | Period: 1~12<br>*interval*=1000ms |
| Period: 15~30<br>*interval*=125ms<br>*prob*=0.8 | Period: 3~12<br>*interval*=1000ms | Period: 12~18<br>*interval*=2500ms |
| Period: 30~45<br>*interval*=250ms<br>*prob*=0.2 | Period: 12~18<br>*interval*=1500ms | Period: 18~36<br>*interval*=1000ms |
| Period: 45~60<br>*interval*=1500ms<br>*prob*=0.2 | Period: 18~36<br>*interval*=500ms<br>Period: 36~60<br>*interval*=2000ms | Period: 36~60<br>*interval*=250ms |

The experiment results are shown in Fig. 6 and Fig. 7. In these two figures we give the average operation executing time (optime) in each period. For comparison, we also give the data

with adaption turned off and applied **SO** and **FR** constantly. The dash line indicates the point where replication policy switches occurred. The results show that the self-adaption will switch after 3~10 periods when access patterns were changed.

For example, in the 3rd period, Out Agent slowed down to write new tuples, whose *interval* was changed from 500ms to 1000ms, while Read Agent kept reading each 500ms and only approximate 20% the tuples it read were the new ones. Therefore, in the 8th period, when this change was steady, the Decision Maker decided to use **RC** to decrease the time spent by remote read.

In the 15th period, the Read Agent changed to read tuples more frequently (*interval*=125ms), while the Out Agent and Update Agent performed writes in slower rate than before (out interval=1000ms and update interval=2500ms). In this observation, in the 20th period, **FR** is applied. The other switch cases can be explained similarly and meet the expectation of adaption design.

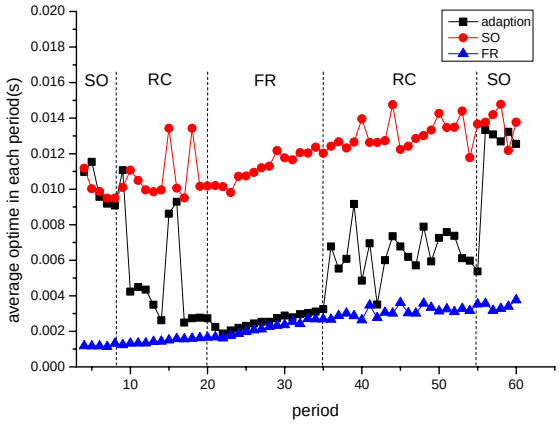


Fig. 5 Read operation executing time along periods

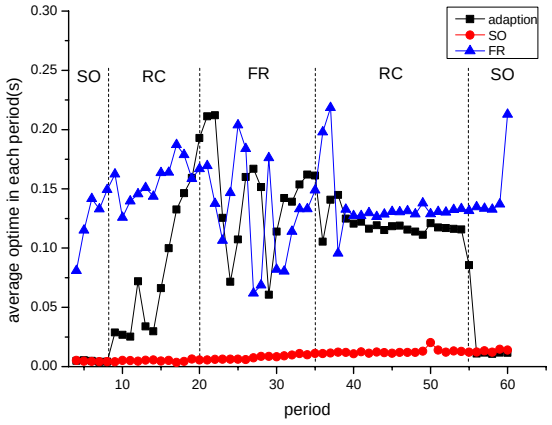


Fig. 6 Write operation executing time along periods

### C. Overhead

The adaption needs to take extra resource, time and bandwidth. Since our design is to monitor the data for each tuple category, we took the experiment to show the relationship between category number and overhead incurred by self-adaption.

Fig. 7 shows the operation executing time when adaption was turned on. Considering 20 categories are beyond the requirement of most applications using DTS we have experienced, we use 20 as the upper bound of this experiment. The figure illustrates that when the category number is growing, no affects to the conventional operation executing can be observed.

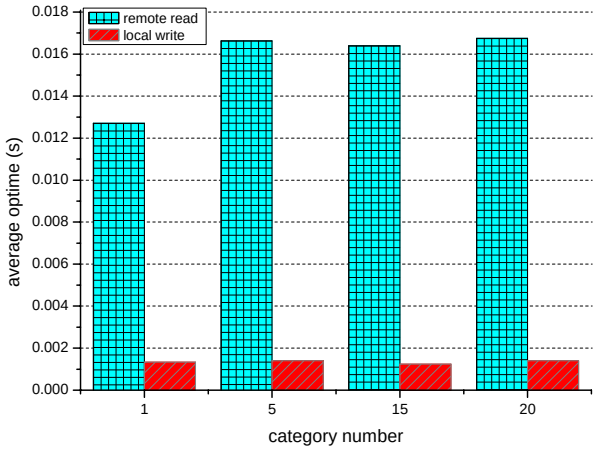


Fig. 7 Write operation executing time along periods

And we also measured the bandwidth taken by the exchange information to facilitate the adaption. Each 30 seconds, a package of size 200bytes were exchanged for each tuple category between two nodes. In the worst case, 10 interconnected nodes are running the DTS and each of them contains all 20 categories of tuple. Therefore, total 180KB data needs to be transmitted each 30 seconds, which can be accepted in conventional LAN or WAN. However, the current design is not suitable for the large scale of nodes.

## VI. Related Work

The initial effort on replicable tuple space can be traced to the early [9] and [10].

Murphy *et al.* proposed Replicable LIME in [11] to support the location-aware collaborative application TULING[5].

Replicable LIME adopts profile-based manner to setup replication, deciding where to get replicated and updated. Our work, however, pay more attention to how frequent the replicas get updated and how to decrease the footprint brought by update.

GSpace [12, 13] was proposed to promote performance and availability by replication, sharing the similar goals as what is presented here. But GSpace actually uses centralized management. One Adaption Module on a node controls the replication policy of a single category of tuples distributed among all other nodes.

PeerSpaces[14] was designed for the P2P resource sharing, in which all the published data are immutable. Therefore, no consistency was concerned in its replication management.

## VII. Conclusion

In this paper we proposed an approach of self-adaptive replication management for DTS. The adaption effectively avoids the problem that it is difficult for users to predefine the replication policies for tuple categories. The policies now can be automatically adjusted according to the reads and writes of tuples to achieve better performance. And the overhead that spend on time and bandwidth is negligible if the group is small.

In the future research, we tend to take other factors into the decision of replication policy self-adaption besides tuple read and writes, such as memory taken, bandwidth taken and availability. We also need a better switch execution implementation to avoid the negative effects brought by the burst of replicas' creation because of the policy switch. We hope to release the developers of applications built upon DTS from thinking over the issues of tuple storing and transmission and make them concentrate on the collaboration itself.

## References

[1] Gelernter, D., "Generative communication in Linda", *ACM T Progr Lang Sys*, 1985, 7(1), pp. 80-112.

[2] Freeman, E., Hupfer, S., and Arnold, K., *JavaSpaces Principles, Patterns, and Practice* (Pearson Education, 1999).

[3] Wyckoff, P., McLaughry, S.W., Lehman, T.J., and Ford, D.A., "T spaces", *Ibm Syst J*, 1998, 37(3), pp. 454-474.

[4] Murphy, A.L., Picco, G.P., and Roman, G.C., "LIME: A coordination model and middleware supporting mobility of hosts and agents", *ACM Transactions on Software Engineering and Methodology* 2006, 15(3), pp. 279-328.

[5] Murphy, A.L., and Picco, G.P., 'Using coordination middleware for location-aware computing: A LIME case study': *Coordination Models and Languages, Proceedings* (Springer Berlin, 2004), pp. 263-278.

[6] Jiankuan, X., Zheng, Q., and Jinxue, Z., "A Decentralized Tuple Space Model with Policy Management for Collaboration", *Chiese Journal of Electronics*, 2010 (Accepted, estimated published in 2010.4).

[7] Bellini, L., "Lime II: Reengineering a mobile middleware", Master Thesis, Politecnico di Milano, Italy, 2004.

[8] Balzarotti, D., Costa, P., and Picco, G.P., "The LighTS tuple space framework and its customization for context-aware applications", *Web Intell. Agent Syst.*, 2007, 5(2), pp. 215-231.

[9] Xu, A., and Liskov, B., "A design for a fault-tolerant, distributed implementation of Linda", in *The Nineteenth International Symposium on Fault-Tolerant Computing*, pp. 199-206.

[10] Bakken, D.E., and Schlichting, R.D., "Supporting Fault-Tolerant Parallel Programming in Linda", *IEEE T Parall Distr*, 1995, 6(3), pp. 287-302.

[11] Murphy, A.L., and Picco, G.P., "Using LIME to support replication for availability in mobile ad hoc networks", in *Coordination Models and Languages, Proceedings* (Springer-Verlag Berlin, 2006), pp. 194-211.

[12] Russello, G., Chaudron, M., and van Steen, M., "Dynamically adapting tuple replication for managing availability in a shared data space", *Coordination Models and Languages, Proceedings* (Springer-Verlag Berlin, 2005), pp. 109-124.

[13] Russello, G., Chaudron, M.R.V., van Steen, M., and Bokharouss, I., "An experimental evaluation of self-managing availability in shared data spaces", *Sci. Comput. Program.*, 2007, 64(2), pp. 246-262.

[14] Busi, N., Montresor, A., and Zavattaro, G., "Data-driven coordination in peer-to-peer information systems", *Int J Coop Inf Syst*, 2004, 13(1), pp. 63-89.